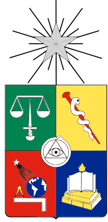


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MODEL-BASED SYSTEMATIZATION OF SOFTWARE ARCHITECTURE DESIGN

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS
MENCIÓN COMPUTACIÓN

DANIEL PEROVICH GEROSA

PROFESOR GUÍA:
MARÍA CECILIA BASTARRICA PIÑEYRO

MIEMBROS DE LA COMISIÓN:
GIOVANNI GIACHETTI
SERGIO OCHOA DELORENZI
LEONARD BASS

Este trabajo ha sido parcialmente financiado por CONICYT y NIC Chile

SANTIAGO DE CHILE
AGOSTO 2014

Resumen

La Arquitectura de Software juega un rol crucial en la Ingeniería de Software, permitiendo el control intelectual, la integridad conceptual, la comunicación efectiva, la administración de un conjunto relacionado de variantes de sistemas, y la reutilización de conocimiento, experiencia, diseño e implementación. Aplicar el conocimiento arquitectónico promueve la calidad, reduce los riesgos, y es esencial para alcanzar las expectativas de los interesados con resultados predecibles. El conocimiento arquitectónico actual es vasto y está en constante aumento, pero a su vez, es heterogéneo y disperso, está expresado en diferentes niveles de abstracción y rigor, y requiere de herramientas que raramente están disponibles en los ambientes de desarrollo. En la práctica, el diseño arquitectónico está limitado por las habilidades y experiencia del arquitecto y por el conocimiento que domina, y requiere de gran esfuerzo para ajustarlo y adaptarlo al escenario de desarrollo. Así, el diseño arquitectónico rara vez alcanza el nivel de calidad que es posible dado el conocimiento arquitectónico disponible. Además, el esfuerzo del arquitecto no es repetible ya que resultan embebidos en las descripciones de las arquitecturas.

Aunque las técnicas de modelado están siendo usadas en Arquitectura de Software, la mayoría de los enfoques carecen de generalidad y homogeneidad, dificultando su integración y aplicación. En este trabajo, usamos megamodelado para definir un mecanismo unificado y homogéneo para capturar conocimiento arquitectónico, haciéndolo compartible, reusable, manejable por herramientas, y directamente aplicable. Definimos una interpretación formal de los conceptos principales de la disciplina en términos de artefactos de modelado. Además, cambiamos el foco de construir la descripción de la arquitectura directamente, a capturar cómo dicha descripción es creada. Para ello, definimos un lenguaje para capturar las acciones de diseño, y lo interpretamos en términos de técnicas de modelado haciendo el diseño repetible. Validamos nuestro enfoque definiendo procedimientos para guiar a la comunidad en cómo capturar conocimiento arquitectónico usando nuestra interpretación formal, aplicando estos procedimientos para capturar las técnicas de descripción y diseño del SEI, y aplicando el conocimiento capturado al diseño de la línea de productos de mallas geométricas.

Nuestro trabajo realiza dos contribuciones originales. Primero, definimos un mecanismo unificado y homogéneo para capturar conocimiento arquitectónico, usando técnicas de Ingeniería Dirigida por Modelos, particularmente el enfoque de megamodelado Global Model Management, y usando semántica denotacional para la formalización. Segundo, definimos una representación de decisiones y soluciones arquitectónicas en términos de un lenguaje específico, haciéndolas descriptivas y aplicables. Así, facilitamos el cambio de foco del arquitecto haciendo el diseño arquitectónico explícito, repetible y reusable, y obteniendo descripciones de arquitectura implícitas y generables en forma automática.

Abstract

Software Architecture plays a pivotal role in Software Engineering, allowing the realization of intellectual control, conceptual integrity, effective project communication, management of a set of related variants of systems, and an adequate basis for reuse of knowledge, experience, design and implementations. Applying architecture knowledge promotes quality, reduces risks, and is crucial to best meet stakeholders' expectations with predictable results. Current architecture knowledge is vast and ever-growing, however, it is also heterogeneous and disperse, is expressed at different levels of abstraction and rigor, and requires varied tool-support that is rarely available in an integrated development environment. In practice, architecture design is bounded by the architect's skills, experience, and the subset of knowledge he masters, and it also requires a huge effort to adjust and apply such knowledge to the development scenario. Thus, architecture design hardly achieves the potential level of quality that is possible given the available architecture knowledge. Moreover, the architect's effort is not repeatable as it is implicitly embedded in the architecture descriptions.

Although modeling techniques are being used in Software Architecture, most approaches lack either generality or homogeneity, making it hard to integrate, adapt and apply them. In this work, we use megamodeling to provide a unified and homogeneous means for capturing architecture knowledge, making it shareable, reusable, tool-friendly, and directly applicable during architecture design. We formally define a mapping from key architecture concepts to interrelated modeling artifacts. Also, we shift the focus from capturing the architecture description itself, to capturing how such a description is created and updated. To this end, we define an architecture design scripting language to capture fine-grained design actions, and we map them to modeling techniques, making architecture design repeatable. We validate our approach by defining step-wise procedures to guide practitioners on how to capture architecture knowledge using our formal mapping, by applying these procedures to capture SEI's techniques to architecture description and design, and by applying the captured knowledge to address the architecture design of the Meshing Tool Software Product Line.

The original contribution of our work is twofold. First, we define a unified and homogeneous means for representing architecture knowledge using Model-Driven Engineering techniques, particularly the Global Model Management approach to megamodeling, and using a denotational semantics approach for the formalization. Second, we define a first-class representation of architecture decisions and solutions in terms of an architecture design scripting language, making them not only descriptive but most importantly applicable. Thus, we facilitate a shift in the enactment of architecture design making it explicit, repeatable and reusable, while rendering architecture descriptions implicit and automatically generated.

To my parents.

Acknowledgements

As I type these words, my thesis is coming to an end. It is a moment of strong feelings and deep reflection, of gratitude and fulfilment. This moment seemed far distant in late 2007 when I first drafted the thesis proposal. Since then, even though I have always felt certain that this moment would eventually come, sometimes along the way it felt more than distant, practically unattainable. This thesis is the result of significant effort and time, not only mine but also from many others that I was fortunate enough to have them walking by my side along this way, making the experience both enjoyable and enriching. I owe all of them a debt of gratitude.

First and foremost, I would like to express my deep appreciation and gratitude to my advisor, María Cecilia Bastarrica. I am extremely grateful for the time she dedicated to my project, for her guidance and support, for making considerable efforts to make it possible for me to attend conferences, to take the internships, and to find funding for my studies, for her collaboration and for repeatedly reviewing this manuscript, for trusting in me and my work, and for finding the balance between leeway and pressure in her effort to impel me to finish my thesis. Cecilia, this work would not have been possible without your patience, encouragement and wisdom. Thank you!

I would like to thank the reviewers of my thesis proposal, Leonard Bass, João Araújo and Éric Tanter, whose insightful comments and suggestions helped me set the scope and focus of my work. I would also like to thank the members of my jury, Leonard Bass, Sergio Ochoa and Giovanni Giachetti. Their comments and revisions of this manuscript were valuable and gratifying, and kindly contributed to improve my work and to place it as a step of a larger stairway. My sincere apologies for making this manuscript too long.

I am grateful to the Computer Science Department of the University of Chile that has given me the opportunity to make my PhD studies possible. I would like to thank the coordinators of the PhD Program in Computer Science, specifically José Pino for accepting me into the program, Claudio Gutiérrez for his continuous encouragement and effort on convincing me that I had enough valuable work to report long before I realized and acknowledged it, and Éric Tanter for urging me to successfully finish this project. I am also thankful to the former and current members of the research team that I belong to. To María Cecilia Bastarrica for her vision in creating such a team, to Andrés Vignaga and Pedro Rossel for all the time we dedicated to prepare for courses and exams, and for the productive discussions that led to several publications and helped each of us shape our theses. To Julio Hurtado and Alcides Quispe for the fruitful collaboration. To Cristian Rojas for letting me participate

in his undergraduate thesis and apply the foundational ideas of my work to his case study. To Sergio Ochoa for many fruitful conversations and for his encouragement. To Nancy Hitschfeld for her time and for sharing her knowledge on the mesh domain and on meshing tool development. To Magaly Zúñiga, Sandra Gaez, and specially Angélica Aguirre, for being always willingly helpful and supportive. Most importantly, I thank them all for making these past years at the Computer Science Department not only productive, but enjoyable!

I would like to thank CONICYT Chile and NIC Chile for providing each a generous grant for a considerable part of the program to fund my studies, and to the FONDEF Project D09I1171 that funded my last year of studies. I am also grateful to the Continuing Education Program of the Computer Science Department of the University of Chile, mainly to its chief coordinator Christian Bridevaux, for the numerous courses I lectured for this program and hence indirectly help me funding my studies. Also, to the CONICYT-INRIA (Chile-France) Hot Mate project that funded my internship at the AtlanMod group from École des Mines de Nantes. I am grateful to Jean Bézivin for receiving me in his group, for the many fruitful conversations, and for his insight in the potential of my work. Also, to Frédéric Jouault, Hugo Bruneliére, and the whole AtlanMod team at that time for the discussions, for their assistance in the AM3 toolset, and for making my stay in Nantes a pleasant one. Also, I would like to thank to Don Batory for attending to my talk during my internship, and for his productive comments and insight. Also, I would like to thank the University of Chile for completely funding my internship at the Software Engineering Institute of the Carnegie Mellon University. I am grateful to Jörgen Hansson and Jorge Andrés Díaz Pace for receiving me in their group, for the time they dedicated to discuss my work, its liabilities, benefits and applications, and for their directions for further improvement. I am extremely grateful to Leonard Bass and Felix Bachmann for taking the time to meet with me during the internship to discuss my work, for their sharp insight and for their vision on best suited application scenarios. I am very grateful to Jorge Andrés and his wife for making me feel welcome during my stay in Pittsburgh. Also, I am grateful to Cristian Bravo, his wife Verónica Achá and their lovely daughter Almendra, for selflessly taking me into their home while I found accommodation in Pittsburgh, and for making me feel as part of the family. Thank you so much! I would also like to thank Bedir Tekinerdoğan and Henry Muccini for their kind comments on how to present my work and their suggestions on further lines of research, while attending WICSA 2014. I am also grateful to the Chilean software company Imagen, mainly to its CEO Juan Huircalaf, for trusting in me and my work to reify his vision of a reusable line of products, making me part of it from 2010 to 2012. Although the day to day urgencies prevented us from fully applying my model-based approach to Software Product Line development, I am pleased that we achieved a feature-based line of Enterprise Content Management products for the Microsoft SharePoint platform, and that we put into production several products of this family which it is still evolving. Also, I am grateful to Imagen's COO Andrés Vignaga and the whole team at Imagen, for making this experience enriching, gratifying and very pleasant.

I would like to thank Andrés Vignaga. We shared the same dream of a PhD degree and to this end we came to Chile. Our friendship started long before, and has been a great support for achieving such goal far from Uruguay, our home. I would also like to thank Carlos Burmester, Verónica Noguéz, and their children Carlos and Joaquín, for making me feel as part of their family for many years since my arrival to Chile. My warmest feelings

to my family and friends in Uruguay. Although the geography and the daily routine have been trying to make us apart, they have only make my love and appreciation stronger. I specially dedicate this work to my grandmother Celeste, to my parents Nelson and Irma, to my brother Ernesto and my beautiful niece Moriana. Thank you all for your unconditional love and support, for being together and always present to each other. I will always thank you! Last but most important, I thank Paula, my partner in life and the greatest gift this country has ever given me. She and her family, which I have felt mine too for many years now, have been of great support. Paula, you have been walking by my side bringing joy and hope, you have been the path and the rising sun at the horizon. I am grateful. I love you!

Contents

1 Introduction **1**
1.1 Context . . . 3
1.2 Problem Statement . . . 7
1.3 Research Hypothesis . . . 9
1.4 Research Goals . . . 9
1.5 Developed Solution . . . 10
1.6 Methodology . . . 16
1.7 Contributions . . . 18
1.8 Structure of the Thesis . . . 21

2 Background **25**
2.1 Software Architecture . . . 27
2.1.1 The Software Architecture Discipline . . . 28
2.1.2 Software Architecture Design . . . 32
2.1.3 Software Architecture Description . . . 34
2.2 Model-Driven Engineering . . . 36
2.2.1 The Modeling Discipline . . . 37
2.2.2 Modeling Constructs . . . 39
2.3 Global Model Management . . . 51
2.3.1 Conceptual Framework & Metamodel . . . 53
2.3.2 Realization & Tool Support . . . 64

3 Model-Based Software Architecture Description **69**
3.1 Standard on Software Architecture Description . . . 71
3.1.1 Architecture Descriptions . . . 74
3.1.2 Architecture Frameworks & Languages . . . 79
3.2 Model-Based Architecture Description . . . 83
3.2.1 Definition of the semantic functions . . . 85
3.2.2 Semantics of Architecture Descriptions . . . 104
3.2.3 Semantics of Frameworks and Languages . . . 146
3.2.4 Communicating Architecture Descriptions . . . 152
3.3 Contributions & Discussion . . . 156

4 Model-Based Software Architecture Design **161**
4.1 Concepts of Software Architecture Design . . . 164
4.1.1 Architecture Patterns, Tactics & Styles . . . 171
4.1.2 Architecture Update Statements . . . 184
4.1.3 Architecture Solutions, Decisions & Rationale . . . 202

4.2 Model-Based Architecture Design . . . 207
4.2.1 Shifting Focus from Description to Design . . . 209
4.2.2 Definition of the semantic functions . . . 212
4.2.3 Semantics of Architecture Design . . . 221
4.2.4 Traceability . . . 245
4.3 Contributions & Discussion . . . 247

5 Model-Based Software Product Line Architecture 253
5.1 Architecture in Software Product Lines . . . 255
5.1.1 Variability . . . 261
5.1.2 Product Line Architecture . . . 267
5.1.3 Product Architecture . . . 273
5.2 Model-Based Architecture in Software Product Lines . . . 275
5.2.1 Model-Based Approach . . . 277
5.2.2 Semantics of Product Line Architecture Design . . . 286
5.3 Contributions & Discussion . . . 292

6 Applying the Model-Based Approach 295
6.1 Capturing Architecture Knowledge . . . 297
6.1.1 Architecture Description Knowledge . . . 298
6.1.2 Architecture Design Knowledge . . . 308
6.1.3 Architecture Design Methods . . . 312
6.2 Meshing Tool Software Product Line . . . 317
6.2.1 Mesh & Meshing Tools . . . 318
6.2.2 Designing the Product Line Architecture . . . 321
6.2.3 Deriving Product Architecture Descriptions . . . 327

7 Conclusions and Further Work 329
7.1 Research Thesis . . . 330
7.2 Contributions & Lessons Learned . . . 333
7.3 Perspectives & Further Work . . . 336

Bibliography 339

List of Figures

1 Introduction **1**

1.1 Abstract model for a software architecture design process. 10
1.2 Iterative & incremental nature of architecture design 11
1.3 Model-based architecture description and design approach 13
1.4 Structural organization of the developed solution 14
1.5 Participants of the developed solutions 15
1.6 Structure of the core of this dissertation 21
1.7 Structure of the main chapters of this dissertation 22

2 Background **25**

2.1 Abstract model for a Software Architecture Design Method. 33
2.2 3+1 organization of the metamodeling approach. 44
2.3 Model weaving. 47
2.4 Model transformation. 49
2.5 Megamodel metamodel : Elements. 54
2.6 Megamodel metamodel : Entities. 56
2.7 Example of the constitution of a megamodel. 57
2.8 Example of a megamodel containing a megamodel. 57
2.9 Megamodel metamodel : Relationships. 58
2.10 Megamodel metamodel : Model weavings. 59
2.11 Megamodel metamodel : Transformations. 60
2.12 Megamodel metamodel : Model transformations. 62
2.13 Megamodel metamodel : External transformations. 63
2.14 Megamodel metamodel : Mixed transformations. 64
2.15 AM3 support to the GMM approach to megamodeling. 67

3 Model-Based Software Architecture Description **69**

3.1 Context of the software architecture description practice 73
3.2 Conceptual model of architecture descriptions 75
3.3 Conceptual model of correspondences 79
3.4 Conceptual model of architecture frameworks 81
3.5 Conceptual model of architecture description languages 82
3.6 Conceptual model traversal for semantic function compositionality 105
3.7 The Three-Forces Approach to Architecture Description 154
3.8 Model-Based Communication Force 157

4 Model-Based Software Architecture Design 161
4.1 Context of the software architecture design practice 165
4.2 Abstract model for a Software Architecture Design Method. 167
4.3 Iterative & Incremental Architecture Design 168
4.4 Exploration of the Solution Space at Architecture Design 169
4.5 Interaction of architecture patterns and architecture tactics 177
4.6 Conceptual model of architecture patterns, tactics and styles 181
4.7 A design step as architecture update statements 186
4.8 Conceptual model of architecture update scripts and statements 187
4.9 Conceptual model of the Start statements 188
4.10 Conceptual model of the Identify statement 190
4.11 Conceptual model of the Use statements 191
4.12 Conceptual model of the CreateArchitectureModel statements 194
4.13 Conceptual model of the CreateCorrespondence statement 195
4.14 Conceptual model of the ImproveDocumentation statements 196
4.15 Conceptual model of the Constrain statements on architecture models 197
4.16 Conceptual model of the Constrain statements for correspondence rules . . . 198
4.17 Conceptual model of the ApplyStyle statement 199
4.18 Conceptual model of the ApplyPattern statement 200
4.19 Conceptual model of the ApplyTactic statement 201
4.20 Conceptual model of ArchitectureSolution, ArchitectureDecision and ArchitectureRationale 205
4.21 Example of denotation of StartFromArchitectureDescription statement 232

5 Model-Based Software Product Line Architecture 253
5.1 Software Product Line Development Process 259
5.2 Metamodel for Feature Models 262
5.3 Conceptual model of variability management for core assets 265
5.4 Context of the product line architecture practice 272
5.5 Context of the product architecture practice 274
5.6 Conceptualization of Variability in Product Line Architecture Descriptions . 281
5.7 Conceptual model of Product Line Architecture Description 283
5.8 Conceptual model of the StartFromProductLineArchitectureDescription statement 287

6 Applying the Model-Based Approach 295
6.1 Metamodel of the Module Architecture Viewpoint 306
6.2 Metamodel of the Component & Connector Architecture Viewpoint 307
6.3 Metamodel of the Client/Server Model Kind 308
6.4 Metamodel of a style specializing the Client/Server style 313
6.5 Steps of the Attribute-Driven Design Method 314
6.6 Feature Model for Meshing Tools 323
6.7 Product Architecture Description (fragment) Concerning Algorithms 326
6.8 Product Architecture Description of a Non-Distributed Mesh 327
6.9 Product Architecture Description of a Distributed Mesh 328

List of Tables

3 Model-Based Software Architecture Description **69**
3.1 Syntactic domain of the semantic functions 88
3.2 Semantic domain: model repositories 90
3.3 Semantic domain: conforms-to, is-kind-of and requires assertions 92
3.4 Semantic domain: model fragments 93
3.5 Semantic domain: representation-of functions 96
3.6 Semantic domain: descriptors of non-entity elements 99
3.7 Semantic functions 100
3.8 Semantic equations: Concern 107
3.9 Semantic equations: Stakeholder 110
3.10 Semantic equations: ModelKind 112
3.11 Semantic equations: ArchitectureModel 120
3.12 Semantic equations: ArchitectureViewpoint 122
3.13 Semantic equations: ArchitectureView 127
3.14 Semantic equations: Correspondence 131
3.15 Semantic equations: CorrespondenceRule 135
3.16 Semantic equations: ArchitectureDescription 138
3.17 Semantic equations: Architecture and SystemOfInterest 146
3.18 Semantic equations: ArchitectureFramework 148
3.19 Semantic equations: ArchitectureDescriptionLanguage 150

4 Model-Based Software Architecture Design **161**
4.1 Patterns, Tactics & Styles 180
4.2 Syntactic domain of the semantic functions 214
4.3 Semantic domain of the semantic functions 215
4.4 Semantic operators $\mathcal{T}_{\mathsf{tmm}}$ for the transformation engines 218
4.5 Semantic equations: ArchitecturePattern 223
4.6 Semantic equations: ArchitectureDecision 227
4.7 Semantic equations: ArchitectureSolution 228
4.8 Semantic equations: ArchitectureUpdateScript 229
4.9 Semantic equations: StartFromScratch statement 230
4.10 Semantic equations: StartFromArchitectureDescription statement 231
4.11 Semantic equations: StartFromReferenceArchitecture statement 233
4.12 Semantic equations: Identify statement 234
4.13 Semantic equations: UseArchitectureFramework statement 235
4.14 Semantic equations: CreateEmptyArchitectureModel statement 238
4.15 Semantic equations: ImproveDocumentationOfArchitectureView statement . . 239

4.16 Semantic equations: SelectCorrespondenceRule statement 240
4.17 Semantic equations: ApplyPattern statement 242
4.18 Semantic equations: CallScript statement 243

5 Model-Based Software Product Line Architecture 253
5.1 Conceptualization of the post-order traversal of required features 285
5.2 Semantic equations: ProductLineArchitectureDescription 289
5.3 Semantic equations: StartFromProductLineArchitectureDescription statement . 291

Chapter 1

Introduction

Software-intensive systems have pervaded our daily lives, from personal electronic gadgets, to house appliances, transportation, the work environment, and the management and operation of organizations of any scale. The constant evolution of customer markets, the increasing demand for highly customized products and services, and the heterogeneity in the user preferences, in their localization and mobility, in the capabilities of devices, and in the supporting software, hardware and networking platforms, have triggered an exponential growth in the complexity and variability of modern software systems. Competition and customers are demanding software development projects to run on tight budget and time-to-market restrictions, but keeping the high-quality expectations on the software systems. As a consequence, appropriately balancing the cost-time-quality equation is a critical success factor for development projects. Software development is hard, and as the size and complexity of the software systems gets greater, the task of developing them gets exponentially harder. A software development project involves many stakeholders, spans for several months and requires numerous multi-disciplined teams. Also, it requires the construction of dozens of software artifacts, hundreds of components and database tables, thousands to millions of lines of code, and production environments running several computers. Failing to deal with this complexity yields to over-budget projects, late deliveries, and unacceptable levels of quality, and as a consequence, to low customer satisfaction and lower to none return of investment.

Software Engineering is the engineering discipline focused on the effective development of high-quality software systems. It encompasses the systematic, disciplined, and quantifiable application of scientific, industrial, and technological knowledge, methods and experiences, to the construction of software systems. It comprises all aspects of software development, from the early stages of requirement elicitation, system design, implementation and validation, to the late stages of deployment, operation, maintenance and evolution. Software development companies adopt Software Engineering practices to make their processes conveyable and repeatable, to structure their teams, to organize the development effort along the development projects, to manage changes and risks, and to standardize languages, techniques and tools for building software artifacts and systems.

Software Architecture plays a pivotal role in Software Engineering. It allows customer organizations to meet their business goals by enabling them to acquire systems that successfully

Chapter Contents

1.1 Context 3
1.2 Problem Statement 7
1.3 Research Hypothesis 9
1.4 Research Goals 9
1.5 Developed Solution 10
1.6 Methodology 16
1.7 Contributions 18
1.8 Structure of the Thesis 21

satisfy the required functionality within the expected levels of quality. It allows development companies to realize intellectual control, conceptual integrity, effective project communication, management of a set of related variant systems, and an adequate and effective basis for reuse of knowledge, experience, design and implementations. Software architecture is not confined to a phase in a software development process, but rather it is intertwined with other activities in software development. Software Architecture provides the team with a means for dealing with complexity, for managing and controlling risks, for effectively and efficiently reusing previous efforts, for making system-wide long-term decisions on how to address the stakeholders' requirements and expectations, and for the early assessment of quality. Thus, software architecture is crucial for appropriately balancing the cost-time-quality equation in a development project. Software architecture knowledge is large, heterogeneous and ever growing, encompassing architecture description, design and evaluation. By applying such knowledge, the architect reifies the potential of the software architecture discipline in the context of a software development project.

Modeling is an intrinsic and essential activity for people to conceptualize and reason about an existing or potential reality. It underlies our ability to think and imagine, to see patterns, to predict and manipulate processes and things, to express meaning and purpose, and to communicate. Modeling has been incorporated to the Software Engineering practice to deal with the complexity of both software systems, and of developing such systems. The prominence of modeling in software development has grown in the last decade. Developers rely on models to plan, analyze, design, validate, document and communicate the multiple aspects of software systems. Moreover, the emergence of model-driven techniques, based on domain-specific languages and model transformation and generation engines, has positioned models as the primary work products in software development. The researcher and practitioner community has begun to use these techniques to capture and encapsulate knowledge for particular domains: languages allow the abstract conceptualization of the domain, and transformations allow the abstract manipulation and synthesis of such models. Model-Driven Architecture (MDA) is the most notorious initiative in this context. Despite its name, it does not refer to the application of model-driven techniques to architecture, but rather it defines the architecture or structure of model-driven methodologies. It proposes the successive refinement of platform-independent models, to platform-specific models, to the system implementation, automating that refinement by means of model transformations. Modeling has also pervaded

the Software Architecture discipline, mainly in the context of the architecture description practice. Architecture design is still underusing the potential of model-driven techniques, which are being used mainly by generative approaches to derive partial detailed design and implementation skeletons from architecture descriptions. The MDA approach is positioned in this context.

In our work, we aim for a thorough application of model-driven techniques to provide conceptualization, formal meaning, and tool support for the Software Architecture practice. We define a model-based interpretation of software architecture knowledge concepts, that conforms a homogeneous means for expressing such knowledge, making it shareable, reusable, and directly applicable during architecture design. Thus, our goal is to apply model-driven techniques to capitalize architecture knowledge.

1.1 Context

The software architecture of a system constitutes the conceptual essence of the system, the principal decisions regarding its design, and the key abstractions that characterize it [TMD09]. It is devised and designed by the software architect in order to address the critical concerns of the significant stakeholders. The software architecture is an intangible concept or idea that must be captured and described in order to be preserved and communicated, analyzed and evaluated. The architecture description is the actual tangible work product that reifies or represents the architecture of a system [ISO11]. The architect follows an architecture design process to devise and decide the most suitable software architecture for a system, and to capture it by means of an architecture description [BCK03]. Architecture design is iterative and incremental [FCK07], spans system conception, construction and evolution, and is intertwined with the most relevant activities of software development processes [TMD09]. It is a broad, creative and dynamic activity that is much more about discovering stakeholders' concerns, devising and evaluating alternative solutions, and making tradeoffs, than simply capturing information [RW05]. However, capturing and describing the architecture is not a task separate from design. It is an essential part of it as the architecture description serves as a ready vessel for holding the impact of architecture decisions as soon as those decisions are made [CBB+10].

The iterative and incremental nature of architecture design allows the architect to cope with size and complexity, favoring the resolution of the most critical concerns first, possibly at the expense of others. The architect delves into alternative solutions by trial-and-error, going forward and backward in the paths of the tradeoffs and decisions made, and deciding the solution that provides the best system-wide long-term benefits while actually meeting the stakeholders' expectations. The software architecture community has detected that the relevance of architecture decisions goes beyond the moment they were made [Bos04, TABGH05]. There was a paradigm shift from capturing the consequences of the decisions made in terms of architectural elements, to capturing the decisions themselves, the evaluated alternatives and their rationale [Kru04]. Architecture decisions provide stakeholders with the fundamental understanding on why the architecture is shaped in a specific way instead of any other, and they allow architects to informedly reconsider the alternatives when changes occur to the

decision factors, facilitating system maintenance and evolution. As a consequence, current practice for architecture descriptions emphasizes the importance of capturing and documenting such decisions and the supporting rationale, in addition to the elements characterizing the structure and behavior of the system [CBB+10, ISO11, TMD09].

Capturing and maintaining architecture decisions in architecture descriptions require a substantial effort, and even more if the architect intends to be thorough in accurately capturing traceability information from requirements to decisions, and to the impacted architecture elements. For practitioners, this effort seems greater than the perceived benefit [HAZ07]. Although they recognize the importance of documenting decisions and rationale, most of them face barriers like the lack of standards and tool support, and the restrictions in time and budget [TABGH05]. A standard representation mechanism and the appropriate tool support would clearly ease and improve the adoption of capturing decisions along with the architecture description. We claim, however, that while this enhanced support is necessary, it is still not enough. A significant improvement in productivity can be achieved by using architecture decisions not only to facilitate understanding, but also to encapsulate the architect's expertise making it shareable and most importantly, reusable when exploring alternative architecture solutions or when developing similar products. This goal cannot be achieved when architecture solutions are captured in terms of textual descriptions, or when their effect is tangled and scattered in the architecture description. Architecture decisions and solutions need to be captured in a way that they can be used as tools by themselves, applied in new scenarios beyond the ability to manually reproduce their effect in a new architecture description.

As a discipline, Software Architecture is the study of how software systems are designed and built [TMD09]. Its study began in the 1990s, and since D. Perry and E. Wolf's paper on software architecture [PW92], the community has actively researched the theoretical and practical aspects of the field. After a decade, the discipline was understood as a still-maturing discipline [Lar02], its adoption in industry grew [Bos04], and its community started to face the popularization of the discipline [SC06]. Medium and large companies had their chief architects, used precooked architectures in the form of patterns and platforms, and counted on architecture knowledge for building software [KOS06]. Also, the relevance of architecture decisions was detected [Bos04, Kru04, TA05, TABGH05], and the application of model-driven techniques and aspect-orientation on software architecture, as well as the improvement of tool support, was foreseen [Fra02, KWB03]. Since then until today, the practice of software architecture has become indispensable technically, and a critical enabler of competitive advantage organizationally [KBC12]. On the one hand, the discipline keeps changing due to the emergence of new development processes, the types of systems to be addressed, the available technological platforms, and mainly due to the ubiquitous nature of software. On the other hand, the fundamentals of the discipline, captured by design mechanisms, techniques and methods, still conform the basis for the new application scenarios.

The software architecture body of knowledge is still ever-growing, as it influences and is influenced by processes, methods, technologies, and the industrial experience in the broader field of Software Engineering. It is large, as practically two decades of research effort and industrial application have produced cumulative techniques and methods that compliment each other. It is heterogeneous, as it deals with different aspects of architecture, from the identification of business goals and the engagement of stakeholders, to the elicitation and

renegotiation of requirements, to decision making, to analysis and evaluation. It is diverse, as a plethora of methods, techniques and mechanisms are available to choose from, even for the same aspect of the discipline. It is disperse, as it is covered in books, research and industrial articles, educational courses, platforms, reference architectures and tools. Also, it has different scopes. While there is a large body of knowledge publicly available to the community, other resides in the context of development companies, and other resides in the actual experience and expertise of each software architect. While this nature of architecture knowledge has promoted the development and growth of the discipline, it has hindered its application to the particular context of each development project. Ideally, practitioners should select the subset of such knowledge that best fits their organizational context, the skills and experience of external stakeholders, the expectations on the system functionality and quality, and the skills and expertise of the development teams. However, in practice, practitioners are not only restricted by the subset that they actually know about and master, but also by the capability of adapting and adjusting such knowledge to make it applicable in their particular development scenario, usually with scarce tool support. As it is available today, architecture knowledge poses an obstacle to any productivity improvement aiming to make architecture decisions and solutions shareable, reusable and applicable in different scenarios. To the best of our knowledge, the community still lacks a homogeneous and unified means for capturing architecture knowledge that enables reuse and effective tool support.

The benefits of reuse in Software Engineering have been recognized for decades, being the improvement in quality, productivity and reliability, and the reduction of the development effort, cost and time to market, the most notorious ones [AMC+06]. Reuse is the utilization of previously developed artifacts across the development of multiple systems [FVBS09]. Reuse is not something that just happens [Tra88]. Unplanned ad-hoc reuse is only opportunistic and at most has a short-term effectiveness. For reuse to be successful, it must be planned from the onset of development projects, or even before at an organizational level. However, planned reuse requires an upfront investment to identify the technological opportunities and limitations, and to acquire or develop a core set of assets to be reused or adjusted in particular scenarios. Despite this entry barrier, planned reuse has positioned as the critical success factor in the continuous demand on software development for lower costs, faster deliveries, better quality, and the management and provisioning of the expected system variants. Particularly, the Software Product Line (SPL) approach [Nor99] has positioned as one of the most effective for planned reuse [Som06]. The SPL approach undertakes the development of a set of related products as a single, coherent development process, where the individual products are developed from a core base of reusable assets. These core assets capture the commonalities across the products in the line, and the variabilities between individual products [CHW98]. Reuse of core assets is key to productivity and quality gains [Jéz12].

In the context of SPLs, the role of software architecture is even more important than in single-product development, as it allows practitioners to capture the commonalities and variabilities supported by the different products [OM07]. It makes SPLs better by improving design consistency and traceability, cheaper by requiring the definition of interfaces that reduce the need for non-essential or redundant implementations, and faster by providing mechanisms to reuse or adapt core components in particular products [MSG96]. The product line architecture (PLA) is an early and prominent member in the set of core assets, that is expected to persist over the life of the SPL [NCB+14]. It addresses the critical functionality,

the expectations on quality attributes, the constraints and the potential risks of failure that are common to all the products in the SPL. Additionally, it identifies and captures those parts where the products differ, and explicitly provides the variation mechanisms to support the diversity among the products. The complexity of designing a PLA lies in the integration of the critical decisions to be made with the expected variability of the SPL [KM04]. This is a challenging task as variability can be expected on both functional and quality requirements [NI07]. Choosing the appropriate variable parts and variation mechanisms is among the most important tasks for the architect, and is decisive in the potential products that can participate of the SPL [NCB+14].

Architecture design of SPLs is a notable scenario for planned reuse of architecture decisions and architecture solutions. While the software architecture of a single system is the set of principal decisions made, the product line architecture is the set of principal decisions that are simultaneously applicable to multiple related systems, with explicitly defined points of variation [TMD09]. Hence, architecture decisions and solutions are naturally reused in multiple products of the product line. Moreover, we claim that in a PLA description, the variation mechanisms for a variation point are actually making different architecture decisions, and thus, selecting variants is actually including the corresponding architecture solutions in the product architecture. Then, counting with a homogeneous means to capture and use architecture knowledge that enables to explicitly capture decisions and solutions making them reusable and directly applicable, allows the architect to deal with the complexity of architecture design of SPLs, providing an automated mechanism to create particular product architecture descriptions from the PLA description. Such an application of architecture decisions is just recently being analyzed in the research community, as discussed in [GAWM11].

Model-Driven Engineering (MDE) [Ken02] is positioned as a paradigm-shift from code-centric software development to model-based development [Béz05b]. It promotes the systematization and automation of the construction of software artifacts, raising the level of abstraction in which developers reason about the problem to be addressed and the solution to be provided. MDE allows developers to express domain concepts effectively by means of domain-specific modeling languages, to capture different aspects of a system by means of models expressed in terms of those concepts, and to automate the manipulation of those models by means of model transformations [Sch06]. Practical applications of MDE techniques are usually intensive in modeling artifacts, involving a large, complex, heterogeneous, and interrelated set of them [Mod08b]. Modeling-in-the-large poses additional challenges beyond the particular technologies used for modeling-in-the-small, such as expressing modeling languages, models and model transformations [BJRV04], requiring techniques and tools for managing modeling artifacts and capturing their interrelations. Megamodeling is a model-based approach to modeling-in-the-large [BJV04], being Global Model Management its most notorious realization, providing both a conceptual framework for megamodeling as well as integrated tool support [Mod08b].

MDE techniques have slowly pervaded the practice of software architecture. As stated by the ISO/IEC/IEEE 42010:2011 standard [ISO11] on the architecture description practice, architecture descriptions are organized as an aggregation of architecture views aggregating architecture models that consist of the representation of particular aspects of the system of interest. Moreover, the standard introduces the notion of model kind as the reusable

system-independent asset that captures the principles governing the construction of architecture models, particularly using metamodels for the definition of the modeling languages. However, the standard is not fully exploiting the benefits of MDE as it is not committed to a model-based approach to capture architecture descriptions as a whole, allowing also the application of document-centric and repository-based approaches. In the context of architecture design, its practice is still underusing the potential of MDE. The most notorious application is in the case of generative technologies [TMD09], being the Model Driven Architecture (MDA) initiative [OMG03] the most outstanding approach. By their means, an architect can automatically generate design and implementation skeletons from architecture descriptions. Software architecture description and design can be regarded as a particular case of modeling-in-the-large, where the architecture description is a set of modeling artifacts captured by an integrating megamodel. However, the notion of megamodeling is practically alien to the software architecture practice. In the context of Software Product Lines, model-driven techniques are being exploited to a larger degree than in the Software Architecture discipline. There are techniques for variability modeling, and purpose-specific or extensions to modeling languages to cope with variable parts within the models in a development project. Model transformations are used to automatically derive product-specific models from product line models defined as core assets [Jéz12]. These model transformations encapsulate the developers' knowledge on how to produce such derivations. In the context of software architecture, product architecture descriptions are automatically derived from the product line architecture descriptions that embeds all possible variability. These approaches rely on the architect's ability to build a complete product line architecture description, capturing and documenting commonalities and variabilities in the architecture views and architecture models. Architecture decisions and solutions are then embedded in the architecture description without taking advantage of planned reuse techniques for them.

1.2 Problem Statement

Architecture design is the creative and dynamic endeavor performed by the architect to iteratively and incrementally design and build the software architecture of a system of interest. It is a challenging tasks that requires leadership and technical skills and experience, to guide stakeholders and development teams, and to devise and decide solutions that provide the most system-wide long-term benefits while actually satisfying functional, quality and variability expectations. The available architecture knowledge provides the architect with methods, techniques and tools that capture studied and proven practices, allowing to predict the potential benefits and liabilities of the systems built by their means. Applying such knowledge promotes quality and reduces risks, improving the chances of success of the development effort and the customer satisfaction on the resulting systems.

Architecture knowledge is ever-growing, large, heterogeneous and diverse, and it is dispersed in numerous publications, specifications, platforms, reference architectures, and tools. An architecture design effort requires knowledge from different sources, published or communicated at different levels of abstraction or rigor, in different languages, and possibly requiring varied tool-support that is rarely available in an integrated development environment. As a consequence, as it is available today, architecture knowledge is not directly applicable in prac-

tice. The software architect must devote a considerable effort in adjusting and adapting such knowledge to make it applicable in a particular development scenario. Thus, the problem is that in practice, architecture design hardly achieves the level of quality that is possible given the available architecture knowledge. Moreover, the design effort is not straightforwardly repeatable as the architect's experience and expertise are implicitly embedded in the produced architecture descriptions, hampering the possibility of reusing architecture decisions and solutions in new scenarios.

To the best of our knowledge, the community still lacks a homogeneous means to capture the diverse architecture knowledge that (i) covers both architecture description and architecture design, (ii) makes architecture knowledge directly applicable, shareable, reusable, and tool-friendly (iii) provides first-class representation for architecture solutions and architecture decisions, and (iv) automates traceability between requirements, architecture decisions and architecture elements conforming the architecture description. The goal of this work is to develop such a unified and homogeneous means to capture and disseminate architecture knowledge.

The problem stated in our work was originally postulated in late 2007, and since then it has suffered practically no changes. At that time, we realized that our work was related to hot research topics in the Software Architecture discipline, such as the conceptualization of architecture knowledge, first-class representation for architecture decisions, and the application of modeling techniques to the architecture practice. As a consequence, we needed to keep aware of advances and results in those areas during the timespan of our work, not only to integrate relevant results adjusting ours, but also to determine the validity of the stated problem and the developed solution.

In this time period, the community devoted a large research effort with respect to first class representation of architecture decisions, as envisioned by key players in the discipline some years before. Also, the standard on the architecture description practice evolved reaching a new version in 2011 [ISO11]. Moreover, model-driven techniques evolved, counting since 2009 with a better understanding of modeling-in-the-large, its conceptualization in terms of megamodeling, and its realization by the Global Model Management approach [Mod08b]. Those concepts and techniques that were directly related to our problem were incorporated to our developed solution, as reported in this thesis.

Most importantly, the context and the stated problem are still relevant to the research and practitioner community. In the joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA) held in 2012 [WIC12], six out of the eleven sessions are directly related to the research topics of our work: architecture methods and techniques, documenting software architectures, architecture knowledge and decision making, architecture decisions, product lines, and variability. During this conference, two panels were held to discuss the past, present and future of the discipline, and the impact of research in practice, where key players in academy and industry presented their perspectives[1]. Researchers envisioned focus on organizational and cultural aspects of the discipline, more prominence of models, and the need of large and coordinated research efforts.

[1]Presentations at `http://wicsa2012.soberit.hut.fi/technical-program/technical-programme/` (last accessed on August 2014).

Practitioners remarked the gap between research and practice priorities, and questioned the proliferation of techniques with little cooperation and reuse, and their scarce transference to industry.

Noticeably, the problem we stated in this work is still current. Practitioners emphasized that the large body of architecture knowledge available is not being appropriately transferred to industry and it is not being applied to its full extent in practice. Thus, there is a need for facilitating the communication of such knowledge and to make it directly applicable in practice. Also, and even more important, the goal we pursued in this work moves one step forward addressing current research and practical needs. A unified and homogeneous means to capture and disseminate architecture knowledge can facilitate the cooperation and reuse of existing approaches and techniques, and it can favor the development of large coordinated research efforts.

1.3 Research Hypothesis

Our work aims to prove the following:

Hypothesis: Current Model-Driven Engineering constructs, techniques and tools, can provide a homogeneous and unified means for capturing and communicating architecture knowledge on architecture description and design, making such knowledge directly applicable by architects, favoring reusability, automating traceability, and enabling tool-support.

Corollary: Planned reuse of model-based architecture decisions and solutions can automate product architecture design in architecture-centric Software Product Line development.

1.4 Research Goals

The general goal of this thesis is directly derived from the research hypothesis:

General goal: To define a model-based mechanism that allows the community to consistently and completely represent and capture architecture knowledge on the architecture description and design practice.

The specific goals derived from this general goal are:

Specific goal 1: To make the model-based mechanism to be aligned to current conceptualization and standardization of the architecture description and design practice, but that remains independent of any particular architecture description technique and any particular architecture design method.

Specific goal 2: To formally specify the model-based mechanism in order to provide the formal basis for domain-specific tool support.

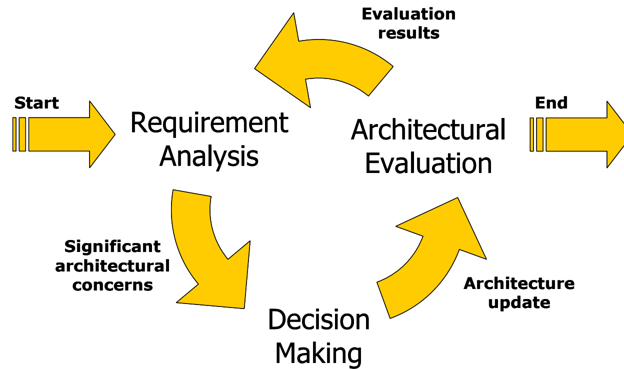


Figure 1.1: *Abstract model for a software architecture design process.*
The figure is an adaptation of the proposals in [FCK07, HKN⁺05]. The iteration starts by performing Requirement Analysis in which significant architectural concerns are identified. Then, in the Decision Making activity a solution for these concerns is decided and the architecture description is updated. Afterwards, the new architecture is evaluated. The process ends when all architectural significant concerns are satisfied by the architecture. When they are not, a new iteration is performed in order to cope with uncovered concerns.

1.5 Developed Solution

We address the problem of underusing existing architecture knowledge during architecture design by applying model-driven techniques to the definition of a unified and homogeneous means for capturing such knowledge, making it shareable, reusable, tool-friendly, and directly applicable by the architect. Thus, we use model-driven techniques to provide conceptualization, formal meaning and tool-support for the architecture design practice.

An architecture design process guides the architect through the design of the architecture of a system of interest, using an architecture description as the main artifact to capture and communicate the architecture decisions made and their effect in terms of interrelated architecture elements. Several architecture design processes are available [BCK03, Bos00, Kru03, RW05] as well as comparisons between them [DN02, FCK07, HKN⁺05]. Even though there is no consensus on a unified process that fits all architecture design scenarios, the community recognizes three main goals for these processes: to understand the problem, to solve it, and to evaluate the solution. Thus, in general, an architecture design process consists of three major activities that are repeated iteratively. We illustrate this general process in Figure 1.1. First, the architect identifies and captures the significant concerns by analyzing the requirements, constraints and risks that are yet to be addressed, in the light of the architecture built so far for the system of interest. Second, the architect devises one or more potential solutions to address the identified significant concerns, compares and weights these solutions, and decides the one that offers the best system-wide long-term benefits despite its drawbacks. Then, the architect updates the architecture description being built in order to make it reflect the decision made. Third, the architect and the appropriate stakeholders, evaluate the architecture to detect inconsistencies and flaws, and to determine additional or pending requirements to be addressed. The iteration ends when all significant concerns of all involved stakeholders are satisfied by the architecture that is captured and communicated by means of a complete- and detailed-enough architecture description.

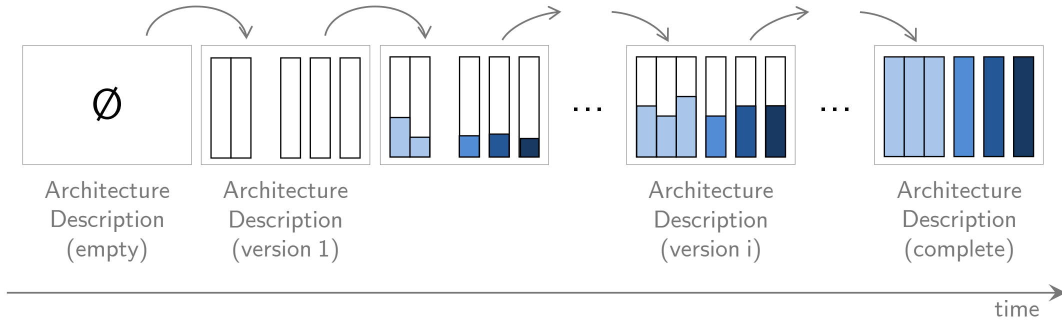


Figure 1.2: *Iterative & incremental nature of architecture design.*

The figure illustrates the incremental construction of the architecture description, iteration by iteration. The figure uses a large rectangle to represent an architecture description, a small vertical rectangles to represent architecture views and architecture models. The colored section represents the presence of architecture elements populating the architecture models. In the figure, the architect starts with an empty architecture description, decides the main structural organization of the architecture description, and iteratively and incrementally populates the architecture views and architecture models to capture and reflect the decisions made.

The architecture description is incrementally constructed, iteration by iteration. At each iteration, new architecture views or architecture models can be added to the architecture description. Also, the architecture elements populating the architecture models can be updated, refined or decomposed, in order to materialize the architecture solution devised by the architect. We illustrate the iterative and incremental nature of architecture design in Figure 1.2. However, architecture design does not progress linearly in practice. At each iteration, the architect explores different alternative solutions and develops them to the sufficient extent to convince significant stakeholders that the selected solution is the most suitable one. Besides, the architect might need to change direction, possibly even significantly, partway through architecture design as a result of new information uncovered or a redefinition of the requirements or their prioritization [RW05]. Figure 1.2 presents only the successful branch of the tree of alternative solutions that is actually developed by the architect in an architecture design effort.

In the current state of the art, while the architecture description of the system being developed is explicit, the design of this architecture is implicit. The architect focuses on creating and maintaining the architecture description, an explicit artifact that is communicated to significant stakeholders. However, the actual activities performed by the architect are not explicitly captured, but rather, they are implicitly reflected in the constructed architecture description. The architect follows the coarse-grained activities defined by an architecture design process, but enacts them by performing fine-grained steps regarding *how* to query and update the architecture description to capture relevant information and to reflect the impact of the decisions made. In terms of the illustration in Figure 1.2, during architecture design the architect constructs the architecture descriptions — the rectangles — by carrying out the corresponding fine-grained updates — the arrows —, either manually or assisted by tools. The arrows are not captured explicitly, they are implicit in the resulting architecture description built. As the relevance of decisions goes beyond the moment they were made [Bos04, TABGH05], an architecture description also captures the decisions made, describing the considered alternatives and the justifying rationale [Kru04]. By this means, the

architect facilitates stakeholders' understanding on *why* the architecture description is shaped in any particular way. However, the knowledge on *how* the architect proceeded to built such an architecture description is lost, and as a consequence, it cannot be communicated, shared, reused, or directly applied in a similar architecting scenario.

Our solution is based of shifting the architect's focus from capturing the architecture description itself, to capturing how the architecture description is created or updated. At each iteration, instead of updating the architecture description manually using of text and diagram editor tools, the architect captures the fine-grained steps that must be performed on the current version of the architecture description to produce its next version. In terms of Figure 1.2, the architect explicitly captures the arrows, not the rectangles. Informally capturing *how* the architect proceeds provides no benefits as a tangible architecture description is still needed in practice to be used or evaluated by significant stakeholders. However, formally capturing it, i.e. in a way that it can be machine-processed, allows for the automatic generation of the architecture description by processing the captured fine-grained steps. These fine-grained activities provide architects with an architecture design scripting language, that conforms the underlying support to the coarse-grained activities of architecture design processes. Thus, in our solution, architecture description is implicit while architecture design is explicit. For us, architecture decisions and the involved architecture solutions are not only descriptive, but also they are prescriptive as they capture the intention of the architect, and they are applicable as they can be applied on one version of an architecture description to produce the next one.

We apply model-driven techniques to provide conceptualization, semantical foundation and tool-support for capturing how the architect proceeds to develop architecture descriptions, making such knowledge explicit, shareable, reusable, and directly applicable in architecting scenarios. We understand the practice of software architecture description and design as a particular case of modeling-in-the-large [BJRV04], and as such, we apply the megamodeling approach [BJV04], realized by the Global Model Management conceptual framework and integrated toolset [Mod08b].

We encode an architecture description as a set of modeling artifacts preserved in a model repository. These modeling artifacts are of different nature, such as reference models, terminal models, and transformation models, and they are interrelated to each other. Thus, the set of modeling artifacts reifying an architecture description also contains megamodels that preserve the characterization, classification and interrelation of all the constituent modeling artifacts stored in the model repository. We also use modeling artifacts to encode system-independent reusable architecture knowledge on architecture description, such as architecture frameworks and architecture description languages. Hence, the same conceptual framework provides support for system-dependent and system-independent architecture knowledge, facilitating shareability and reuse. Modeling artifacts are also used to encode fine-grained architecture design steps. We define an architecture design scripting language as a modeling language, and provide its formal semantics and tool support by means of model transformations on the modeling artifacts conforming the architecture description. Then, model transformations encode all the modifications than are performed to architecture descriptions, from system-independent patterns and tactics, to system-specific updates.

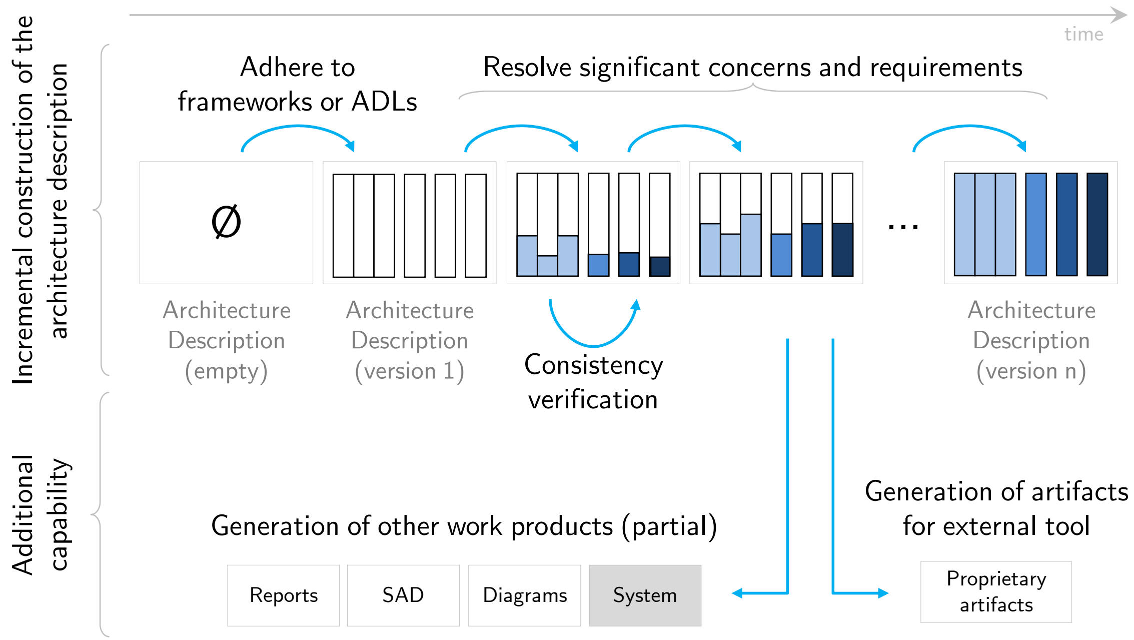


Figure 1.3: *Model-based architecture description and design approach.*
Our model-based approach to architecture description and design captures architecture knowledge using modeling artifacts such as reference models, terminal models, transformation models, among others. System-independent reusable knowledge, such as architecture frameworks, architecture description languages, architecture patterns and tactics, are captured by means of modeling artifacts that can be directly included and applied in various architecting scenarios. System-dependent architecture knowledge, such as architecture descriptions, architecture views and architecture models, are also captured by means of modeling artifacts stored in a model repository. Megamodels [BJV04] are used to characterize, classify and relate the participant modeling artifacts. Fine-grained architecture design steps are defined as a modeling language that allows the architect to express the set of updates that need to be performed to an architecture description to reflect architecture decisions and solutions. These steps are reified by means of model transformations on the modeling artifacts conforming the architecture description.
In the figure, large rectangles represent model repositories containing the modeling artifacts that reify the architecture description, illustrated as small rectangles. Arrows represent the application of architecture update scripts using system-independent architecture design knowledge and system-specific updates. They are explicitly encoded using modeling artifacts, as well as the reference models for architecture descriptions. However, the architecture description within large rectangles are automatically derived by machine-processing the arrows. Our model-based approach supports additional capabilities, by means of reusable transformations, to generate reports, diagrams, system skeletons, and proprietary artifacts of external tools.

Architecture knowledge on architecture description and design is then homogeneously captured by the different kinds of artifacts supported by model-driven techniques. We illustrate our approach in Figure 1.3. Particularly, the architect expertise on how to proceed to build an architecture description is reified by a set of architecture update scripts that, as they are modeling artifacts and mainly model transformations, can be reused and executed in new architecting scenarios.

However, the reuse of architecture decisions and architecture solutions, embodied as architecture design scripts, is mostly opportunistic. The architect must detect that a previously defined decision and its involved solutions might be helpful to deal with a new problem at

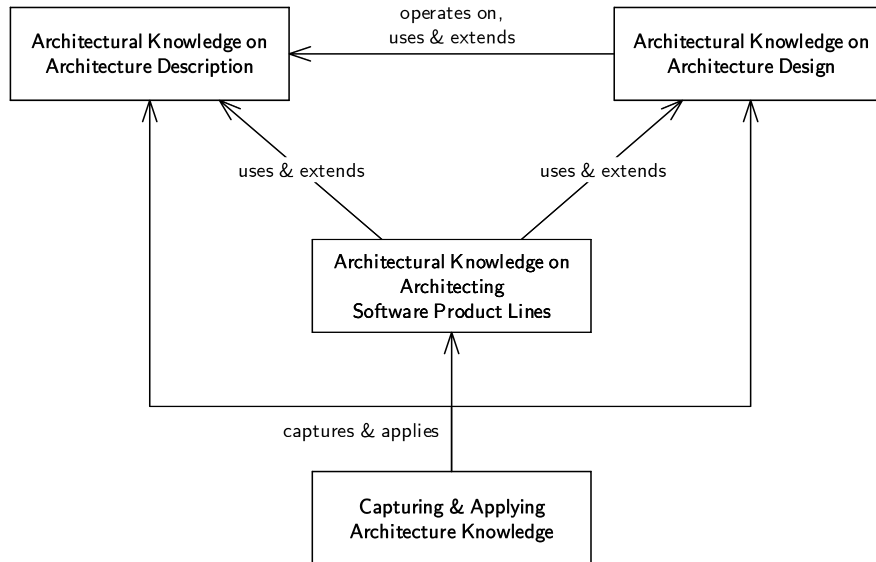


Figure 1.4: *Structural organization of the developed solution.*
The figure illustrates the structural organization of the solution we developed to address the problem stated in Section §1.2. We tackle architecture knowledge on architecture description, architecture design and on the architecture practice for Software Product Lines. We validate the solution by capturing SEI techniques to architecture description and design, and we apply the conceptualization and formalization, along with the captured knowledge, to the design of the architecture of a real-world software product line.

hand. Also, those solutions must be generic enough — or otherwise equal — to be directly reused in a new context. The major productivity boost by reusing architecture decisions and solutions is actually achieved when such reuse is planned in advance. Planning reuse of architecture decisions and solutions implies to develop a single core of several similar architectures, dealing with their commonalities and variabilities, allowing other architects to cost-effectively generate particular architectures by reusing the core one. The Software Product Line approach provides this exact architecting scenario where the architecture of each product is derived from the core architecture of the product line. Our approach to architecture design allows the product line architect to define architecture decisions and solutions in a way that they can be directly selected to automatically generate the product architecture corresponding to a given product configuration.

In order to reify our model-based approach to capture architecture knowledge making it explicit, shareable and reusable, we proceed incrementally. First, we work on architecture knowledge regarding the architecture description practice. We use the ISO/IEC/IEEE 42010:2011 standard [ISO11] on the practice as the base conceptualization of architecture description. Second, we work on architecture knowledge regarding the architecture design practice. As opposed to architecture description, there is no conceptualization for the practice. Then, we develop our own conceptualization, extending that for architecture descriptions and including the results of related research efforts and practice experiences in the area. Third, we extend the conceptualization of both architecture description and design to deal with architecting in the context of Software Product Lines. By this means, we incorporate and achieve planned reuse of architecture decisions and solutions, obtaining the expected

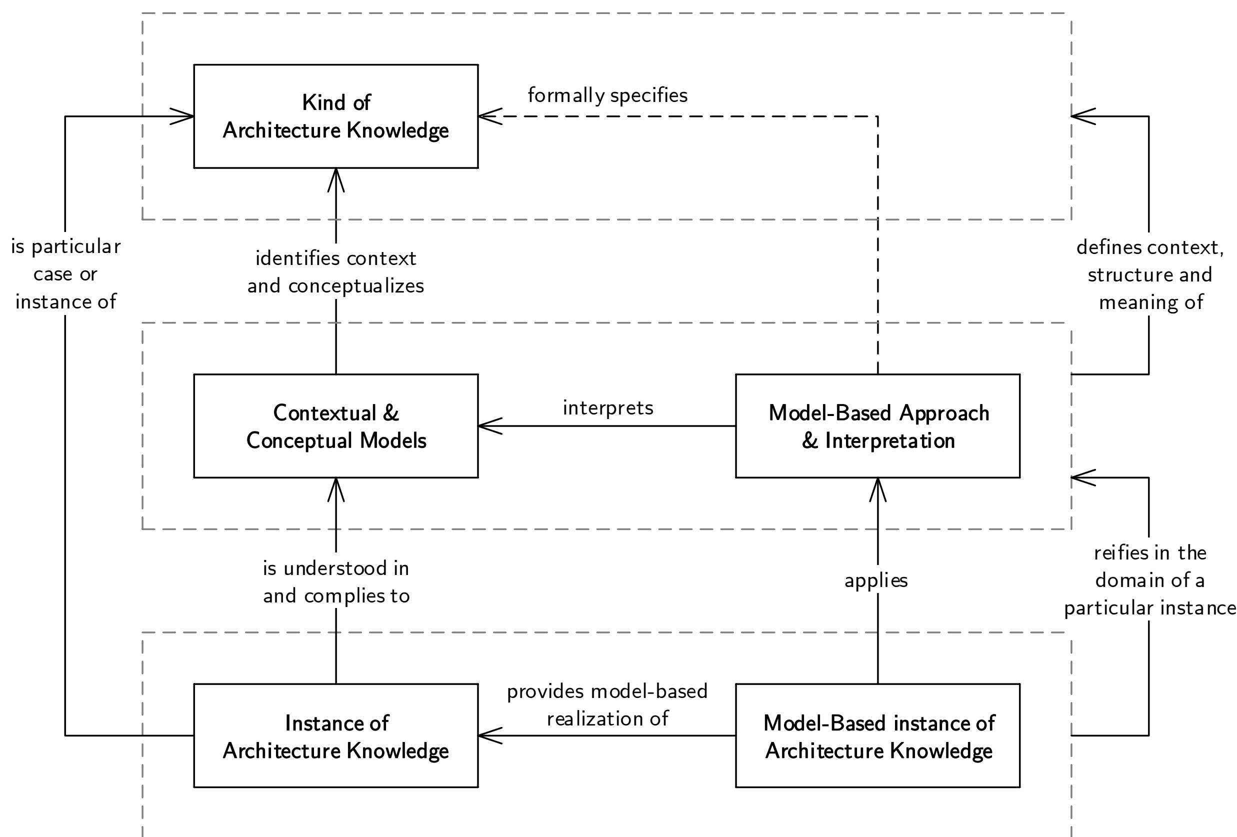


Figure 1.5: *Participants of the developed solution.*
The figure illustrates the main participants of the developed solution when addressing a particular kind of architecture knowledge. This structure is followed for each kind presented in Figure 1.4.

productivity improvement of our approach. We develop a model-based interpretation or mapping of our conceptualization of the software architecture practice using a denotational semantics approach [NN92, SK95] for its formal definition. The formal semantics rigorously establishes the modeling constructs, artifacts and techniques, required to correctly and completely capture the key concepts on the practice. We use the Global Model Management conceptual framework to modeling-in-the-large as the target of the formal semantics so as to characterize the properties and interrelationships of the produced modeling artifacts. Thus, modeling constructs and techniques conforms the unified and homogeneous means to architecture knowledge representation. Finally, we validate and exemplify our model-based conceptualization and formalization of the architecture description and design practice. To this end, we define systematic step-wise procedures that guide practitioners on how to capture system-independent architecture knowledge on the practice, and we use techniques from the Software Engineering Institute of the Carnegie Mellon University (SEI/CMU) as the running example. Then, we use the conceptualization and formalization, along with the captured architecture knowledge from SEI, to design the product line architecture of the Meshing Tool Software Product Line. Figure 1.4 outlines the structural organization of our developed solution.

The three major kinds of architecture knowledge that we tackle in our solution are architecture description, architecture design, and architecting in the context of Software Product

Lines. We follow the same strategy for building the solution for each of these kinds. Figure 1.5 illustrates this strategy. For a given kind of architecture knowledge, we first define a contextual model and a conceptual model of the practice. While the contextual model identifies the relation of the main concepts with respect to their real-world context, the conceptual model identifies the main concepts pertaining the practice and their interrelation. Hence, while the former provides an external perspective, the latter provides an internal one. Second, we define our model-based approach and interpretation to formally specify the kind of architecture knowledge. The model-based approach defines the governing principles for our solution, and the model-based interpretation defines the formal semantics of the conceptual model in terms of modeling constructs and techniques. Third, we validate our solution for each kind of architecture knowledge. In the context of architecture description and design, we define the procedures that guide the application of our model-based approach and formal interpretation to capture architecture knowledge, and we exercise these procedures by capturing some of the techniques developed by SEI/CMU which conforms a real-world instance of architecture knowledge on architecture description and design. In the context of architecting in software product lines, we apply our model-based approach and formal interpretation, as well as the captured knowledge, to the architecture design of a real-world case study, particularly the Meshing Tool Software Product Line.

1.6 Methodology

This work underwent a long way before taking this final form. While the initial problem statement and research goal have remained practically unchanged, the solution has suffered successive refinements since its original conception. In fact, our original approach to address the stated problem was to develop a model-driven specialization of the software architecture description and design techniques proposed by SEI/CMU. By this means, we intended to demonstrate the feasibility of a unified means to capture architecture knowledge by actually constructing a particular case. We soon found that our essential idea of shifting the focus towards explicitly capturing architecture design to automatically derive architecture description, can be used beyond a particular case. There was a more general solution underlying our envisioned approach, one that could provide a foundational means for software architecture techniques based on Model-Driven Engineering ones. As a consequence, instead of building "yet another software architecture description and design method," we developed and formalized a method-agnostic approach to capture software architecture knowledge on architecture description and design practice in terms of modeling constructs and techniques. Then, SEI's approaches played the role of validation of our general solution.

This work was performed in three consecutive stages. As we were working on strongly active research topics, we required a methodology that allowed us to incrementally develop and refine the produced results. These three stages were:

Fundamentals & Feasibility. The goal of the first stage of this work was to study the fundamentals of the Software Architecture and the Model-Driven Engineering disciplines, and to perform a feasibility study of the essential aspects of the envisioned solution. We surveyed the relevant results available for both disciplines, and we studied the ongoing

research on the application of model-driven techniques to software architecture. As the emerging techniques for capturing architecture or design decisions had a descriptive purpose, we explored and developed different alternatives for first-class representation of decisions with a prescriptive and applicability purpose. To this end, we used model-driven techniques to capture decisions on how to derive system or component designs and implementations from system or component specifications.

Development. The goal of the second stage was to develop the core solution of our work. To this end, we applied model-driven techniques to reify SEI's approaches to software architecture description and design, using small case studies. The ability to encapsulate, share and reuse architecture decisions and solutions led us to explore the application of our approach in the context of planned reuse, and to apply it to a larger case study. Thus, we studied Software Product Lines, we extended our solution to handle commonality and variability, and we applied it to the Meshing Tool Software Product Line. This case study is used in this dissertation.

Refinement & Formalization. The goal of the third stage was to refine the developed solution, adapting or adjusting it to cope and cover new relevant research results on the research topics of our work. While performing the previous stages, new research results were published, some of them directly impacting our work. On the one hand, not only a new version of SEI's approaches was published, but also the research and practitioner community came up with a new revised version of the standard on the software architecture description practice. On the other hand, the modeling community kept evolving, producing important results on megamodeling, and realization techniques such as Global Model Management emerged. As a consequence, we continued to develop our solution, constructing a unified conceptualization of the software architecture practice, and its formal specification in terms of model-driven concepts and techniques. These results are reported in this dissertation.

The practical applications of our solution were key to our work, not only for validation, but also for improving and enhancing our results. Along the three stages we tackle various systems of different complexity. In the first stage we worked on service-oriented systems, both in lab and in industry, to grasp the fundamentals of real-world software architecture and the actual opportunity for Model-Driven Engineering techniques. We also worked on a small case study, the Point-of-Sale System taken from [Lar04], to create the proof-of-concept of our solution. In the development stage, we adopted the Meshing Tool case study as it had the appropriate complexity for our purpose and it was sufficiently understood in the context of our research group to allow us to focus on the architecting issues only. In the third stage, along with the refinement of the Meshing Tool case study, we pursued the transference to industry of our solution. In this latter context, we worked as the leader architect in the development of a software product line for Enterprise Document Management. This endeavor takes place in parallel to our research, is not part of goals of our work, and it is still undergoing. As a result, we do not report the partial results here. However, we comment on this experience in the conclusions in Chapter §7.

1.7 Contributions

Our work makes the following original contributions:

- *A first-class representation of architecture decisions and solutions, making them prescriptive and applicable.* The software architecture community is undergoing a paradigm shift towards capturing architecture decisions within architecture descriptions, improving understandability and maintenance of the architecture and the system. We claim that explicitly capturing architecture decisions and solutions can be useful beyond description. For us, they can also serve a prescriptive purpose rendering them directly applicable during architecture design, and improving productivity by making them shareable and reusable. To this end, our original contribution is a shift in the current enactment on architecture design. Instead on focusing on capturing the effect of decisions in terms of a description, our approach encodes this effect in a way that it can be repeatedly applied with the corresponding tool-support. Hence, we focus on making design explicit, rendering description implicit and automatically generated.

- *A homogeneous means for capturing architecture knowledge making it shareable, reusable, tool-friendly and directly applicable during architecture design.* In the current state of the art, architecture knowledge is large and rich, but diverse, disperse and communicated in a variety of means that make it hard to be applicable by architects without a considerable effort in adjusting and adapting such knowledge. Moreover, those adaptations cannot be shared afterwards in a way that they can be directly applicable in other scenarios, limiting the opportunity of reuse. To this end, our original contribution is a homogeneous means for capturing architecture knowledge.

We materialize these two main contributions of our work applying model-driven techniques to the conceptualization and formalization of software architecture knowledge, in a way that architecture decisions and solutions play the prescriptive and applicable role we aim for. Thus, we identified and captured the main concepts in the software architecture practice, and we developed a mapping or interpretation of these concepts in terms of model-driven ones. The realization of these activities yielded the following specific results that also conform the contributions of this work:

- *A contextual and a conceptual model for fine-grained architecture design steps.* In the context of the architecture description, the ISO/IEC/IEEE 42010:2011 standard [ISO11] provides a contextual and conceptual model for the practice. However, to the best of our knowledge, there is no analogous and integrated contextual and conceptual model for architecture design. While coarse-grained activities of architecture design processes may vary, we claim that the fine-grained steps followed by the architect to produce the architecture description can be contextualized and conceptualized. Thus, we extended the standard's models for architecture description, in order to integrate our conceptualization for our architecture design scripting language consisting of fine-grained update statements on architecture descriptions. This language embodies the core of our prescriptive and applicable architecture decisions and solutions, making them shareable and repeatable.

- *A formal model-based interpretation of architecture description concepts.* We developed a formal mapping from architecture description concepts, as understood by the standard, in terms of megamodeling constructs. We applied a denotational semantics approach to formally specify this mapping. The purpose of this formalization is a thorough and complete definition on which modeling artifacts are required to capture each concept of the practice, which properties those artifacts have, and how they are interrelated. This mapping is essential when enacting architecture design using model-driven techniques.

- *A formal model-based interpretation of fine-grained architecture design steps.* We extended the formal mapping for architecture description concepts to also include the reification of the architecture design language in terms of model-driven techniques. The formal interpretation of architecture description and architecture design conforms the homogeneous and unified means for capturing architecture knowledge.

- *A normalized and tool-independent metamodel for megamodels.* We used the megamodeling approach as the target domain of our model-based interpretation. Architectural concepts are mapped to modeling constructs as defined by that approach, and particularly the Global Model Management (GMM) [Mod08b] realization. However, in the current state of the art, these approaches do not provide a clear separation between technology-independent and technology-dependent constructs. While GMM makes a great effort in this direction, using extensions to deal with specific technologies or tools, the metamodel for megamodels is not captured separately and self-contained. Then, we defined our own version of a metamodel for megamodels, strongly inspired in those defined by GMM, and preserving tool compatibility with GMM's tool support [Atl09]. Our normalized metamodel provides a single, abstract and complete metamodel that we required for our interpretation, explicitly excluding any tool-related concept. In addition, we defined a formal notation based on set theory for formally expressing assertions on modeling artifacts. This formal notation is used then for characterizing the results of mapping architectural concepts in terms of model-driven techniques.

- *A model-based approach to planned reuse of architecture decisions and solutions.* The purpose of our approach to explicitly capture architecture decisions and solutions is to facilitate sharing and reusing architecture knowledge. While capturing architecture knowledge by means of our model-based interpretation allows architects to reuse community or organization knowledge directly in their projects, capturing architecture decisions and solutions allows architects to reuse them in the development of similar systems. In order to improve productivity during architecture design, the ability of reusing architecture decisions and solutions must go beyond opportunistic reuse. To this end, we defined a model-based approach to architecture description and design in the context of Software Product Lines, enabling planned reuse of decisions and solutions, and providing automatic derivation of product architectures from a core product line architecture and a particular product configuration.

This dissertation integrates, extends and formalizes the results that we have presented in [PBR09, PRB09, RPB09]. We did our best effort to make this dissertation self-explanatory and self-contained, not requiring the reader to have in-depth background on the three key

topics of our work, namely Software Architecture, Model-Driven Engineering and Software Product Lines. These topics are introduced in Chapter §2 and are thoroughly studied and discussed in the core chapters of this work.

What this thesis is not about

Although this thesis is concerned with architecture knowledge on architecture description and design, it does not propose nor define a new set of techniques or methods for their practice. From the perspective of architecture description, we do not define any new architecture description language, architecture framework, architecture viewpoint or model kind. From the perspective of architecture design, we do not define any new architecture design process, architecture pattern or tactic. Rather, this thesis is concerned with architecture knowledge representation, i.e. with providing a solid basis for representing any existing architecture description method, technique or language, facilitating the applicability of the represented or captured knowledge during architecture design. Also, we define and promote a paradigm-shift in the enactment of architecture design making it explicit, applicable and reusable, rendering architecture description implicit and automatically derived. By this means, we also provide a solid basis to assist the representation of the architect's effort, independently of the particular architecture design process he follows. However, this thesis does not define an architecture expert technique or tool that automates the conception of architecture solutions. Our approach relies on the architect to devise such solutions, but provides him with a domain-specific language to explicitly capture his effort. Besides, this thesis is not concerned with architecture knowledge management directly. While we formally define a unified and homogeneous mechanism for architecture knowledge representation, we do not define any technique to categorize and classify the captured knowledge, and we do not define any strategy for knowledge look-up. Our approach only considers the characterization of the captured knowledge in terms of the architecture concerns that are affected or assisted by any architecture construct, as supported by the ISO 42010 standard that our conceptualization is based on.

Even though we rely on model-driven techniques for our formal interpretation of the architecture description and design practice, we built such an interpretation to be technology-agnostic. While we rely on specific tools available today to work on the case study, the interpretation uses only the conceptualization of model-driven constructs provided by the Global Model Management approach to megamodeling. By this means, our work is not tied up to any particular technology and keeps current while the technologies evolve. Besides, this thesis is not about developing a new integrated environment for architecture design. Building our own new purpose-specific development environment would have required subsequent research and engineering effort to maintain and evolve the tool beyond the life-span of this thesis, in order to allow practitioners to actually rely on the toolset. The development of such a toolset is desirable, focusing mainly on usability so as to favor the adoption of our approach and the actual achievement of knowledge shareability and reuse in practice. However, the development of this toolset is out of the scope of our work. Our goal is to demonstrate the feasibility of model-based architecture knowledge representation, relying on the current tool support for Model-Driven Engineering techniques that is developed and

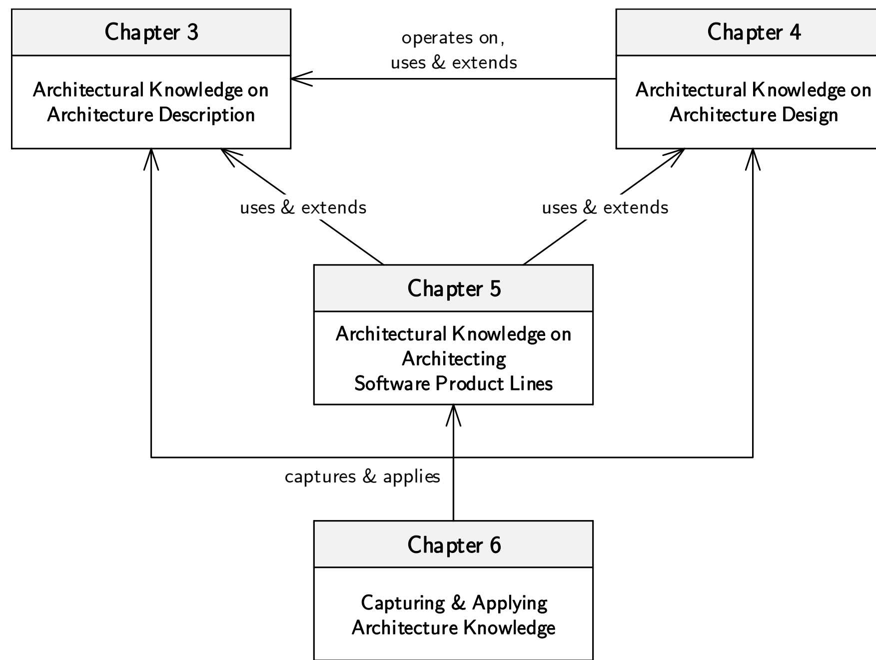


Figure 1.6: *Structure of the core of this dissertation.*
The figure presents the mapping between the structural organization of the developed solution that we illustrated in Figure 1.4, to the chapters of this dissertation.

supported by an active research and practitioner community. By this means, our conceptualization and formalization of the architecture practice set the foundation for the development of such purpose-specific toolset, which in turn can be developed on top of long-lived tool support of the continuously-evolving research results from the active and growing community of Model-Driven Engineering.

1.8 Structure of the Thesis

The remainder of this thesis is structured as follows.

Chapter §2 sets the background of this work. In the context of Software Engineering, this chapter reviews the Software Architecture discipline and provides a brief overview of software architecture description and design. It introduces the Model-Driven Engineering discipline and techniques, defining and discussing the different kinds of modeling constructs. It also introduces the Global Model Management conceptual framework, a particular realization of the megamodeling approach to modeling-in-the-large with available tool-support, and defines our refined metamodel for megamodels.

Chapters §3, §4, and §5, conform the core of this dissertation, dealing with the conceptualization and formalization of architecture knowledge on architecture description, architecture design, and architecting in the context of Software Product Lines, respectively. Figure 1.6 presents the mapping between the structural organization of the developed solution presented in Figure 1.4, to the chapters that address each kind of architecture knowledge. These three

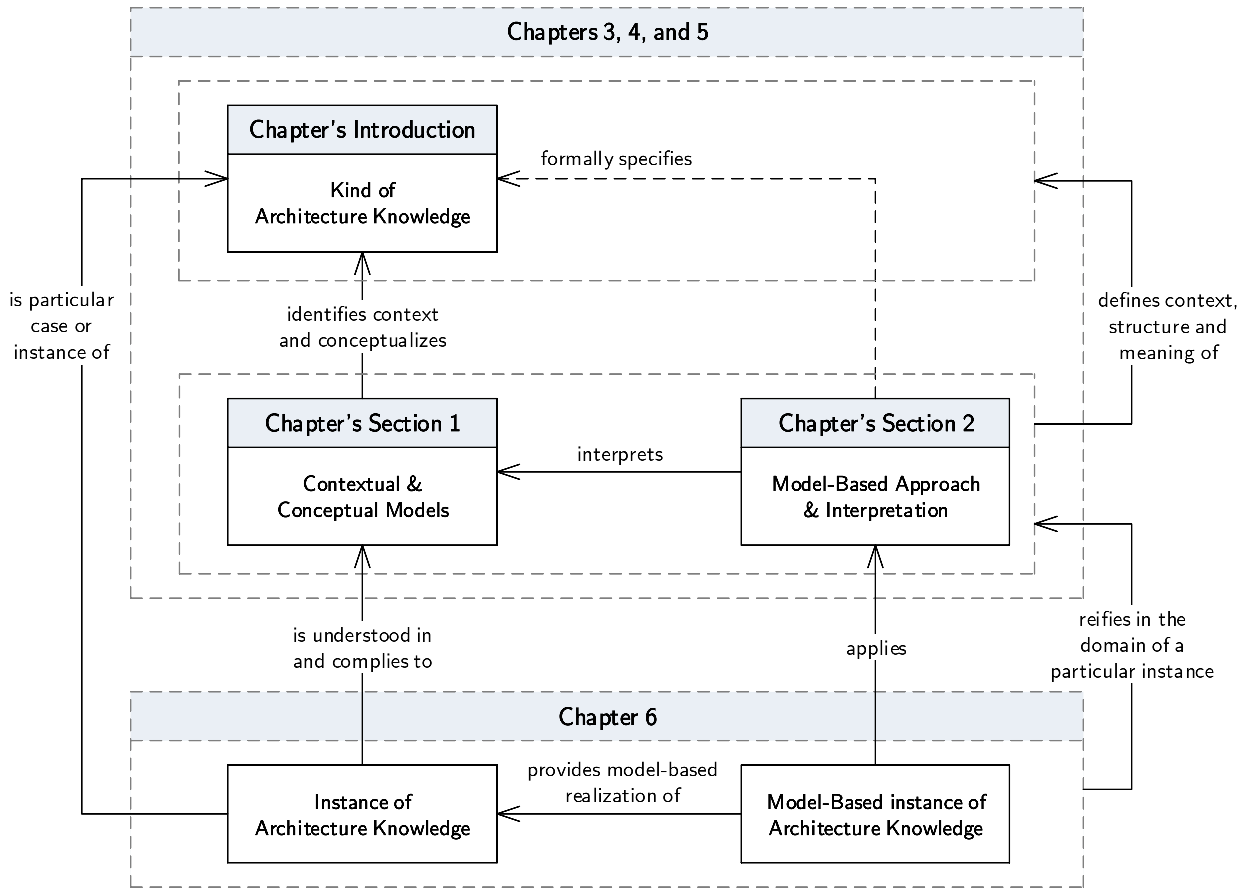


Figure 1.7: *Structure of the core chapters of this dissertation.*
The figure presents the mapping between the participants of the developed solution to the sections of the core Chapters §3, §4 and §5 of this dissertation. These participants were illustrated in Figure 1.5.

chapters are organized in the same way. The chapter's introduction briefly reviews the specific kind of architecture knowledge targeted by the chapter. Section 1 of each chapter defines the contextual model providing an external perspective for the main concepts of the kind. Also, it defines the conceptual model providing an internal perspective, i.e. the main concepts participating in the kind. Section 2 of each chapter defines our model-based approach and its formal interpretation in terms of modeling constructs. Finally, Section 3 ends with contributions and discussion. Figure 1.7 presents the mapping between the main participants of our developed solution to the sections that cover them.

Particularly, Chapter §3 reviews and thoroughly discusses the contextual and conceptual model for the architecture description practice defined by the ISO/IEC/IEEE 42010:2011 standard [ISO11]. It defines the theoretical foundation for our model-based approach, introducing the denotational semantics approach to formally define the interpretation of architectural concepts in terms of modeling constructs, particularly those conceived in the context of the Global Model Management conceptual framework.

Chapter §4 defines our contextualization of architecture design by extending the contextual model defined for architecture description. Also, it defines our conceptual model for the prac-

tice. It discusses patterns, tactics and styles, defines the scripting language for architecture design, and conceptualizes architecture solutions and decisions. It defines our model-based approach to explicitly capture architecture design rendering architecture description implicit, enabling reuse of architecture decisions and solutions. The rigorous interpretation of Chapter §3 is extended to cover the key concepts in architecture design, and to provide formal semantics to our architecture design scripting language.

Chapter §5 motivates the need for variability in software development and discusses planned reuse as the cost-effective approach to improve quality, productivity and reliability, presenting our conceptual model for variability. It reviews Software Product Lines (SPL) and defines our contextual model for architecting in this scenario. It extends the conceptual model and formalization defined for architecture description and design, in order to provide variability management and to enable automatic derivation of product architectures from a model-based product line architecture that explicitly captures decisions and solutions.

Chapter §6 validates and exemplifies the application of our model-based conceptualization and formalization of the architecture description and design practice. It defines systematic step-wise procedures that guide practitioners on how to capture system-independent architecture knowledge on the practice. We use SEI techniques as running examples, particularly the Views & Beyond approach to architecture documentation [CBB+10] and the Attribute-Driven Design method [SEI14a] as the coarse-grained method to be assisted by our model-based approach to architecture design. Also, this chapter illustrates a second level of validation of our work. We use the conceptualization and formalization that we defined in previous chapters, along with the particular instance of architecture knowledge captured in this chapter, to design the product line architecture of the Meshing Tool Software Product Line.

Chapter §7 concludes with a summary of our results, a discussion of the main contributions of our work, and an outline of possible directions for further work.

Chapter 2

Background

Software Engineering comprises all aspects of the production of software systems [Som06]. It is an engineering discipline as it encompasses the systematic, disciplined, and quantifiable application of scientific and technological knowledge, methods and experience, to the design, implementation, testing, and documentation of software systems [ISO93, ISO09]. The production and evolution of software systems are guided by software development processes. The adoption of a process is crucial as it comprehends repeatable practices and techniques that organize development and favor software quality. A software development process defines the workers' roles in a project, the work products they build and use, the activities they perform, and how these activities are structured along the project time-line. In other words, a software development process determines who should do what, how and when, so as to achieve a software product that meets the stakeholders' needs. All processes rely on techniques from the Software Engineering discipline, but they vary on the particular life cycle they are based on, the roles that are involved in the project, and the roles' responsibilities in terms on work products and activities. Additionally, processes differ in the level of rigor and formality that is expected during a project, and on the guidelines and tool support they rely on. Two inter-related and simultaneous sub-processes embody a software development process: management and construction [Kru03]. Management is responsible for coordinating project activities to ensure that the project fits its budget, the available resources, and the time restrictions. Construction is responsible for building the actual software system in the constrained context of the project. Hence, technical decisions during construction are focused not only on building a software system that meets its functional and quality requirements, but also on building it with the available budget, team and deadlines.

Software development processes are categorized and structured in major areas of concern called disciplines. Each discipline comprises a cohesive set of roles, work products and activities, independently of the actual life cycle of the software process. Disciplines in the management sub-process include Project, Configuration and Change Management. Disciplines in the construction sub-process follow the levels of abstraction at which the software system can be conceived. Requirements guide the recollection and elicitation of the system's expected functionality and quality attributes. Analysis refines the requirements building a developer-oriented vision of them. Design defines the detailed structure and behavior of the software system in order to provide the expected functionality while showing the desired

Chapter Contents

2.1 Software Architecture **27**

2.1.1 The Software Architecture Discipline 28

Definition of Software Architecture 29

Software Architecture in the context of Software Engineering . . 30

Software Architecture Knowledge 32

2.1.2 Software Architecture Design 32

The Decision Making activity 34

2.1.3 Software Architecture Description 34

2.2 Model-Driven Engineering **36**

2.2.1 The Modeling Discipline 37

Model-Driven Architecture 38

2.2.2 Modeling Constructs 39

Models 39

Definition 42

Metamodels 42

Model Weaving 45

Model Transformations 47

2.3 Global Model Management **51**

2.3.1 Conceptual Framework & Metamodel 53

Metamodel for megamodels 54

2.3.2 Realization & Tool Support 64

quality. Implementation codes the design targeting a particular technology and platform, obtaining an executable system. Deployment guides the installation and start-up of the system into a production environment. Validation & Verification activities are orthogonal to the previous disciplines. Verification is focused on the correctness of the artifacts and system being built, while Validation is concerned with how those artifacts and system fulfill the stakeholders' expectations.

Software Architecture is playing a critical role in current software development processes [TMD09]. Balancing the cost-time-quality equation is the main concern of the project manager who is in charge of the management sub-process. However, such equation is determined by the system-wide long-term decisions made on the underlying platform support, the integration and interoperation with external systems and devices, and the system's overall structure and behavior. Architecture-centric software development processes assign the responsibility of making these decisions to a software architect. A software architect leads the construction sub-process and works coordinately with the project manager to make the software development project succeed. The software architect is involved in the critical decisions

along all disciplines in order to build a software system that meets the stakeholders' expectations while reaching a compromise on quality attributes such as reliability, availability, robustness, security, performance and evolution, among others.

In this chapter we set the background of this work. In the context of Software Engineering and software development processes, we present the Software Architecture discipline as a means to deal with the complexity on developing software systems, to control risk and to effectively achieve quality attributes. Then, we introduce Model-Driven Engineering and explain its basic concepts. Although model-driven techniques aim for the development of entire software systems by using models and model transformations, we focus on using these techniques as the foundation for Software Architecture knowledge representation, particularly, knowledge on software architecture description and design. We describe the principles on Global Model Management (GMM) in Model-Driven Engineering, an emerging approach to deal with complex modeling processes. Designing the software architecture of a software system is a complex task and involves multiple architectural elements that, from several points of view, define the system's structure, its behavior, and the architecture decisions made to build it in a way that meets its requirements. GMM conforms the foundation of our work as we use it as the means for Software Architecture Knowledge representation.

This chapter is structured as follows. Section §2.1 presents the Software Architecture discipline, and reviews software architecture description and design. Section §2.2 describes Model-Driven Engineering and explains its basic constructs. It also discusses the Model-Driven Architecture framework. Finally, Section §2.3 introduces the purpose and principles of the megamodeling approach, and particularly of Global Model Management, and presents its current tool support.

2.1 Software Architecture

Building medium- and large-scale software systems is hard. Quoting Fowler's introduction to his book [Fow02]: "as the complexity of the system gets greater, the task of building the software gets exponentially harder." This kind of systems usually requires millions of lines of code, thousands of database tables, and hundreds of components, all running on dozens of computers [RW05]. Altogether, their development involves many stakeholders, spans for several months and requires numerous teams. This scenario presents formidable challenges to software development. Failing to address these challenges yields to over-budget projects, late deliveries, and unacceptable level of quality. Software architecture plays a pivotal role in overcoming these challenges. It is the proper primary focus of Software Engineering for the production of high-quality successful products [TMD09].

From the business perspective, software architecture allows organizations to meet their business goals. The additional effort and cost pay for themselves by enabling an organization to successfully achieve systems fulfilling the expected functionality with acceptable levels of quality, and also by expanding their software capabilities [BCK03]. Software architecture is an asset that holds tangible value to the developing organization beyond the project for which it was created. Giving preeminence to architecture offers the potential for realizing

intellectual control, conceptual integrity, effective project communication, management of a set of related variant systems, and an adequate and effective basis for reuse of knowledge, experience, design and code [TMD09].

From the development perspective, software architecture is the centerpiece of modern software system development processes. It plays an essential role for achieving intellectual control over a sophisticated system's enormous complexity [KOS06]. A software system's architecture embodies the critical decisions made during its development and any subsequent evolution [TMD09], and the system success depends upon those principal decisions made. Most software development processes have turned into architecture-centric processes for the effective, efficient and competitive development of medium- and large-scale software products. Being architecture-centric provides a means for dealing with system complexity, managing and controlling development risks, and for early assessment of quality. Building a software architecture is not a phase of development. It follows its own sub-process all along the system construction and evolution timespan. Such a sub-process is called a Software Architecture Design Method. Architecting is intertwined to other activities in software development as it looks for achieving the overall potential and capabilities of the software system.

In this section, we introduce the Software Architecture discipline. We review the definition of software architecture, its impact in the Software Engineering discipline, and the notion of architecture knowledge. Also, we overview software architecture design, mainly the decision making activity, and we briefly review software architecture description. We study software architecture descriptions thoroughly in Chapter §3 and architecture design in Chapter §4.

2.1.1 The Software Architecture Discipline

Software Architecture is positioned as a discipline by itself. Although the term was coined before, since Perry and Wolf's paper [PW92] on Software Architecture, an evolving community has actively studied its theoretical and practical aspects. In the years that followed, the adoption in industry has been broad and the research community has grown [Bos04].

However, Software Architecture is a still-maturing discipline. The number of conferences and journals in this area is growing, as well as the research effort devoted to it [Lar02]. This fact is reinforced by considering the key players' perspectives on the discipline. J. Bosch in [Bos04] identifies current software architecture problems and envisions future research towards first-class representation for design decisions. M. Shaw and P. Clements in [SC06] analyze the maturity of the field and suggest that the community is currently facing the popularization of the discipline. Also, they envision the improvement of Architecture Description Languages (ADLs), mainly based on the Unified Modeling Language (UML), and agree on the importance of counting on a representation for design decisions. P. Kruchten et al. in [KOS06] agree on the popularization of the discipline. They claim that medium and large software companies have their chief architects, use precooked architectures in the form of patterns and platforms, and count on the available architecture knowledge for building software. The authors foresee future research on design decisions and also on the application of Model-Driven Architecture [Fra02, KWB03] and aspect-orientation to software architecture, as well as the improvement of tools for assisting the architects' task.

Definition of Software Architecture

R. Taylor et al. in [TMD09] state the three fundamental understandings of software architecture that all authors in the field usually agree on. These assertions helps situate software architecture in the context of the Software Engineering discipline:

- every software system has a software architecture;
- every software development project has at least one software architect; and
- software architecture design is not a phase in software development.

However, what the community understands for software architecture is not agreed upon. Different authors have proposed their definitions over the years and there is no actual consensus on a satisfying, short, crisp definition of the term [KOS06]. Most of these definitions are listed at the software architecture practice web-site [SEI14b] maintained by the Software Engineering Institute of the Carnegie Mellon University (SEI/CMU). The joint effort on developing the IEEE 1471 Standard on recommended practices for software architecture descriptions [IEE00] defines Software Architecture as:

> *The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*

The ISO/IEC/IEEE 42010 Standard on architecture description [ISO11], the successor of [IEE00], updates the definition of the architecture of a system. The agreed definition was chosen to fit the broad meaning of the term *system* and by considering that all systems have architecture and it embodies what is fundamental to them, even if the architectures are not materialized as a tangible work product. For the standard, Software Architecture is:

> *The fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.*

Although architecture decisions lack first-class representation in the IEEE Standard 1471, in the ISO/IEC/IEEE 42010 Standard they are actually covered. However, architecture decisions are missing from the definition of software architecture. Several authors like J. Bosch in [Bos04] remarked the importance of architecture decisions, both during design and in architecture descriptions, and claimed that the focus of architecting is about making such decisions. This is reified in the definition provided by R. Taylor et al. in [TMD09], for whom Software Architecture is:

> *The set of principal design decisions made about the system.*

For the authors, the architecture of a system refers to the conceptual essence of the system, the principal decisions regarding its design, and the key abstractions that characterize it. There are good architectures, bad ones, elegant ones, and curious ones. An application's structure may be elegant and effective, or clumsy and dysfunctional. A precise and systematic definition of the design and representation facets of Software Architecture boosts understanding and skills of stakeholders, architects and developers, improving the chances for successful software systems. However, ultimately it comes to the actual decisions the architect makes to determine the success of the architecture and the usefulness of the systems.

Noticeably, in the software architecture community the focus is shifting from considering the software architecture as the architectural elements that characterize a system to considering it as the set of principal decisions that lead to these architectural elements.

Software Architecture in the context of Software Engineering

To understand software architecture as a specific phase in software development is simplistic and inaccurate [TMD09]. Confining software architecture to a phase, either early or late in a project life cycle, is to restrict the architecture to a subset of the actual critical decisions that must be made to fully characterize the system. If done early, the decisions might be contravened by subsequent decisions made outside the reach of the software architect. If done late, already made decisions, generally with a narrow vision of the whole system, might constrain the architect's task. While the devoted effort to create and maintain the architecture is usually more prominent in a particular phase, architecting activities take place during the whole project life cycle, and even during the evolution of the system. The software architect is in charge of the construction sub-process of a software development process, and as such, the architect's responsibilities span across all disciplines, from requirements to validation. In other words, the activities of a software development process are anchored in the architecture.

Envisioning the software architecture of a software system begins at the outset of any development activity. While evaluating the feasibility of a software system given the stakeholders' vision of the software, the role played by the software architect is crucial as he or she must envision a potential and achievable architecture of the system. In industry projects, the software architect is involved in the definition of project proposals, even before the external stakeholders approve the project and the actual development begins. Architecture decisions have an impact on team effort and cost, and hence, they impact the final price if the project aims to build a commercial software product. During a software development project, the software architect is responsible for all critical decisions concerning all aspects of the software being built, from achievable functional requirements and quality properties, to the system structure, behavior, supporting platform and inter-related systems. During software maintenance and evolution, it is the software architecture of the system that establishes the foundational decisions and that constrains and guides the potential paths for evolution. The software architecture seeks the balance between what is needed and what is possible in a particular real world scenario, favoring or constraining either the requirements or the architected solution.

In the traditional view on software requirements, a full understanding on the expected functionality and quality properties should precede any work towards the solution. In practice, articulating the requirements independently on any concern on how those requirements might be met is not only a complex task in real world scenarios, but also might be detrimental [Nus01]. Generally, stakeholders already have one or more organizational software platforms they rely on, existing software systems or tools that need to be replaced or integrated to, and they possibly envision functionality and quality properties that might not be achievable with the budget and time they count on. Then, early envisioning the solution to be built helps adjusting the requirements to a feasible level, clarifying and setting the

expectations of the stakeholders, and agreeing on actual characteristics of the outcome of the development project. Quoting R. Taylor et al. in [TMD09], “the starting point for a new development activity includes knowledge of what exists now, how the existing systems fail or lack to provide, and what on those systems can or cannot be changed or improved.” In addition, P. Kruchten states in [Kru03] that previously developed systems, reference architectures, technologies, and the actual architecture being built, not only drive requirements gathering and elicitation, but also provide a vocabulary for articulating them and serve as an inspiration of new opportunities.

In an architecture-centric development process, the activities of analysis, design and implementation proceed in an enriched and integrated fashion comparing to traditional software development. The boundary between requirement analysis and design is diminished as the expected functionality and quality properties are envisioned altogether with the design that might actually achieve them. Also, the boundaries between design and implementation are also blurred. Not only the software design guides the system implementation, but also the potential of the supporting platform, the existing libraries and the available components drive the actual design to build and might constrain or enrich the functionality to consider in the requirements.

Design quality correlates well with software quality as it would be extremely unusual to find a high-quality software system with a poor design [TMD09]. The distinction and relation between Architecture, Design and Implementation can be analyzed using the intention/locality criteria, as proposed in [EK03]. Architecture involves strategic design as it is concerned with global (non-local) decisions impacting the whole system and its development. For instance, such decisions include the guiding design principles, the architecture styles to apply, and the selection of standards, platforms and technology. Design involves tactical (local) design as it is concerned with local decisions impacting a particular part of the system and that are not affected by the expansion of the system. Examples of such decisions are the detailed internal structure of a part of the system, applied design patterns and programming idioms, and fine-grained refactoring. Although this notion is formalized by [Ede05], in practice, the actual distinction between whether a decision is strategic or tactical relies on the architect expertise and the foreseen impact of the decision on the system construction and evolution. Intuitively, architecture (strategic) decisions are the significant decisions that shape the system, where significance is measured by cost of change [Boo14]. Both architecture and design are intentional, as they refer to an abstraction of the system that specifies the intended structure and behavior of the system.

In contrast to Design, Implementation is extensional as it involves the actual code-level reification of the system. In practice, there is substantial interaction between architecture, design, and implementation, as they generally occur in parallel. A variety of architecture decisions can be made to reduce the implementation effort, such as generation technologies, frameworks, middleware and reuse [TMD09]. The implementation has an impact onto the architecture when key design decisions are made while working on source code, and usually take place in the context of agile methods to software development.

With respect to Validation and Verification, technically rich architecture descriptions are present before source code, and hence serve as the basis for and subject of early analysis.

Additionally, verification can be guided by the architecture, offering the prospect of early detection of errors and efficient and effective examination of the product.

Software Architecture Knowledge

Current software architecture knowledge includes techniques for describing, documenting, designing and evaluating the architecture of a system. Also, it offers a variety of patterns, guidances and best practices for defining the structure and behavior of a system in a way that it covers the desired functionality while its quality attributes can be predicted. However, there is no consensus on the best techniques for these activities. Practitioners are faced to a plethora of description languages and design methods, most of which are usually partially documented as they lack specificity and preciseness.

As stated by the Rational Unified Process [Kru03], a crucial factor of successful software architecture construction is based on the architect's experience. However, architecture knowledge is available today in books and articles, both printed and published online, and educational courses are included in software engineers curricula, facilitating the learning curve of newcomers. While architecture design methods encode the architecture knowledge on how to proceed to build an architecture, patterns and tactics encode the architecture knowledge of well-known solutions to common problems or requirements. Quoting M. Shaw et al. in [SC06], several architecture patterns, like N-tier and Client/Server among others, are examples of enormously successful practice recommendations. Tactics have a lower impact than patterns in the academic community, but they are beginning to be used in industry [BBK03]. While patterns solve general quality attributes mainly from a logical perspective, tactics have a broader architectural impact. However, tool support is essential so as to ease their application on architecture descriptions and to try out and evaluate different alternatives.

The large set of alternatives for software architecture design and description promotes the development of the discipline and allows practitioners to select the subset that best fits the organizational context, the stakeholders, the expectations on the system functionality and quality, and the skills of the development team. However, the lack of a unified means for documenting such architecture knowledge makes it difficult to share, reuse and provide tool support.

2.1.2 Software Architecture Design

A Software Architecture Design Method (SADM), a.k.a. Architecture Definition Process, is a method by which the critical concerns and needs of the stakeholders of a system are captured, elicited and prioritized, a software architecture to meet these needs is designed, the architecture is clearly and unambiguously described via an architecture description, and the architecture is evaluated so as to ensure that the expected requirements are actually addressed [RW05]. In other words, a SADM is a method for deriving a software architecture from the system requirements and for refining these requirements accordingly to what can be actually built in the context of the system's development project.

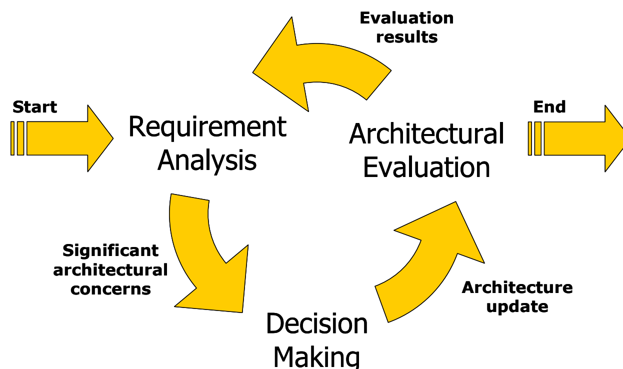


Figure 2.1: *Abstract model for a Software Architecture Design Method.*

The figure is an adaptation of the proposals in [FCK07, HKN+05]. The iterative method starts by performing Requirement Analysis in which significant architectural concerns are identified. Then, in the Decision Making activity a solution for these concerns is found and the architecture description is updated. Afterwards, the new architecture is evaluated. The process ends when all architecturally significant concerns are satisfied by the architecture. When they are not, a new iteration is performed in order to cope with uncovered concerns.

In the current state of the art, several SADMs are available for architects to chose from, such as those described in [BCK03, Bos00, Kru03, RW05]. There is no consensus on a unified design method that fits all development scenarios, and as studied in [FCK07], no method actually satisfies all architect's needs. The incursion of related technologies like Model-Driven Engineering [Sch06] and early aspects is motivating the emergency of new approaches. Several comparisons of SADMs have been published [DN02, FCK07, HKN+05]. Some of these authors also identify the three phases which are common to most of these methods: to understand the problem, to solve it, and to evaluate the solution. Thus, a SADM consists of three major activities: Requirement Analysis, Decision Making and Architectural Evaluation. We depict the process in Figure 2.1. The system's architecture is actually built during the Decision Making activity, which is intrinsically the core of a SADM. This activity consists of deciding the best alternative for addressing the most critical requirements and reflecting the decisions made in the system's architecture description. However, in order to be enacted successfully, the other two activities should be carefully integrated [HKN+05, RSM+04]. Eliciting and prioritizing the requirements correctly, guarantee that we are making system-wide long-term decisions that actually address the critical aspects of the system. A fine-grained method for this activity is SEI's Quality Attribute Workshop (QAW) [BEL+03]. To assess the resulting architecture guarantees that the architecture decisions made are the best compromise given the potential alternative solutions. Fine-grained methods for this activity are SEI's Software Architecture Analysis Method (SAAM) [KBWA94] and Architecture Tradeoff Analysis Method (ATAM) [KKC00].

Not all the SADMs that are available provide in-depth guidelines and techniques [FCK07]. Some of them consist of general guidelines and checklists while others, like Attribute-Driven Design [BCK03], provide precise techniques for resolving quality attributes at the architectural level. Despite this, no SADM is precise enough to encode all details on how a software architecture must be manipulated when performing an activity of the design method. In some cases, like in the Rational Unified Process (RUP) [Kru03], these details are somehow delegated to a companion toolset.

The Decision Making activity

Designing an architecture so that it achieves its significant requirements, mainly the expected quality attributes, is one of the most demanding tasks an architect faces. It is demanding as requirements usually lack specificity, knowledge on quality attribute resolution is not precisely documented or lacks tool support, and trade-offs are involved in achieving quality attributes. As studied in [BBKS05], there are basically three approaches to software architecture design with respect to quality attributes.

The Non-Functional Requirement approach [CNYM99] is based on decision trees of architecture decisions which state how each decision helps or hurts various quality attributes. The main problem with this approach is that whether a decision helps or hurts is a matter of context, i.e. there are a number of implicit assumptions in each branch of the decision tree. The second approach is the quality attribute model approach. A large community studies such models and has developed methods for measuring these attributes; ISO/IEC 9126-2 and 3 [ISO03a, ISO03b] list several metrics. The most notable success can be found for performance, while models for other attributes are less successful but provide insights for understanding system behavior. The main problems with this approach are that not every quality attribute has an existing predictive model (e.g. security and usability), not every model may scale well, and not every architecture is amenable to analysis by various models [BBKS05].

A third approach is formed by the intuitive design approaches, such as the Attribute-Driven Design (ADD) [BB01a, BCK03, SEI14a, WBB+06] method and the QASAR [Bos00] method. These methods are effective in organizing requirements but depend to a large extent on the architect to find solutions for the satisfaction of specific quality attributes. In particular, ADD is a recursive method in which a part of the system is selected for decomposition, architectural drivers are identified, architecture patterns and tactics that satisfy them are applied, and pending requirements are refined in terms of the new system organization. Then, by using these methods, the architect incrementally constructs the software architecture by resolving iteratively quality attributes, deciding among existing alternatives in terms of patterns and tactics.

2.1.3 Software Architecture Description

While the main goal of a software architecture design method (SADM) is to design a software architecture that meets stakeholders' needs and expectations, its main deliverable is the description of the architecture of the system being built. Creating an architecture is not enough, it needs to be communicated in a way its stakeholders use it properly to accomplish their goals [CBB+10]. The architecture of a system must be described in enough detail, without ambiguity, and organized so that others can quickly find what they need. Although practitioners tend to build the description after the fact, which generally leads to a too-general or incomplete description, the architecture description must be built during architecture design. The description not only deliver value to stakeholders, but also to the architect as it serves as the container holding the results of the design decisions as soon as they are

made. Moreover, a well-thought documentation scheme can make architecture design more systematical.

P. Clements et al. discuss in [CBB+10] the appropriate term for the activity of building the architecture description: specification, representation, description or documentation. *Specification* implies to render the architecture in a formal language. However, a formal specification is not always practical due to the effort and skills required from the architect to produced it, and required from the stakeholders to consume it. *Representation* denotes a model, an abstraction, a reproduction of the architecture that is separate or different from the architecture itself. While the architecture remains intangible its representation is tangible. While the model is not actually the thing being modeled, a model is tangible and hence it can be communicated, shared and reused, and also very important, feasible to be processed by tools. The term *description* has been widely adopted by the community to refer to the formal languages that specify certain aspects of a software architecture. Although the term description suggests an informal connotation it is used in practice for formal artifacts as well. Last, *Documentation* connotes the creation of a tangible artifact, namely a document, formal or not, containing models (representations) or not, as appropriate. Documents are mainly a vehicle to describe or specify that architecture.

None of these terms has a standardized meaning in the Software Architecture discipline and the distinction between them is rarely used. Moreover, these terms are generally interchanged. However, the essence of the activity is to produce and keep current a tangible work product containing the result of the architecture decisions so that the stakeholders have access to the information they need in a non-ambiguous form. We claim that both *representation* and *documentation* must be the deliverable of a software architecture design method. While a *representation* or model is an internal artifact to be produced, maintained and used by the software architect to capture the design decisions made and to represent the architectural elements characterizing the structure and behavior of the system, one or more architecture *documentation*s offer a communication means to reach the multiple stakeholders with the exact information they need. As maintaining all these artifacts is impractical, tools to generate the documentation from the representation are indispensable.

Currently, the software architecture community is adopting the term *architecture description* not to refer to Architecture Description Languages, i.e. the formal specification languages, but to refer to the artifact produced by the architect during architecture design. The ISO/IEC/IEEE 42010 Standard [ISO11] distinguishes between *architecture* and *architecture descriptions*, and defines the latter as:

> *The work product used to express an architecture.*

The standard does not specify any format or media for recording architecture descriptions. It is intended to be usable for a range of approaches of architecture description including document-centric, model-based and repository-based.

An architecture description consists of an aggregation of architecture views, each addressing a particular set of architectural concerns of the system and targeting a particular set of stakeholders [BH06, RW05]. Which architecture views must be designed depends on the type of system being built, the stakeholders needs and expectations, the skills of the devel-

opment team, the project restrictions on budget, time and quality, among others. Several proposals for views are available, being some of them compliant with the standard conceptualization [CGB+02, HNS99, Kru95, Put00, RW05]. Proposals vary on the concerns addressed, on level of genericity and abstraction of the views and their composing models, on the kind of system they target, and on the languages used to describe it, among others. Although some authors position the Unified Modeling Language (UML) as the one-fits-all architecture description language [SC06], other authors question to what extent it can be considered an architecture description language by itself [GCK02].

Then, we state that an *architecture representation* is, in terms of the standard, a model-based architecture description. Hence, an *architecture representation* is a set of modeling artifacts that captures the architecture of a system of interest, and that is structured accordingly to the ISO 42010 standard. We study software architecture description in depth in Chapter §3, and we thoroughly discuss the distinction between architecture representation and the architecture documentation to be communicated to stakeholders in Section §3.2.4.

2.2 Model-Driven Engineering

Modeling has been an intrinsic and essential activity for men to conceptualize and reason about reality [Béz05b]. According to J. Rothenberg [Rot89], modeling is the cost-effective use of something in place of something else to enable or assist some cognitive purpose. It allows us to use something that is simpler, safer, or cheaper than reality, instead of reality itself, for a given purpose. Modeling underlies our ability to think and imagine, to use signs and language, to communicate, to generalize from experience, to deal with the unexpected, and to make sense out of our raw perception. It allows us to see patterns, to appreciate, predict and manipulate processes and things, and to express meaning and purpose.

Models have played a crucial role in several human activities, mainly in the contexts of science and engineering. Modeling has been incorporated in the Software Engineering practice to deal with the complexity of both the development of software systems, and of software systems themselves. Software developers rely on modeling to plan, analyze, design, validate, document and communicate multiple aspects of software systems. The major advantage of models is that they can be conceived at different levels of abstraction and can deal with specific concerns instead of all concerns at the same time. The system implementation targeting the specific technological platform of the production environment onto which the system is to be deployed, can be considered as the most important artifact in software development. The implementation itself can be considered as a model representing exactly how the system achieves its functionality in the given platform. The programming language or languages used act as modeling languages. Besides, in order to build such an implementation, multiple models are usually required and developed that act at least as enablers to the implementing team. Even agile methods rely on intermediate modeling at the design level to devise and plan the implementation. Model-centric software development processes strongly rely on models at all the disciplines. In these processes, models are not intermediate disposable artifacts, they are part of the set of artifacts that actually compose the software product being built. These models are useful not only during development, but also during maintenance

and evolution. Such processes define which models must be built, which roles produce and consume them, and how to proceed to build them.

In this section, we first introduce the modeling discipline in the context of the Software Engineering discipline, and we review the Model-Driven Architecture framework. Then, we examine the definition of the term *model* and we study the related modeling constructs.

2.2.1 The Modeling Discipline

In the context of Software Engineering, the modeling discipline has emerged as a paradigm shift from code-centric software development to model-based development [Béz05b], promoting the systematization and automation of the construction of software artifacts. The modeling community deals with a variety of acronyms to refer to approaches within the modeling discipline, being Model-Driven Development (MDD) and Model-Driven Engineering (MDE) the most remarkably used. Whether these terms are the same one or not is a matter of interpretation. Some authors claim that MDE is a broader concept than MDD, involving not only software development but also all engineering techniques including evolution, discovery, recovery and migration, among others. However, like in [TPT09], some authors use these terms interchangeably. In this thesis we will use the term MDE.

The essential characteristic of Model-Driven Development is that models are the primary work products and focus on software development [Sel03]. The main idea behind MDD is that it is possible to create models of a system that can be transformed into the deployable and executable implementation of the system [MCF03]. Systems are modeled at several levels of abstraction and perspectives. Models are used to generate other work products automatically or to improve existing ones, by means of model transformations. Additionally, models are transformed into running systems by means of generators or by executing the models at run-time [FR07]. Model transformations are increasingly seen as key assets in software development as they encode, formalize and automate the developers expertise on model manipulation within a software development process [SK03]. A model transformation conforms the mechanization of a refinement step and a chained model transformation a successive refinement needed to obtain a concrete work product [Jéz04], avoiding errors associated with manual manipulation of models [GLZ06].

Model-Driven Engineering was introduced by S. Kent in [Ken02], who aimed to set out a framework to be used as point of reference in the field. For D. Schmidt, MDE technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [Sch06]. MDE combines Domain-Specific Languages (DSLs) with transformation engines and generators. DSLs are languages tailored to precisely match the semantics of a particular domain, in contrast to general-purpose languages that do not express in a direct manner the domain concepts and intent. Transformation engines and generators analyze certain aspects of models and then synthesize various types of work products. These two mechanisms provided by MDE allow to encapsulate the knowledge of a particular domain: domain specific language constructs allow abstract conceptualization of the domain, and transformations allow abstract manipulation and synthesis of such models.

Our position is that MDE encompasses the systematic, disciplined and quantifiable application of scientific and technological knowledge, methods, tools and experience to the construction of model-based techniques. It provides the foundations for Model-Driven Development processes, which deal with the definition and systematization of processes by means of modeling.

Model-Driven Architecture

In the context of MDE, the Model-Driven Architecture (MDA) [Fra02, KWB03, OMG14b] is a framework [OMG03] defined by the Object Management Group (OMG) that is intended to reduce software development complexity and costs by means of MDE techniques. OMG is an industry-driven consortium for developing standards that enable the implementation of Model-Driven Development (MDD) processes, being MDA the most commonly known example nowadays [PTTT09]. MDA focuses on the separation of platform independent and platform specific concerns to improve reusability, portability and interoperability. It introduces three viewpoints for modeling a software system [FR07], computation independent, platform independent and platform specific, defined as [OMG03]:

Computation Independent (CIM). Focuses on the requirements and the environment of the system; the details of the structure and processing of the system are hidden or yet undetermined.

Platform Independent (PIM). Focuses on the operation of a system while hiding the details necessary for a particular platform. A platform independent view shows that part of the complete specification that does not change from one platform to another. A platform independent view may use a general purpose modeling language, or a language specific to the area in which the system will be used.

Platform Specific (PSM). Combines the platform independent viewpoint with an additional focus on the details of how a specific platform can be used by a system.

Related proposals suggest an additional level of abstraction, considering a more concrete viewpoint focused on the implementation details of the system. However, some authors argue that this viewpoint, called Implementation Specific Model (ISM), can be part of the platform specific viewpoint itself.

Then, MDA conforms a framework which guides the direction of model transformations, from abstract models to concrete models. Model transformations encapsulate the knowledge on how to incorporate concrete details (more specific or technology related) to abstract models. The application of this approach resembles that of the Rational Unified Process (RUP). In RUP [Kru03], several models at different levels of abstraction are considered, namely Use-Case, Analysis, Design and Implementation Models, and specific mechanisms are suggested for specifying quality attributes at a given level and realizing them at the lower levels. Such mechanisms are called Analysis, Design and Implementation Mechanisms and somehow guide the refinement of the system models to cope with quality properties.

2.2.2 Modeling Constructs

The term *model* is derived from the Latin word *modulus*, which means measure, rule, pattern, example to be followed [Lud03]. According to H. Stachowiak in [Sta73], an artifact must meet the three criteria to be considered a model:

Mapping criterion: There is an original object or phenomenon that is mapped to the model. The original is called *system*.

Reduction criterion: Not all the properties of the original are mapped on to the model. The model is somehow reduced to reflect a relevant selection of the properties of the original.

Pragmatic criterion: The model can replace the original with respect to some purpose, i.e. the model is useful.

The mapping criterion does not imply the actual existence of the system; it may be planned, suspected or fictitious [Lud03]. Models can be either descriptive or prescriptive [Rot90]. A *descriptive* model is a representation of an existing system. At first glance, descriptive models can only be created after the original. However, descriptive models can be prognostic and describe something that does not exist yet (e.g. a model of the weather). Precisely, the main characterization of descriptive models is that they are not intended to influence the system, their purpose is to describe it. A *prescriptive* model is a representation of a system which intended to be built. Their main purpose is to allow planning and early validation of the system [Küh06], i.e. to serve as blueprints for building systems.

In the context of a software development project, a descriptive model is built to specify the stakeholders' needs. This abstract model is subsequently refined by means of series of prescriptive models. Each prescriptive model is at a lower level of abstraction than its preceding model, refining it and gaining understanding on the system to be built. For instance, a descriptive requirements model is refined onto a prescriptive analysis model, which in turn is refined by a design model, later by an implementation model, an so on. However, in practice, the lack of formality and rigor applied to build models, and the focus on the implementation level of abstraction disregarding the relationships to models at higher levels, tend to reduce the prescriptive effectiveness of models yielding inaccurate and incomplete models. The goal of model-driven approaches is to devote a major effort to generating, maintaining and keeping synchronized all models involved in the development of a software system.

Models

In [Per06] we studied the community understanding of the term *model* and its related terms, and we propose a framework for characterizing its applicability. Provided that the term is broadly used in science and mainly engineering, and that it was incorporated in the software community, several definitions are available and are being revisited successively with minor changes. From our understanding of the term, the following aspects are involved in the concept of models.

(I) What is being modeled. A *model* is a representation of another thing, being such thing a physical, abstract or hypothetical reality [MSUW04]. The model is not the actual thing, a model just describes or represents certain aspects of it. This aspect corresponds to the *mapping criterion* explained above.

A. Kleppe et al. discuss in [KWB03] the need for a particular term to name what is being modeled. We use the term *system* to name what is being modeled. According to the authors, even though the term *system* generally refers to a software system in the Software Engineering discipline, the term is conceived in a broader sense and complies to whatever the target of a model is; for instance, for a business model the system is the business itself. Hence, a model is not the system itself, they are separate things.

The relationship between a model and the modeled system (a.k.a. system under study) is called *representation of* [Béz04] and it was studied by J. M. Favre in [Fav04c].

> *A model is a representation of a physical, abstract or hypothetical reality, called system.*

(II) An abstraction of the system. Some authors use the term *abstraction* to characterize the relationship between a model and the system. The actual system, either a physical entity or not, is generally complex and unmanageable directly. The model is not the actual system, but neither it is a complete detailed representation of the system. It is an abstraction in the sense that it pays special attention to certain characteristics of the system, and excludes everything else [RJB05]. This aspect corresponds to the *reduction criterion.*

A model focuses on the elements and properties of the system that are considered essential, ignoring the others which are considered as details. Which aspects are essential and which are details is a matter of judgement and depends on the intention of the modeler for the model.

> A model is a representation of *(part of)* a physical, abstract or hypothetical reality, called system, *capturing its essential aspects while ignoring the details.*

(III) Level of meaning. The term *abstraction* has also been used in the sense of different level of meaning. On one end, the level of meaning is related to a non-technical human understanding of the system. On the other end, the level of meaning is related to machine dependent models. In the context of Software Engineering, each discipline conforms its own level of meaning. This aspect also corresponds to the *reduction criterion.*

For instance, in the Rational Unified Process [Kru03], while the Requirement and Analysis levels of meaning are at the higher (human-friendly) end, the Design level is intermediate and the Implementation level is at the lower end. As another example, the Model-Driven Architecture [OMG03] framework identifies three levels of meaning, from Computation Independent, through Platform Independent, to Platform Specific models. The separation in levels of meaning is a particular case of separation of concerns.

> *A model is conceived at a particular level of meaning.*

(IV) Purpose and perspective. According to J. Rumbaugh et al. in [RJB05], models are used for several purposes, from capturing and thinking about the requirements, design and implementation of a system, to organize, master and explore alternatives of a complex system. The better understood the purpose, the most useful is the model for this given purpose. Additionally, a model is built from a particular perspective. Several stakeholders are involved in a software development project and different aspects of the system are of their particular interest.

A model focuses on a particular set of concerns for a given perspective and omits those that are irrelevant. Then, a model is partial with respect to the whole set of concerns at a particular level of meaning, but it is complete with respect to the set of concerns that are of interest of certain stakeholders' perspective. Hence, a model serves a given purpose of a particular set of stakeholders and covers a particular set or concerns of the system. This aspect corresponds to the *pragmatic* and the *reduction criterion.*

> A model is conceived at a particular level of meaning, *serves a specific purpose and deals with a particular set of concerns of the system that are of interest of certain stakeholders.*

(V) Constituted by elements. A model is a composed artifact. It consists of a set of *model elements* that are structurally defined by means of properties and interrelated by means of relationships such as composition, aggregation, and specialization, among others.

According to J. Rumbaugh in [RJB05], in the Unified Modeling Language a model comprises a containment hierarchy of packages in which the top-level package corresponds to the entire system. We consider such a description to be too specific, and so we prefer the constituent elements to be of any kind and related by any sort of relationship. A *model element*, together with its properties and relationships, is the representation of a particular characteristic of the system. Thus, the interrelated set of model elements conforms the complete representation of the concerns being modeled.

> *A model is composed by a set of model elements interrelated by means of multiple kinds of relationships.*

(VI) A medium in which it is expressed. A model is actually regarded as an abstract and intangible thing. It needs to be reified, either by a physical or digital object, to make it tangible enabling developers to work with it. A model must be somehow denoted in order to make it persistent in some medium and shared across stakeholders and any other interested audience. Then, developers can actually create and update such denoted model to achieve their purpose.

Models are expressed in terms of *modeling languages*, which define the set of constructs that can be used to express models in the language. This set of constructs establishes, and thus determines and limits, the kind of representation of a given *system* that can be achieved by means of the language. In other words, a particular set of aspects or concerns of a system can be represented by those modeling languages that provide the necessary constructs to fully represent these aspects or concerns at a specific level of meaning and precision. A modeling

language determines both the syntax, either textual or graphical, and the semantics for each kind of construct. The clearer the syntax and the more unambiguous the semantics, the clearer and more unambiguous the models built by means of the language.

Counting with a well-defined modeling language enables the storage and manipulation of models, particularly using computer-assisted tools. In this context, a tool that allows developers to view the constituent elements of a model is called a *model viewer* or *model browser*, a tool that allows developers to change or update a model is called *model editor*, and a tool to persist and store a model is called *model repository*. In addition, and arguably even more valuable to developers, models and model elements can be automatically manipulated. Such tools are called *model transformations* and allow the automatic generation of a given output artifact from a particular set of input artifacts, being these artifacts mostly models. These tools are the basis for model-driven approaches. A computer-assisted modeling environments generally involve these kinds of tools.

> *A model is expressed in terms of a modeling language which defines both the syntax and the semantics of the set of constructs that determines and limits the kind of representation of a system that can be made by models built on this language. A modeling language makes a model tangible and operable, and hence enables the development of computer-assisted tools for viewing, editing, storing and transforming models.*

Definition

Summing up these six aspects, we define the term *model* as following:

> A *model* is a representation of (part of) a physical, abstract or hypothetical reality, called *system*, capturing its essential aspects while ignoring the details. A model is conceived at a particular level of meaning, serves a specific purpose and deals with a particular set of concerns of the system that are of interest for certain stakeholders. A model is composed by a set of *model elements* interrelated by means of multiple kinds of *relationships*. A model is expressed in terms of a *modeling language* which defines both the syntax and the semantics of the set of constructs that determines and limits the kind of representation of a system that can be made by models built on this language. A modeling language makes a model tangible and operable, and hence, enables the development of computer-assisted tools for viewing, editing, storing and transforming models.

Metamodels

In the language theory a *language* is defined as a set of sentences, using the term *set* as in set theory. As languages are abstract systems, there is a need for a practical means to deal with languages, leading to the derived notion of *model of a language* [FN05]. For instance, a grammar is a model of a language whose sentences are constructed by means of the grammar's alphabet, using the defined tokens and grammar rules. Models of languages must

not be confused with modeling languages (a.k.a. languages of models). While the former is a representation (model) of a language (system), the latter is a set (language) of the models (sentences) expressed in the language. In fact, according to J. M. Favre in [FN05], one of the most important concepts in Model-Driven Engineering is the concept of *models of languages of models*, i.e. *models of modeling languages*. These models are called *metamodels*. The relationship between languages and metamodels was studied by J. M. Favre in [Fav04b].

The Meta Object Facility (MOF) specification [OMG11b] generally defines a *metamodel* as a model used to model modeling itself. For E. Seidewitz in [Sei03], a *metamodel* is a specification model for a class of systems where each system in the class is itself a valid model expressed in a certain modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language. For J. Bézivin in [Béz05b], a *metamodel* is a formal specification of an abstraction, usually consensual and normative. From a given system we can extract a particular model with the help of a specific metamodel. A metamodel acts as a precisely defined filter expressed in a given formalism. Metamodels facilitate the separation of concerns, and promote the definition of Domain Specific Languages for accurately coping with the representation of the multiple aspects of a system.

Then, a metamodel is an explicit model of the constructs and rules needed to build specific models within a domain of interest. Then, a model must *conform* to its metamodel. This conformance relationship was introduced by J. Bézivin in [Béz05b] to characterize the relationship between a model and the model of the modeling language (metamodel) used to express the model. Figure 2.2 illustrates this relationship.

As metamodels are also models, they must be precisely defined by means of a modeling language. Then, the concept of *metametamodel* is introduced as the model of a language used to define metamodels. In other words, a metametamodel models a domain specific language for defining metamodels. In the same sense as a model conforms to its metamodel, a metamodel conforms to its metametamodel. The particular characteristic of metametamodels is that each metametamodel conforms to itself, i.e. the language for the metametamodel must allow the representation of the metametamodel itself. Concrete metametamodels include MOF [OMG11b], ECore [SBPM09] and KM3 [JB06].

The Unified Modeling Language (UML) specification [OMG11c] introduces a four-layer metamodel hierarchy to organize the concepts of system, model, metamodel and metametamodel, together with their relationships. Quoting J. Bézivin in [Béz05b], this classical structure should more precisely be named a *3+1* organization provided that the relationship between the system and the model is of a different nature. Figure 2.2 illustrates the 3+1 organization. The left side of the figure depicts the system, which is the real world thing being modeled. Real world systems conform the M0 level in the UML specification. The right side of the figure depicts the modeling world containing the three kinds of models. The M1 level at the bottom consists of a model which is a representation of the system. Models in this level are called *terminal models*. The terminal model conforms to its metamodel at the M2 level. In turn, the metamodel conforms to its metametamodel at the M3 level, which conforms to itself. Metamodels and metametamodels are called *reference models* as their intention is to specify the modeling languages for other models.

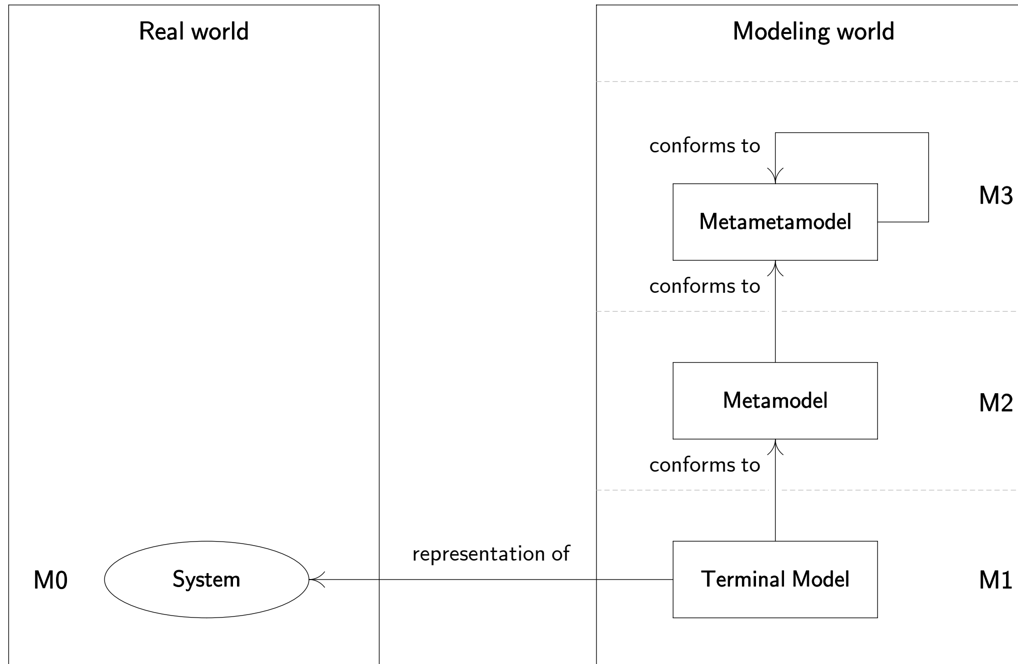


Figure 2.2: *3+1 organization of the metamodeling approach.*
This figure is a reproduction of the 3+1 refinement proposed by J. Bézivin in [Béz05b] of the four-layer metamodel hierarchy of the Object Management Group. It illustrates a real world system represented by a terminal model, and the conformance relationship between a model and its metamodel, the metamodel and its metametamodel, and the metametamodel to itself.

We discussed so far what is known as linguistic metamodeling, which is concerned with the use of metamodels as a language definition tool. As studied by C. Atkinson et al.in [AK03], this dimension of metamodeling allows the definition of the concepts that are available for the creation of models and the rules governing their use, and how the elements of a model relate to (i.e. represent) real world elements. However, this dimension does not facilitate the creation of domain-specific extensions of existing metamodels and the creation of models conforming to them. This extension capability involves the addition of new types or the specialization of existing ones, in order to better characterize a specific domain. This dimension is called ontological metamodeling. This dimension is recognized also by other authors that claim that the notion of metamodel is strongly related to the notion of ontology [Béz05b]. The ontological dimension is orthogonal to the linguistic dimension; not only both metamodeling dimensions are valid, but also useful. As discussed by R. Gitzel et al. in [GH05], a layer of the linguistic hierarchy (for instance M2 level in Figure 2.2) can be structured in terms of a hierarchy of metamodels, producing a non-linear hierarchy for metamodels. Current tool support, however, tend to prioritize the linguistic dimension relegating the support for the ontological dimension.

Model Weaving

In a software development project, developers build several models to represent different aspects of concerns of a given system. As we discussed before, each model is constituted by a set of model elements with its properties, and various kinds of relationships between these elements. Separating the representation of a system in multiple models allows dealing with complexity by means of separation of concerns. However, these models represent concerns of the same system and, as such, it is expected that there are correspondences between them. In the same sense as we have relationships between model elements in the same model, we have relationships between model elements in different models. These kind of relationships between model elements in separate models is called *model weaving*. A model weaving is essentially a mapping, possibly n-ary, between model elements, usually residing in different models. It can be regarded as the mathematical concept of *relation* if each model is regarded as a set of model elements.

Different application scenarios for model weaving were studied by M. Del Fabro et al. in [FBV06]. Model weavings can be used for tool interoperability. Different tools use separate models usually conforming to different metamodels. As we discuss later, developers use model transformations to generate a model used by a target tool from a model used by a source tool. Then, the knowledge of how the metamodels are interrelated is implicit in the model transformation. A model weaving between the metamodels (recall that metamodels are also models) allows the developers to abstractly define their correspondence independently of any model transformation language. Model weaving between metamodels is also useful for model alignment and model composition. In the latter, a model weaving defines the relationship of the concepts in different domains (two separate metamodels) an so specifying how a the model composition should proceed. Currently, practical applications of model weaving are being published [VCFM10].

One of the most important usages of model weavings is to allow the developers team to establish correspondences between the elements in the terminal models representing the system. Model weavings establish semantic relationships between model elements representing different aspects or concerns of the system. The alternative of solely using naming conventions to relate elements, i.e. use a name in one model to refer to an element of another model that has the same name, is error prone and generally leads to inconsistent models. A model weaving can be used to define such correspondence, enabling the automatic navigation between model elements in different models, and to define richer relationships with its own properties on each particular link between elements. R. Salay et al. in [SME09] introduce *macromodels* as a collection of models on which a developer can define relationships to accurately express the modeler's intent. The concept of *macromodel* is strongly related with that of *megamodel* that we discuss later in Section §2.3. Macromodels are also related to model weavings as they allow semantic relationships between models. However, model weavings allow the definition of the semantic relationships at the model element level, not just at the model level.

A special usage scenario for model weavings is that of model annotation. Annotations allow developers to capture additional information on the model elements of given model. In the case that the metamodel of the given model provides the necessary constructs for

representing such additional information, it can be directly captured in the model itself. However, if the metamodel does not provide such capability, annotations must be captured separately from the model. This might be regarded as a limitation of the metamodel, but also it can be regarded as a technique for separation of concerns. While the metamodel focuses on a particular purposes, by means of annotations residing outside the model developers can capture others concerns in isolation. However, annotations are related to the model as they provide additional information regarding model elements in this model. Model weavings are used to support this requirement. In this special case, model weavings of arity 1 captures the annotations and attaches them to the references to the model elements defined in the woven model.

A model weaving is an abstract entity that need to be reified in order to be stored and processed. Model weavings are captured by means of *weaving models* which are constituted by the different kinds of links between the model elements being woven. The special characteristic of weaving models that distinguishes them from other terminal models is that they are not self contained [FBV06], i.e. a weaving model is useful only if the related models also exist. The links between model elements have different semantics depending on the application scenario. Hence, different weaving metamodels may be defined to express different semantic relationships. Figure 2.3 illustrates these concepts and their relationships. A model weaving W defining semantic relationships between models MA and MB is represented by a weaving model MW that conforms to the weaving metamodel MMW. The weaving model MW takes models MA and MB as its woven models, annotated as *left* and *right* provided that, in the example, W is a binary model weaving. Models MA and MB conform to their corresponding metamodels MMA and MMB, and all metamodels conform to the same metametamodel MMM.

Weaving models can be manually created by a developer by means of a model editor specific for this purpose. Also, as weaving models are models, they can be automatically created by means of model transformations. Particularly, as we discuss later, a special kind of weaving models are tracing models which define the links between the model elements in a the source model and the resulting model elements in the target model that were created by the execution of the model transformation on the source model.

The AtlanMod Model Weaver (AMW) [FBJ+05] provides tool support for model weaving. Provided that weaving models are mainly about link management, AMW defines a *core weaving metamodel* to support the common requirements [FBV06]. Although this metamodel can be used as a metamodel for weaving models, it uniquely establishes the presence of woven models, its referenced elements, and the links between them. The core metamodel is expected to be extended by developers to encapsulate the knowledge of the multiple semantic relationships between model elements in the different models built during software development. AMW provides the tool support to create such concrete extensions. The approach follow by AMW is an example of application of ontological metamodeling allowing to define domain-specific weaving metamodels by extending or specializing an existing one.

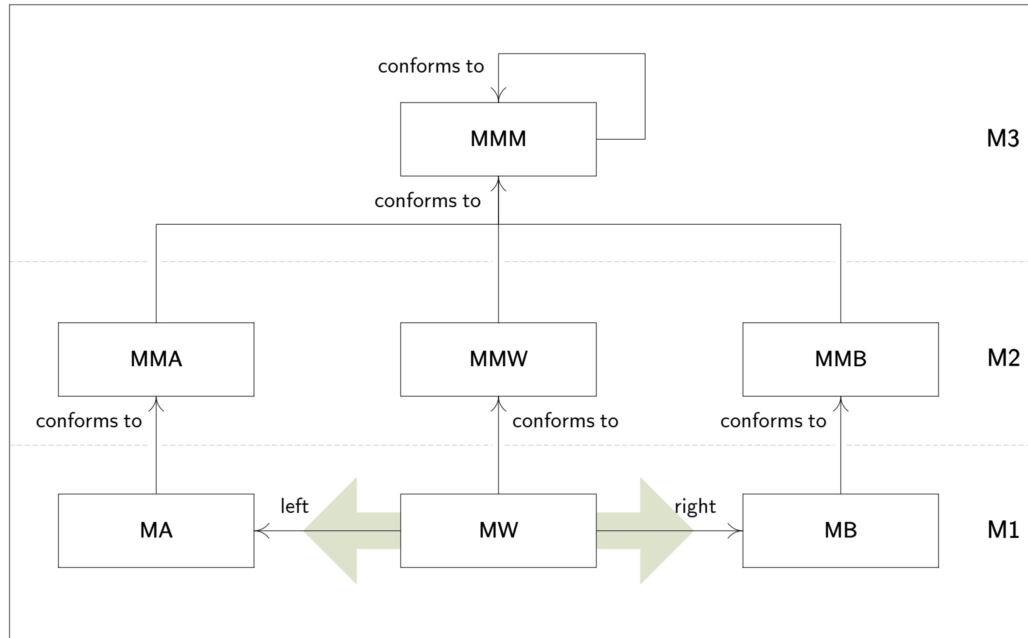


Figure 2.3: *Model weaving.*

This figure illustrates the concept of model weaving in the particular case in which the woven models are terminal models. The model weaving is represented by the weaving model MW that conforms to its weaving metamodel MMW. The weaving model MW is related to the two terminal models MA and MB which model elements are woven by MW.

Model Transformations

In a MDE-based software development project, models are work products built and used by developers to represent certain aspects of the system, facilitating planning, analysis, design, validation, documentation and communication. Using a model consists of exploiting the information contained in it for some particular purpose, such as generating another model or work product, or updating an existing one. Developers use *model browser* tools to visualize and explore the constituent model elements, their properties and interrelationships, and *model editor* tools to update these elements.

However, one of the key prerogatives of Model-Driven Engineering is that models are useful beyond system representation. Those models that are informally expressed as sketches or drawings in paper or whiteboards, or those that are part of electronic documents by means of figures and explained in plain text, provide limited benefits with respect to automating their manipulation. However, those models that are expressed using rigorous modeling languages like the ones corresponding to metamodels, and that are stored in a model repository, are machine processable. Hence, a computer program can use this kind of models as inputs and/or outputs, thus automating the processing and generation of models. These programs are called *model transformations.* A model transformation encapsulates the developer's knowledge on how to extract information from a model and on how to update the model to include aspects or details that were still missing. According to K. Czarnecki et al. in [CH06], the

most common intended applications of model transformations are:

- generating lower-level models, and eventually code, from higher-level models;
- mapping and synchronizing among models at the same level or different levels of abstraction;
- creating query-based views on a system;
- model evolution tasks such as model refactoring; and
- reverse engineering of higher-level models from lower-level ones.

Model transformation is closely related to program transformation [PS83]. The most important differences are the nature of what is being processed, and that in model transformation multi-directionality and model element mapping and traceability are a more common use scenario.

A model transformation is a function in the mathematical sense of the term [FN05], i.e. it is a relation between source models and target models in which each source model has a unique corresponding target model. Each pair is called a *model transformation instance*. Then, a model transformation is an abstract entity that needs to be somehow reified to be operational. Model transformation languages define the constructs that are rules for model transformations. Then, a model transformation is represented by a model, namely a *transformation model*, that conforms to its metamodel (*transformation metamodel*) that represents the model transformation language [BBG+06]. Examples of model transformation languages and their corresponding metamodels are ATL (AtlanMod Transformation Language) [JABK08, JK05] and QVT (Query/View/Transformation) [OMG11a].

Figure 2.4 illustrates these concepts and their relationships. A model transformation T is represented by a transformation model MT. The transformation model MT conforms to its metamodel MMT. The transformation model takes source models conforming to the source metamodel MMA on to target models conforming to the target metamodel MMB. The figure also depicts the model transformation instance, i.e. the application of the model transformation T on a particular source model MA to a particular target model MB. The model transformation instance itself is actually represented by a model serving as the record of the application of the model transformation; the model transformation record is named MR in the figure. The model transformation record conforms to its own metamodel. All metamodels conforms to the same metametamodel MMM that determines the technical space in which the modeling artifacts are expressed.

Provided that a transformation model is actually a model conforming to its own metamodel, it can also be manipulated by means of model transformations. A transformation whose source and/or target models are transformation models is called *higher-order model transformations* (HOTs) [TJF+09]. Practical uses of HOTs include model transformation analysis, generation of model transformations on-the-fly when metamodels are not known a priori, internal composition of model transformations, variability in model transformations, aspect weaving in model transformations such as for debugging or traceability purposes, among others.

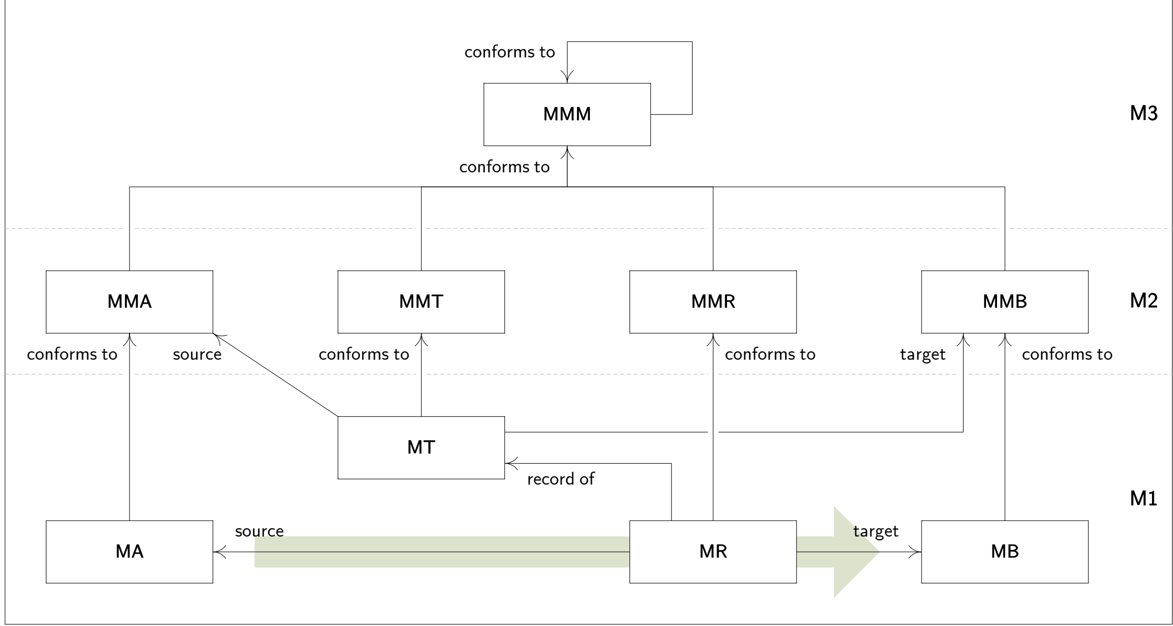


Figure 2.4: *Model transformation.*

This figure illustrates the concept of model transformation, in the particular case in which the participant models are terminal models. It is based on the proposal of J. Bézivin et al. in [BBG+06], but we distinguish between the model transformation and the example of its instance. The model transformation is represented by the transformation model MT, which refers to the metamodels of the potential source and target models but it is not related to any particular model (as in [BBG+06]). In addition, we represent an example of a model transformation instance by the model transformation record MR that is related to actual source and target models.

K. Czarnecki et al. define in [CH06] a framework for the classification of model transformations by means of a feature model to characterize the different aspects involved. The authors identify eight top-level features that we classify in two: internal and external properties of model transformations. Internal properties refer to the choices in the *design* of the model transformation, such as the transformation rules, rule application control and rule organization. External properties refer to the characterization of the model transformation with respect to their usage, i.e. properties of a model transformation that are of interest to the user (developer) of the model transformation. External properties are:

Specification. A model transformation represented in a model transformation language (i.e. a transformation model) defines how the transformation works, how a source model is processed in order to obtain a target model. The specification of a model transformation defines what is the intention of the model transformation, independently of how this intention is realized. Some model transformation languages provide constructs for specification, for instance, by means of pre- and post-conditions.

Tracing. Tracing is understood as the runtime footprint of the execution of a model transformation. A common form of trace information are traceability links connecting model elements in the source model with the corresponding model elements in the target model. Traceability support can be achieved by means of higher-order model transfor-

mations (that take a transformation and generate a new transformation that produces not only the target model but also the traceability information). Additionally, some languages such as ATL and QVT provide dedicated support for tracing. In the case of ATL, a weaving model is created that maps source and target model elements according to the effect of the execution of the model transformation.

Source-target relationship. Some model transformations are intended to process an existing model and to update it, i.e. the source and target model are the same model. The design of this kind of transformations refers to the exact modification that is expected, instead of the specification of how a new target model needs to be created from a source model. Some approaches force the generation of a new target model, while others allow in-place transformations. In the case of ATL, although it mandates the creation of a new target model separate from the source model, in-place model transformations can be simulated through an automatic copy mechanism provided by the model transformation engine. For this kind of transformations, the key functionality is not whether a new target model is created or if an existing one is updated, the key functionality is that the model transformation language allows to define what needs to be updated in the source model.

Incrementality. There are three different aspects with respect to incrementality in model transformations. The *Target-incrementality* refers to the ability to update existing target models based on changes in the source models. In other words, whether changes on the source models are propagated to the target models. The first execution of a target-incremental model transformation will create the target models if they are missing. A subsequent execution with the same source models has to detect that the needed target elements already exists. This can be achieved by means of traceability links. When any of the source models is modified and the transformation is executed again, the necessary changes are determined and applied. *Source-incrementality* refers to minimize the portion of the source model that needs to be re-examined by the transformation when the source model is modified. It is analogous to incremental compilation in programming languages [Rei84]. Finally, a model transformation may be re-executed in order to synchronize the source and target models, but in a scenario in which the target models were manually edited by the user. The *preservation of user edits in the target*, although desirable, is a complex functionality and then it is seldom supported.

Directionality. Transformations may be unidirectional or multidirectional. A unidirectional transformation creates or updates a target model based on a source model. Multidirectional transformations can be executed in multiple directions, from source to target or vice versa, and are useful in model synchronization. A multidirectional transformation can be supported by the model transformation language, by providing support for multidirectional correspondences (rules) or by defining several separate complementary unidirectional transformations.

2.3 Global Model Management

In the mid 1970's the Software Engineering discipline started a major shift on its principles and goals due to the kind of software systems that needed to be built, their higher complexity and larger scale. In 1976, F. DeRemer and H. Kron in [DK75] made explicit the difference between programming in the small and programming in the large, moving research towards formal methods, development processes and a new set of programming languages and tools. The notion of *megaprogramming* was introduced later, in 1992, by B. Boehm and Scherlis in [BS92], moving research towards component-based development and software architecture of large systems. The Model-Driven Engineering followed similar steps but in a shorter timespan. The initial effort was mainly devoted to the definition of languages, methods and tools for modeling and model transformation development, i.e. in modeling in the small. Although this effort still continues yielding improved techniques and tool support, the community is also addressing the need for dealing with large number of modeling artifacts, which are actually required in large and even in medium-scale development scenarios.

Analogously to the evolution of software programming, in 2004 J. Bézivin et al. introduced the notion of *megamodel* [BJV04] and discussed the difference between modeling in the small and modeling in the large [BJRV04]. The term *megamodel* conveys the idea of modeling in the large, establishing and using global relationships and metadata on the basic macroscopic modeling constructs (models, metamodels, transformations, etc.) ignoring the internal details of these constructs. Similarly, J. M. Favre also defined in [Fav04d] his notion of megamodel, providing a formalization framework using set theory. This approach has major similarities to the one defined in [BJV04]. Later, R. Salay et al. introduced in [SME09] the notion of *macromodels* to allow developers to define semantic relationships between models. Macromodels bear similarity to megamodels as they express macroscopic properties and relations of modeling constructs. Macromodels is not an approach to modeling in the large in the general sense. Its goal is to allow developers to express their intent in a development process by means of the relationships between models. However, the megamodeling approach defined in [BJV04] is a general purpose approach. It is not specifically targeted to tackle semantic relationships in development processes. Its goal is to provide the foundations for specification of the interrelations of modeling artifacts in general, providing also extension mechanisms to achieve the definition of semantic relationships.

Software development projects, either following a Model-Driven Development process or simply using Model-Driven Engineering techniques, are intensive in modeling artifacts. An industrial project may involve a huge number of various and varied modeling artifacts [Mod08b]. Refining those identified in [Mod08b], we identify the following characteristics for this set of artifacts:

Large & complex. The increasing scale and complexity of software systems has naturally an impact on the scale and complexity of the models conforming the representation of such systems. When building these models, they soon become populated by a huge amount of interrelated model elements.

Heterogeneous. The modeling artifacts are of different nature. Although the main principle of Model-Driven Engineering establishes that all modeling artifacts are actually conceived

as models [Béz05b], these models can be further categorized. Models can be terminal models, metamodels, metametamodels, weaving models, transformation models, among others. Additionally, each kind of artifact exhibits different properties, relations and behavior.

Evolving. The taxonomy of modeling artifacts evolves as new techniques and technologies emerge. The kinds of artifacts that are available in a modeling environment or that can be stored in a model repository are not yet fixed and keep growing, potentially embracing artifacts of different nature that are even not envisioned yet within the discipline. It is important to notice that this evolution takes place in two separate dimensions. On the one hand, artifacts of different nature of those already conceived might emerge, i.e. a new kind of modeling construct. On the other hand, new technologies and tools may be developed to refine the existing kinds of artifacts already known, i.e. a new specialization of the current constructs. Although both directions are possible, the latter is the most probable to happen.

Interrelated. Modeling artifacts are related to each other through strong semantic relationships. As we discussed in Section §2.2, there is a *conformance* relationship between a model and its metamodel, a weaving model is related to the models being woven, a model transformation is related to the metamodels of the acceptable source and target models; these relationships are illustrated in Figures 2.2, 2.3 and 2.4. Moreover, developers might define new semantic relationships between models as proposed by the *macromodeling* approach [SME09].

Distributed. As we discussed before, model repositories are used to store modeling artifacts. The model repository for a development project does not necessarily contain all artifacts required for the project. Modeling artifacts can be distributed in multiple repositories, even in public locations or ones external to the development company. The diversity of the kinds and purposes of these artifacts promote sharing and reuse by means of publishing them in shared repositories making them available to the developers community. For instance, modeling zoos are public repositories of lots of modeling artifacts.

Megamodeling is a model-based approach for coping with these challenges imposed by modeling in the large [BJB08], centered in the notion of *megamodel*. Global Model Management (GMM) [Mod08b] is the megamodeling approach proposed by the AtlanMod Team at INRIA, which provides both a conceptual framework for dealing with medium- and large-scale applications of Model-Driven Engineering to software development and the required tool support for manipulating and storing modeling artifacts.

In this section, we first present the conceptual framework for the megamodeling approach, and we define our metamodel for megamodels strongly based on the metamodel defined by AtlanMod's GMM metamodel. Second, we review the current tool support available for megamodeling.

2.3.1 Conceptual Framework & Metamodel

A *megamodel* is a representation of a collection of modeling artifacts, their properties and the relationships among them. Thus, a megamodel is a model and its model elements represent modeling artifacts that are available in a model repository or that are reachable in a modeling environment. A megamodel does not actually contain the modeling artifacts, it is a model that defines the metadata of existing modeling artifacts that reside in a model repository. Similarly to a weaving model and its related woven models, a megamodel is not self contained. It is useful only if the represented modeling artifacts actually exists and are reachable in the modeling environment. Provided that a megamodel is actually a model, it needs to be preserved in a model repository and it must conform to a particular metamodel.

As we discussed above, the domain of megamodeling is still evolving as new kinds of artifacts might be conceived and, particularly, new technologies and tools might be developed. Then, the task of defining a metamodel for a megamodel is not as easy task as it must embrace these kinds of evolution. To this purpose, the Global Model Management (GMM) approach to megamodeling uses an incremental definition of such a metamodel. The conceptual framework for GMM is described in [Mod08b], presenting the taxonomy of models and their relationships. A more complete definition of the metamodel is introduced in [Mod09], but it is intertwined with the definition of the AtlanMod MegaModel Management (AM3) tool. As we discuss later, this definition of the metamodel and its supporting tool is structured in a hierarchy of metamodels that modularly describe the megamodeling constructs supported by the tool.

For the purpose of our work, our requirement on the metamodel of the megamodel is twofold. First, we need a technology-independent metamodel that covers all the main concepts and techniques available in Model-Driven Engineering. We use this metamodel to define our model-based interpretation of Software Architecture Description and Design in Chapters §3 and §4, respectively. Second, we need this metamodel to be actually supported by current tools on modeling in the large, and that can be extended to support the modeling technologies that we use in our case study. To achieve these requirements, our definition of the metamodel of the megamodel is inspired in the conceptual framework for GMM [Mod08b], and it is strongly based on the core metamodels defined in the AM3 tool. Particularly, we merge the *AM3Core* metamodel and its top-level extension called *GlobalModelManagement* to define our metamodel. We do not include the other extensions as they are related to specific technologies. However, we actually use them when developing our case study. By means of our merged metamodel, while providing a single, abstract and complete metamodel for our interpretation, we still preserve compatibility with the AM3 tool support. We explicitly exclude from our metamodel those metaclasses regarding tool-related aspects of modeling artifacts, such as element identification and localization. Other minor differences are detailed as we present the metamodel in what follows.

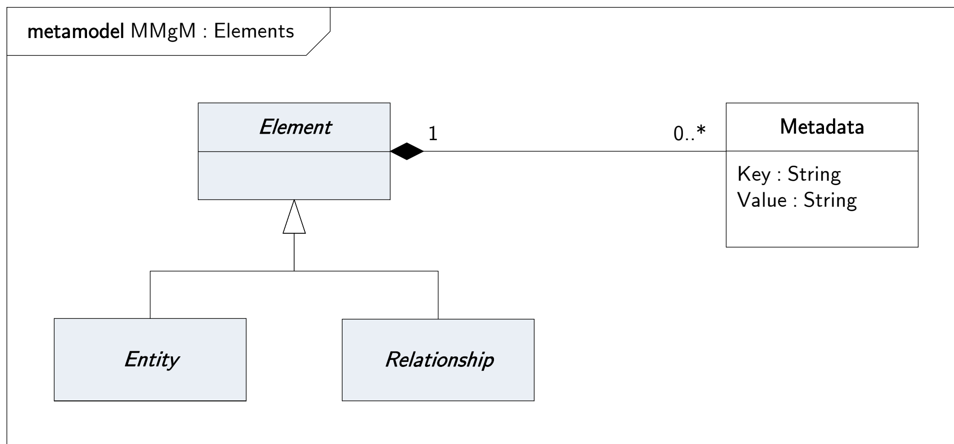


Figure 2.5: *Megamodel metamodel : Elements.*
This figure illustrates the metaclasses conforming the root of the class hierarchy of the metamodel of the megamodel.

Metamodel for megamodels

We use UML Class Diagrams to illustrate the metaclasses and metarelationships conforming the metamodel of the megamodel. In the diagrams, we use grayed metaclasses to emphasize the fact that they are abstract. These metaclasses cannot be directly instantiated and hence they cannot be used as a construct for the megamodel by themselves, a concrete sub-metaclass must be used. Figure 2.5 depicts the top-level categorization of elements in a megamodel. The abstract metaclass Element is the root of the meta-class hierarchy of the metamodel. Then, a *megamodel* is basically defined as a set of elements, i.e. the set of model elements that are instances of Element. Elements can be annotated with *metadata*, mainly key/value pairs that might be used by developers and tools to attach additional data to elements that is not actually modeled by means of a construct in the metamodel. To this end, each Element instance can contain a set of Metadata instances. There are two main sorts of elements that define the basis of a megamodel, *entities* and *relationships*. Entity is the abstract metaclass that represents actual modeling entities, i.e. actual modeling artifacts stored in the model repository or reachable in the modeling environment. Relationship represents a semantic relationship between elements and does not represent actual entities in the repository. These metaclasses are defined in the *AM3Core* metamodel and provide the foundational structure for all modeling constructs to be added in the metamodel. We omit other classes from the *AM3Core* metamodel, such as Container, Group and Chain, as we do not use them in our work.

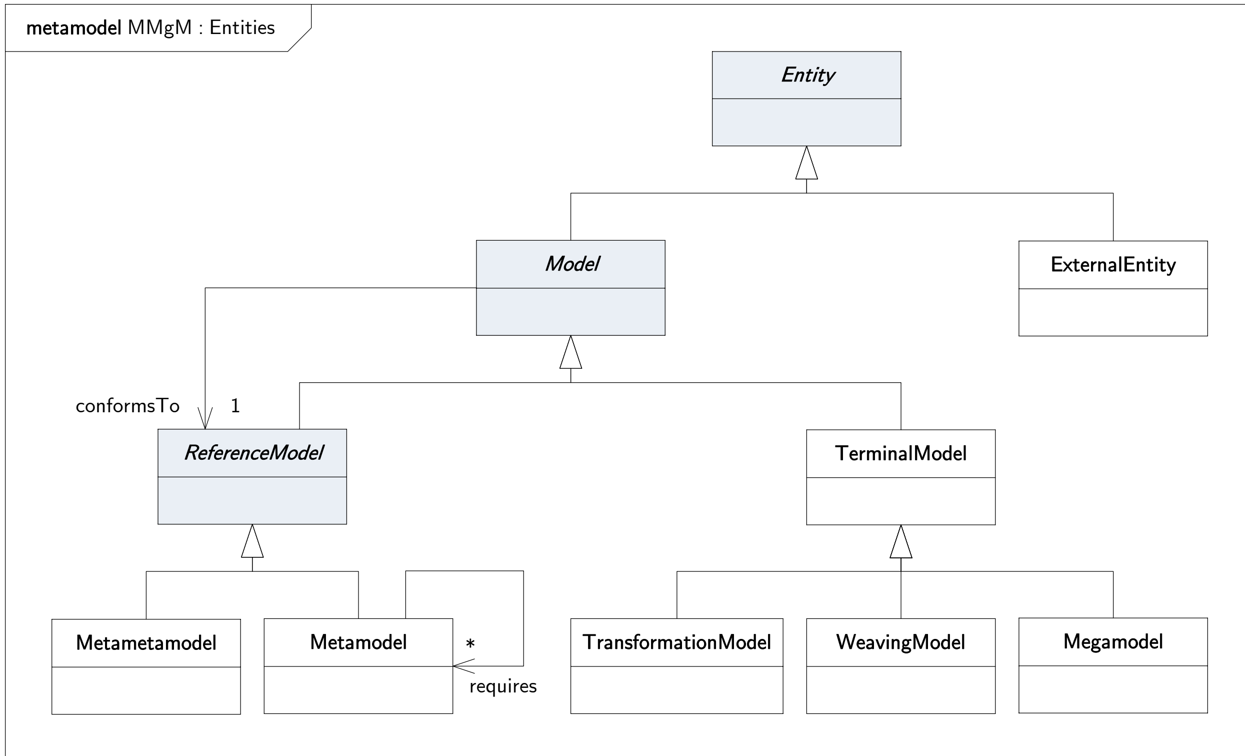
Entities. An Entity can be further categorized in Model and ExternalEntity. While *models* are actual models in the modeling technical space, *external entities* refer to any other kind of non-model artifact that is available in the development environment. External entities denote documents or resources (files for instance) that do not belong to the modeling technical space. As we describe later, model transformations can use external entities as input in order to import information into a model or as output in order to export information. These model transformations are called injectors and extractors, respectively. External en-

tities are generally used with tools that do not conform to the modeling discipline. The ExternalEntity metaclass does not belong either to the *AM3Core* or *GlobalModelManagement* metamodels, while the former introduces the metaclass Entity, the latter introduces the Model metaclass. Another sub-metaclass of Entity is defined in the *GMM4TCS* metamodel, namely TextualEntity. As we will discuss in Section §2.3.2, the *GMM4TCS* extension defines the specific constructs for the Textual Concrete Syntax (TCS) [JBK06] technology. In *GMM4TCS*, TextualEntity represents non-model textual entities belonging to the grammarware technical space. We introduce ExternalEntity as the super-metaclass for any non-model artifact. Then, TextualEntity would be a sub-metaclass of ExternalEntity as it is a particular case of non-model entity.

The Model abstract metaclass represents *model* entities. As we discussed in Section §2.2, a *model* can be either a *terminal* or a *reference model*, each of them conforming to a *reference model*. In turn, reference models can be either a *metamodel* or a *metametamodel*. These concepts are represented by metaclasses with the same name, defining the class hierarchy for Model. The *conforms-to* relationship is represented by the association between a model and a reference model. The 3+1 organization of metamodeling [Béz05b] is not enforced structurally, it is established by means of invariants in the metamodel. The ontological dimension to metamodels is not considered in the GMM approach, even though the existing tool support allows the reuse (reference) of metamodels from other metamodels. This tool capability is essential to allow domain-specific extensions to existing metamodels. To this end, we add an association to represent that a metamodel *requires* or imports the definitions of other metamodels to define its own constructs. The invariant states that the *requires* is acyclical. We represent this extension mechanism solely for metamodels as they belong to the M2 level of the 3+1 organization, and are the main target of domain-specific extensibility. Figure 2.6 depicts these concepts in the metamodel.

Although TerminalModel is a concrete metaclass, it is important to distinguish some *terminal models* that have a specific purpose in model engineering, particularly, *weaving models* and *transformation models*. By means of a UML Object Diagram, Figure 2.7 illustrates the constituting model elements of a megamodel containing the elements from Figure 2.2.

Figure 2.6 also illustrates the Megamodel metaclass that deserves special attention. The Megamodel metaclass represents the terminal models in the model repository, or available in the modeling environment, that are actually megamodels. Hence, within the collection of elements conforming a *megamodel* MgM, we can have any number of terminal models which, in particular, some of them might be *megamodels* themselves. In order to explain this in detail, let MgM1 be the megamodel whose constituent elements are illustrated in Figure 2.7. The Object Diagram in the figure expresses that the megamodel MgM1 is constituted by seven model elements <u>representing</u> seven modeling artifacts stored in the model repository or reachable in the modeling environment. In particular, the seven modeling artifacts represented are the metametamodel MMM, three metamodels MMA, MMW and MMB, two terminal models MA and MB and the weaving model MW. The megamodel MgM1 is actually a modeling artifact stored in the model repository, but in MgM1 there is no model element representing itself, what is represented is the collection of modeling artifacts referred by MgM1. To include a model element E (instance of the metaclass Element) in a megamodel represents the fact that the modeling artifact $\mathsf{A_E}$ represented by the model element E is part of the collection



context Metametamodel **inv: self**.conformsTo = **self**

context Metamodel **inv: self**.conformsTo.oclIsKindOf(Metametamodel)

context TerminalModel **inv: self**.conformsTo.oclIsKindOf(Metamodel)

context Metamodel **inv: self**.requires→closure(mm | mm.requires)→excludes(**self**)

Figure 2.6: *Megamodel metamodel : Entities.*
This figure illustrates the metaclasses conforming the sub-hierarchy of entities in the metamodel hierarchy in the metamodel.

of modeling artifacts of the megamodel. Then, to include a model element E1 representing the modeling artifact $MgM1_{E1}$ means that the collection of elements of MgM1 includes MgM1 itself. Now, let MgM2 be the megamodel illustrated in Figure 2.8. The collection of elements of MgM2 contains three modeling artifacts, the metametamodel MMM, the metamodel for megamodels MMgM and the megamodel MgM1. Although MgM2 contains MgM1 in its collection, the modeling artifacts in MgM1 are not necessarily included in MgM2, particularly, except for MMM, all other artifacts in MgM1 are not part of MgM2. To sum up, a megamodel does not necessarily contain an element representing itself, and to contain a megamodel does not imply to contain its elements.

The previous discussion is important in order to identify what we consider an inconsistency in the definition of the metamodel of the megamodel in AM3. In AM3, there are two metaclasses representing *megamodel*s. *AM3Core* defines the metaclass Megamodel which has a composition association to a collection of Element [Mod09]. This metaclass is intended to represent the megamodel that is being modeled. In terms of our example, the meg-

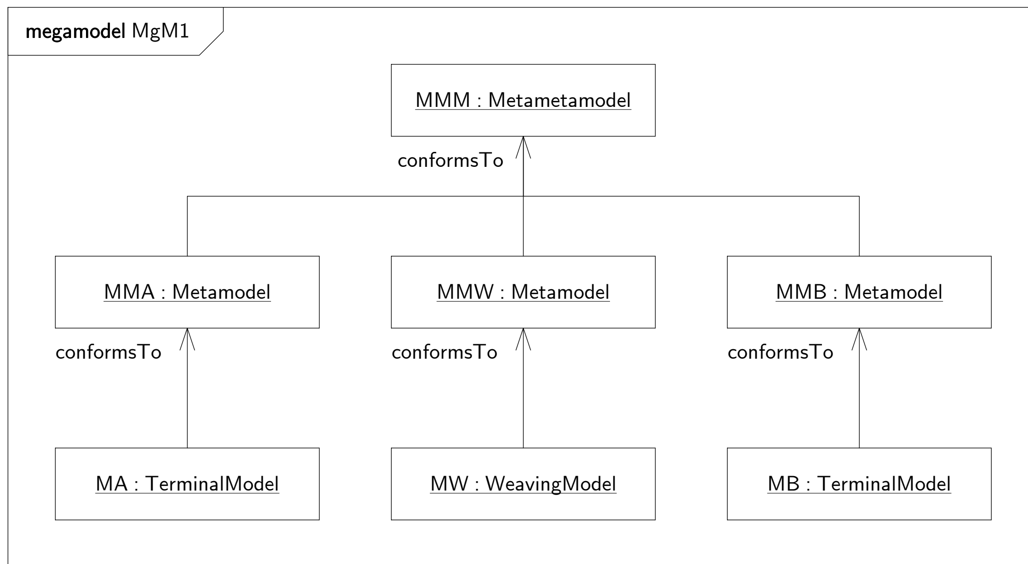


Figure 2.7: *Example of the constitution of a megamodel.*

This figures uses a UML Object Diagram to illustrate the constituting model elements of a megamodel containing the elements used in Figure 2.2.

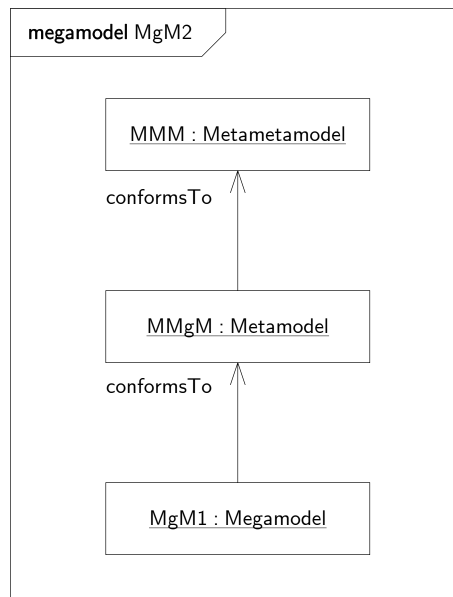


Figure 2.8: *Example of a megamodel containing a megamodel.*

This figure uses a UML Object Diagram to illustrate the constituting model elements of a megamodel that contains another megamodel.

amodel MgM1 would have contained an instance of the metaclass Megamodel of *AM3Core* representing itself. In the metamodeling discipline, including a metaclass representing the system itself, being the root of the containment hierarchy and containing all top-level elements is a design decision of the metamodeler. In our metamodel, we prefer not to include such a class because we consider it to be misleading from a conceptual point of view. The *GlobalModelManagement* extension to *AM3Core* introduces another metaclass MegaModel (notice the capital letter in the metaclass name) to represent megamodels. This metaclass

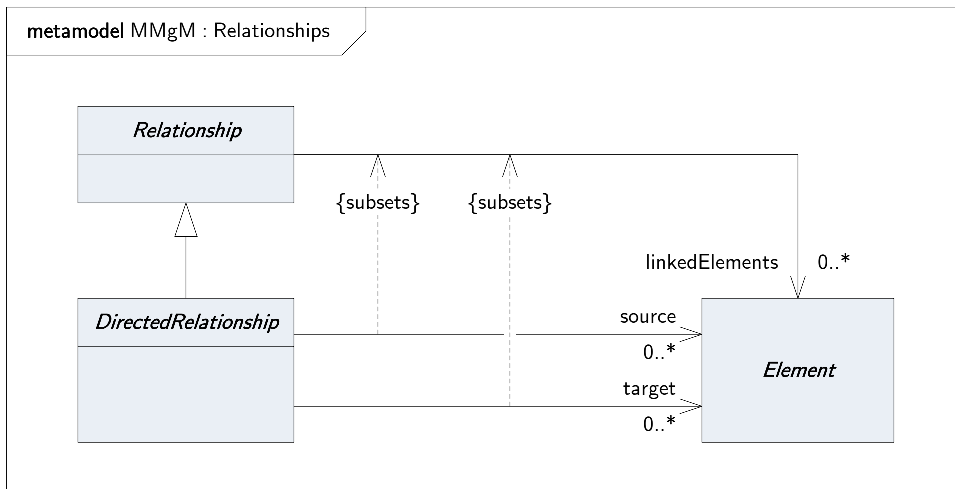


Figure 2.9: *Megamodel metamodel : Relationships.*
This figure illustrates the metaclasses representing relationships.

is different to the Megamodel metaclass defined in *AM3Core* and they are not related by means of any relationship. In *GlobalModelManagement*, MegaModel is a particular case of TerminalModel and represents megamodel artifacts stored in the model repository or reachable in the modeling environment. This metaclass represents the same concept as the metaclass Megamodel defined in our metamodel. However, in *GlobalModelManagement* this class has an association to a collection of Element, representing the fact that these *elements* are contained in the megamodel. This association is not documented in [Mod09], but it is actually included in [Mod08b], it appears in the AM3 Flyer-Poster [Atl09] and it is present in current publications related to the GMM metamodel such as [VJBB09] and [JVB+10]. Having the association from MegaModel to Element implies that for each megamodel MgM_i contained in a megamodel MgM, a link from MgM_i to the elements in its collection of elements must also be added to the megamodel MgM. In terms of our examples in Figures 2.7 and 2.8, this would imply that all elements of MgM1 must also be present in MgM2 and that there must be a link to these elements from the element representing MgM1. We claim that this association must not be part of the metamodel of the megamodel, and hence, we do not include it in our definition. To have such an association requires the duplication of information (elements in a megamodel and links in a container megamodel) and is detrimental to the modularization of modeling artifacts and their intent. Such an association is useful in the case that all megamodels are flatten into a single megamodel representing all the modeling artifacts in the modeling environment. In this scenario, a single megamodel actually exists by itself, while all other megamodels are flatten into the definition of this root megamodel. While this might simplify tool support, this is detrimental to knowledge reuse as individual megamodels cannot be shared as they are collapsed within a global megamodel.

Relationships. The metaclass Relationship represents a semantic relationship between elements and does not represent actual entities in the repository. Each *relationship* has a collection of linked elements, which, in the general case, can be either entities or other relationships. A *relationship* does not represent any directionality among the elements being related. The special kind of relationships in which directionality is important are represented

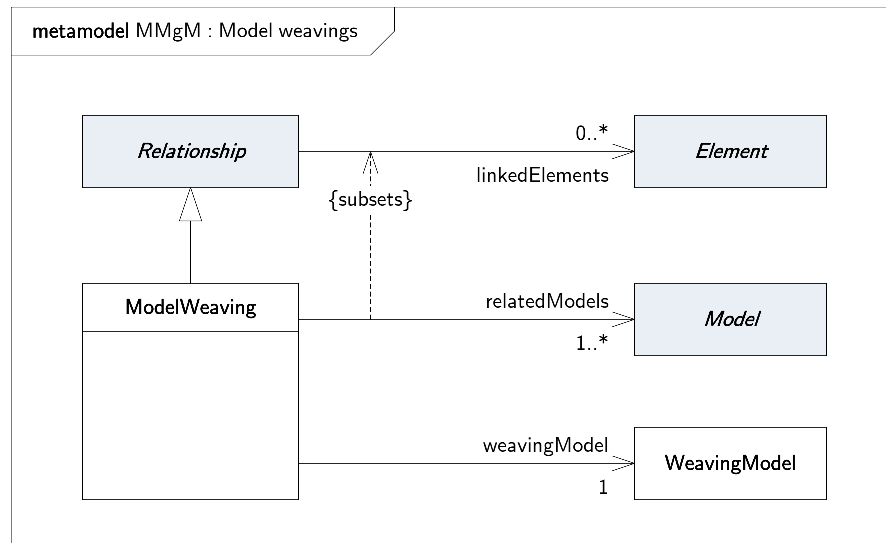
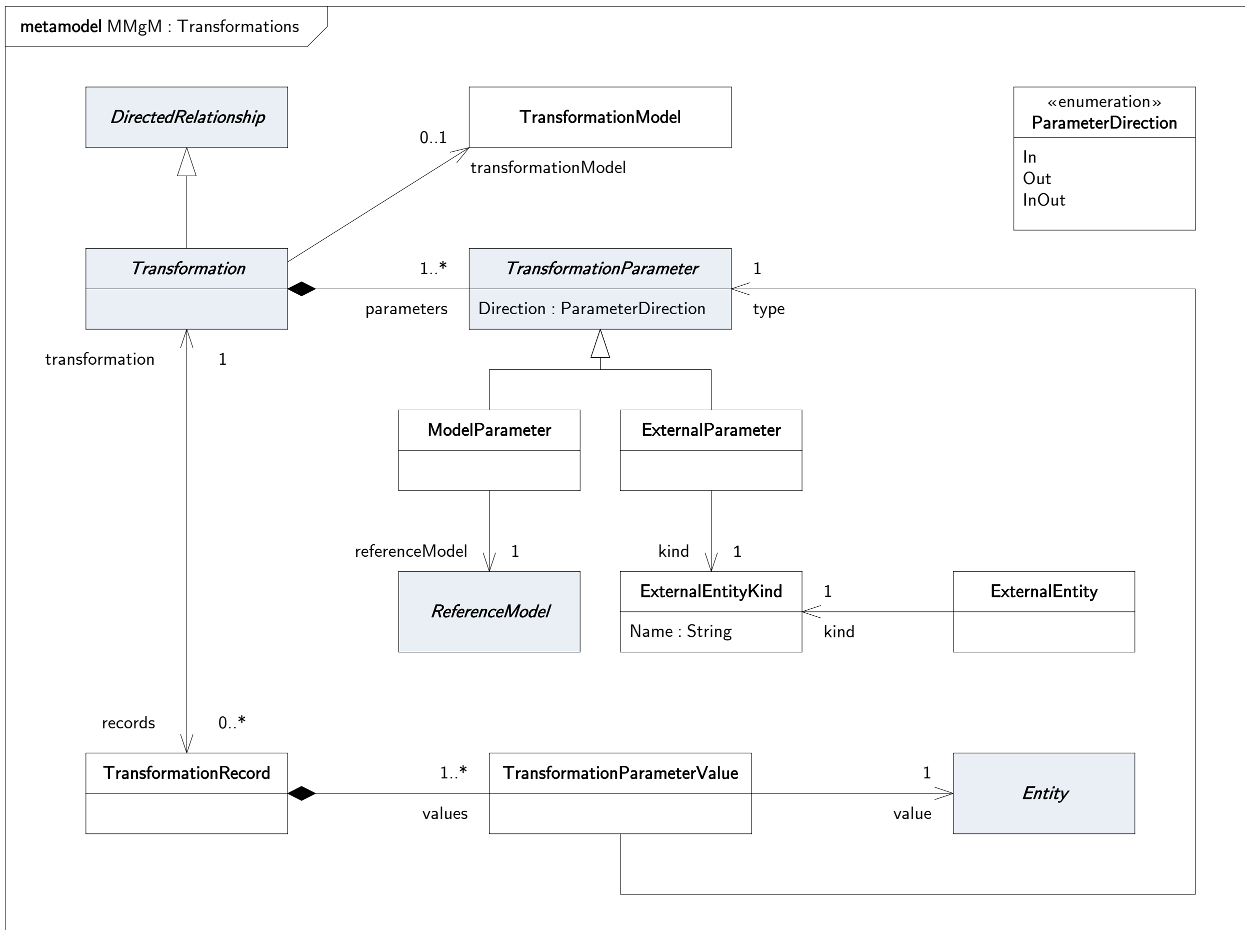


Figure 2.10: *Megamodel metamodel : Model weavings.*
This figure illustrates the metaclasses representing model weavings.

by means of DirectedRelationship. This metaclass has two separate associations, one determining the source elements and the other determining the target elements. Figure 2.9 illustrates these concepts. In the figure, the UML constraint between association called subsets, is used to indicate that in the case of a DirectedRelationship, the collection of source elements and the collection of target elements are a subset of the collection linkedElements elements. In other words, in the case of a DirectedRelationship, the collection of linkedElements is the union of the collections of source and target elements.

As we discussed in Section §2.2, a *model weavings* is a relationship between model elements from separate models and *weaving models* are representations of *model weavings*. Then, the metaclass ModelWeaving is defined as a sub-metaclass of Relationship stating that a set of models are woven. A ModelWeaving has an association to the collection of woven models, namely relatedModels, and to the single WeavingModel entity that represents (preserves) the actual links of the model weaving. The set of woven models can be a singleton in order to cope with the usage scenario of model annotations. The *related models* are the *linked elements* of the relationship. Figure 2.10 illustrates these metaclasses.

Analogously, the metaclass Transformation is also defined as a relationship, but, in this case, it is a sub-metaclass of DirectedRelationship to allow the distinction between source and target entities. It is important to notice that this metaclass represents *transformations* between entities and not necessarily models, i.e. it represents the directed relationship between a collection of source entities and a collection of target entities. The sub-hierarchy of Transformation defines the different kinds of transformations that deserve distinction. Despite this, a Transformation may be linked to a specific TransformationModel which consists of the representation of the transformation. However, this model might not exists as the *transformation* might not be realized in the modeling technical space. For instance, a transformation that takes a document (an spreadsheet for example) and builds a model from it might not be specified by means of a *transformation model* and might be directly implemented by an external tool available in the development environment.



context TransformationRecord **inv: self**.values.type = **self**.transformation.parameters

context TransformationParameterValue **inv:**
  self.type.oclIsKindOf(ModelParameter) **implies**
    self.value.oclIsKindOf(Model) **and**
    self.type.oclAsKind(ModelParameter).referenceModel =
      self.value.oclAsKind(Model).conformsTo

context TransformationParameterValue **inv:**
  self.type.oclIsKindOf(ExternalParameter) **implies**
    self.value.oclIsKindOf(ExternalEntity) **and**
    self.type.oclAsKind(ExternalParameter).kind =
      self.value.oclAsKind(ExternalEntity).kind

context ParameterDirection **def:** IsIn : Boolean =
  self = ParameterDirection::In **or self** = ParameterDirection::InOut

context ParameterDirection **def:** IsOut : Boolean =
  self= ParameterDirection::Out **or self** = ParameterDirection::InOut

Figure 2.11: *Megamodel metamodel : Transformations.*
This figure illustrates the metaclasses representing transformations.

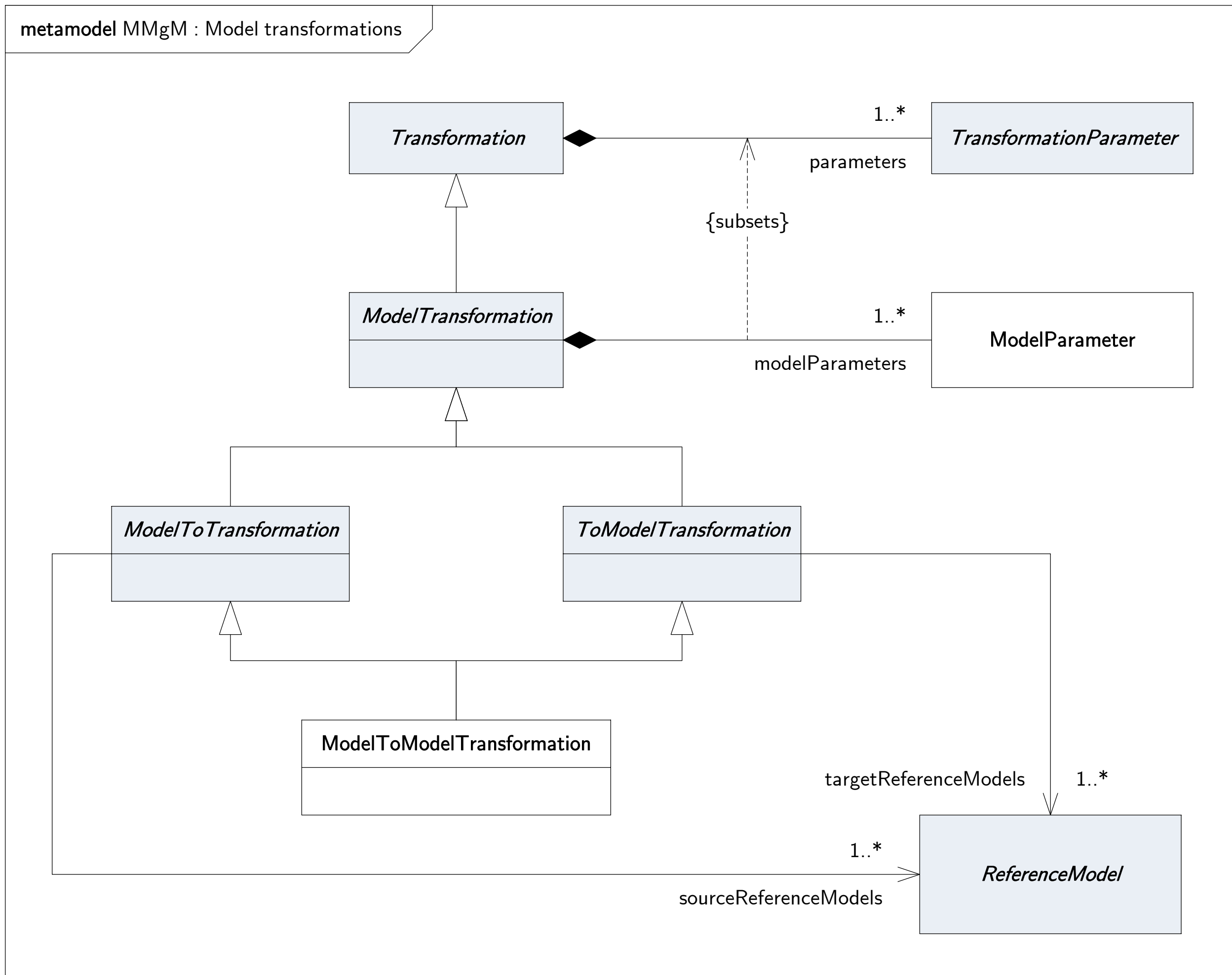
Transformations has a collection of parameters representing the kind of entities they can work on. These parameters are represented by TransformationParameter. They do not directly relate the actual entities participating in the transformation, the parameters indicate the kinds of these entities. Figure 2.11 illustrates these metaclasses. When the related entities are *models*, their *reference model* is considered as the kind for the model. Hence, the metaclass ModelParameter represents a parameter that takes models conforming to a particular ReferenceModel. In the case of *external entities*, we have not introduced yet a means to directly represent its kind, and we need it in order to specify ExternalParameters. For instance, suppose we have a model RPM that describes the runtime platform of the system and that conforms to RPMM, a model transformation T that takes models conforming to RPMM metamodel and that generates a report listing all the computational nodes (workstations, servers, devices) in the runtime platform. Let RPReportKind be this kind of report listing the computational nodes. Then, we would be able to specify that the transformation T has RPMM as the source reference model and RPReportKind as the kind of target. Then, the actual execution of the transformation T on the model RPM yields an actual report of kind RPReportKind. We introduce the ExternalEntityKind metaclass as the means for representing kinds of *external entities*. The *GlobalModelManagement* metamodel does not define this metaclass, instead, it uses a Uniform Type Identifier (UTI) as a type-reference mechanism within the metamodel. We prefer not to use a technology-related mechanism and hence we define a specific metaclass for the purpose.

The fact that a *transformation* has been executed on a collection of source entities, rendering a collection of target entities, is represented by means of a TransformationRecord. This record determines which transformation was executed, and which were the actual entities used as parameters for the transformation. The restrictions that used entities satisfy what is expected by the transformation is stated by means of invariants.

The particular kind of *transformations* that involves models are represented by the metaclass ModelTransformation. Thus, a model transformation has models as parameters. Those model transformations in which the source entities are models are represented by ModelToTransformation. An association to the source reference models is a shortcut for the model parameters with In direction. Analogously, those model transformations in which the target entities are models are represented by ToModelTransformation. In this case, the target reference models are those reached by the models parameters with Out direction. The concrete metaclass ModelToModelTransformation represents actual transformations that take models and produce models. These metaclasses are illustrated in Figure 2.12.

Symmetrically, the particular kind of *transformations* that involves external entities are represented by the metaclass ExternalTransformation. The ExternalToTransformation and ToExternalTransformation abstract metaclasses represent transformations sourced by and targeted to external entities, respectively. The ExternalToExternalTransformation concrete metaclass represents actual transformations taking external entities and producing external entities. Figure 2.13 illustrates these metaclasses.

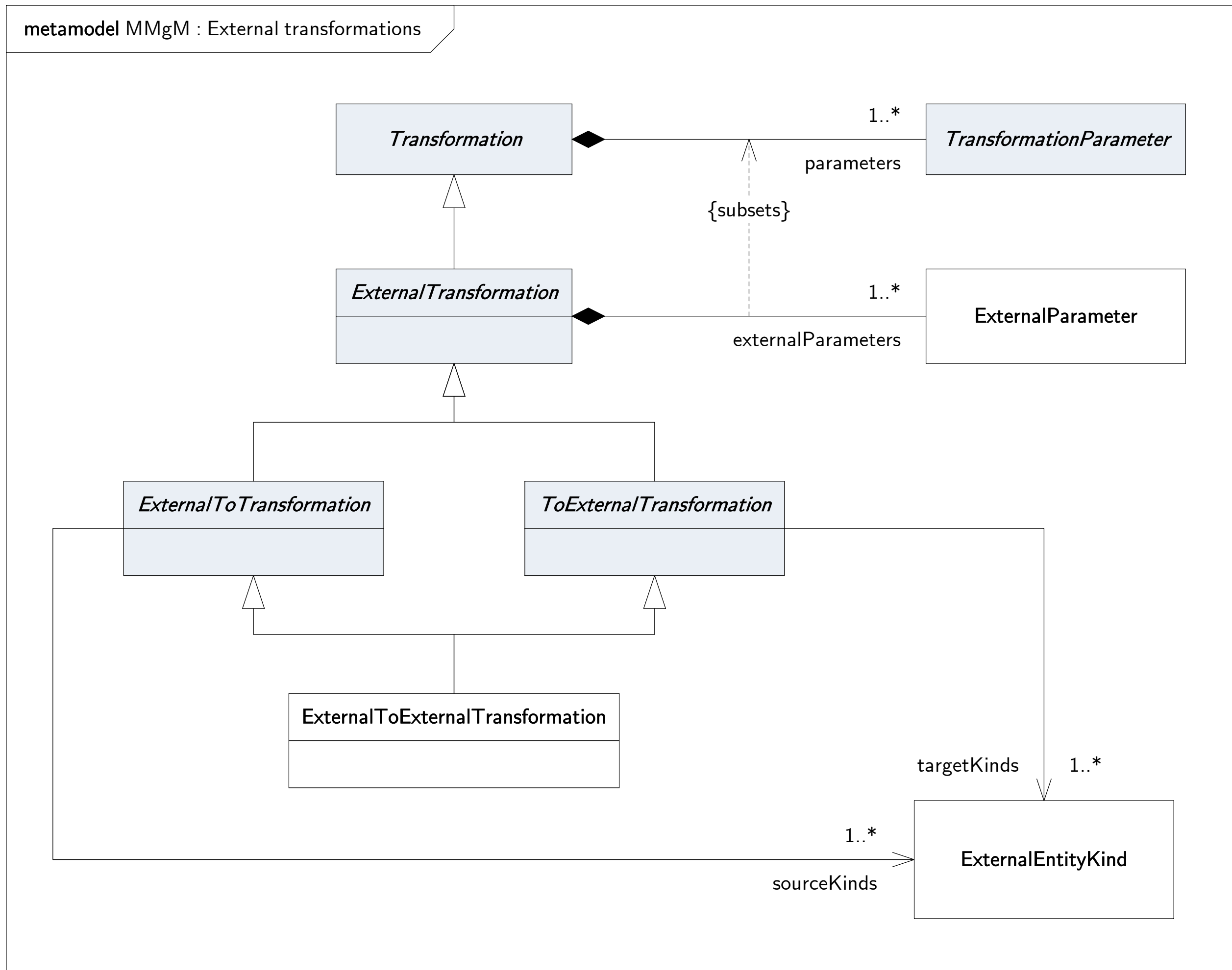
Additionally, mixed transformations can be conceived. *Injectors* are the specific kind of transformations that take external entities and produce models. Symmetrically, *extractors* are transformations that take models and produce external entities. The concrete metaclasses



context ModelToTransformation **inv: self**.sourceReferenceModels =
  self.modelParameters
    →select(p | p.Direction.IsIn)
    →collect(p | p.referenceModel)

context ToModelTransformation **inv: self**.targetReferenceModels =
  self.modelParameters
    →select(p | p.Direction.IsOut)
    →collect(p | p.referenceModel)

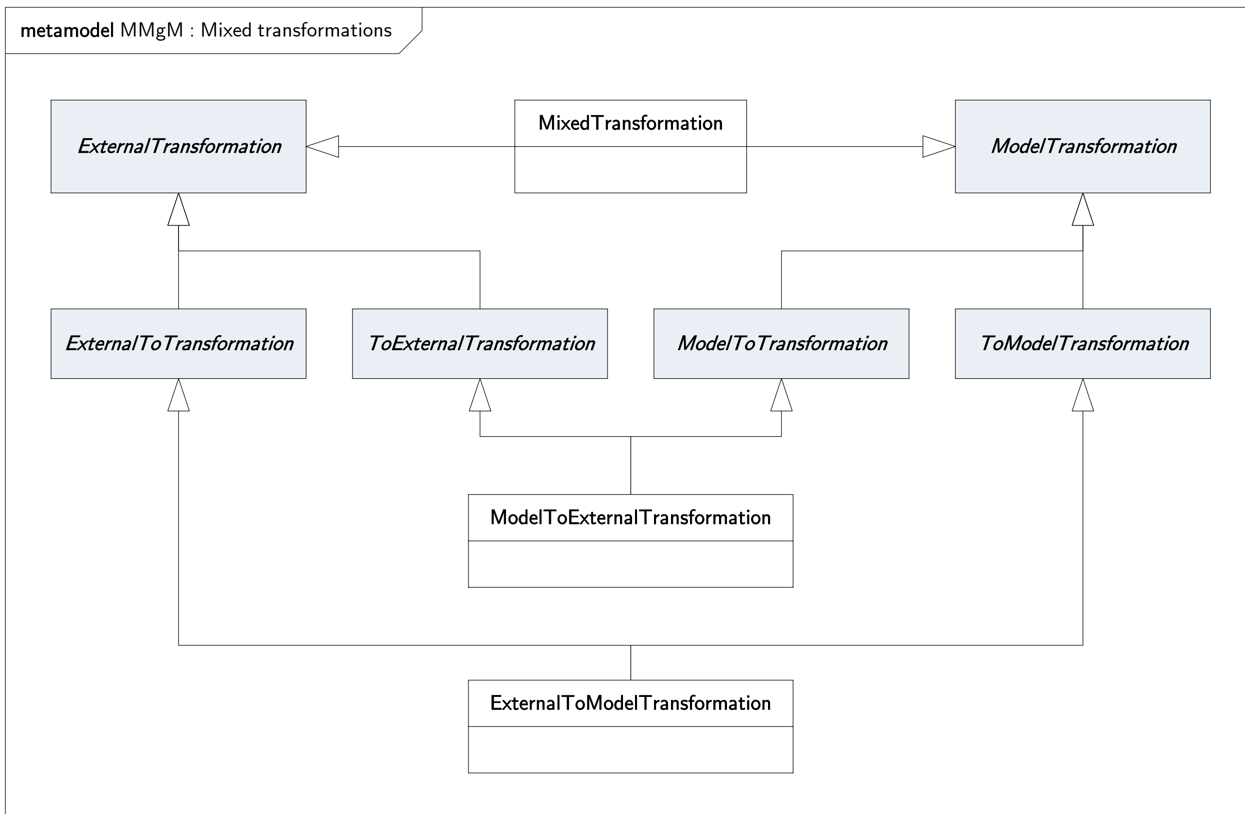
Figure 2.12: *Megamodel metamodel : Model transformations.*
This figure illustrates the metaclasses representing model transformations.



context ExternalToTransformation **inv: self**.sourceKinds =
self.externalParameters
→select(p | p.Direction.IsIn)
→collect(p | p.kind)

context ToExternalTransformation **inv: self**.targetKinds =
self.externalParameters
→select(p | p.Direction.IsOut)
→collect(p | p.kind)

Figure 2.13: *Megamodel metamodel : External transformations.*
This figure illustrates the metaclasses representing external transformations.




```
context MixedTransformation inv:
  self.parameters→exists(p1, p2 |
    p1 <> p2 and p1.Direction = p2.Direction and p1.oclType() <> p2.oclType())
```

Figure 2.14: *Megamodel metamodel : Mixed transformations.*
This figure illustrates the metaclasses representing mixed transformations.

ExternalToModelTransformation and ModelToExternalTransformation represent injectors and extractors, respectively. There is another kind of mixed transformation, one in which the sources entities and/or the target entities are mixed. We introduce the metaclass *MixedTransformation* to represent them. This concept is not directly covered in *GlobalModelManagement* metamodel. Figure 2.14 depicts mixed transformation metaclasses.

2.3.2 Realization & Tool Support

The AtlanMod team [Atl14a] has been intensively working on theoretical and practical aspects of Model Driven Engineering, developing tool support targeting the Eclipse Integrated Development Environment, both for modeling in the small and for modeling in the large. Their AtlanMod MegaModel Management (AM3) tool [Mod09] implements the Global Model Management (GMM) approach to megamodeling. The goal of this tool is to provide practical tool support for modeling in the large, leveraging the Eclipse environment with global resource management of modeling artifacts.

The AM3 project [Ecl14a] has been moved into the Model Discovery (MoDisco) [Ecl14c] that provides an extensible framework to develop model-driven tools to support reverse engineering and software modernization. MoDisco is a sub-project of the Generative Modeling Technologies (GMT). Before being moved into MoDisco, AM3 was a sub-project of GMT. GMT is an incubation project aiming to produce tool support for model-driven software development. GMT is a sub-project of the Eclipse Modeling Project [Ecl14b] which focuses on the evolution and promotion of model-based development technologies within the Eclipse community. In this context, the AM3 tool is positioned as the GMM implementation for a well-established community of developers on the Eclipse environment. AM3 is fully open-source and thus its sources are available for download from the Eclipse website. At the time of writing this work, a binary version of the AM3 plug-ins was not directly available to download. AM3 sources are accessible from Eclipse SVN servers. Detailed instructions to obtain the tool in an operational Eclipse environment is provided in [Atl14b]. The tool is needed if the reader intends to recreate the case study of our work.

The AM3 tool provides a megamodel underlying the modeling environment in which the modeling artifacts and their relationships can be registered. This underlying megamodel can be browsed and updated by the user through the graphical user interface. The AM3 *Megamodel Navigation* provides a tree-view of the hierarchy of metaclasses of the metamodel of the megamodel, and by selecting a metaclass, all the registered elements of the selected metaclass are listed. The properties and metadata of an element can be viewed and edited by means of *editor pages*. There are two different top-most editor pages, one for Entity elements and one for Relationship elements. The editor pages are organized in *tabs*, each providing access to a subset of the properties of the element. As these editors are extensible, extensions to the tool are expected to add new tabs with viewing and editing capability for new properties. Each kind of modeling construct (metaclass in the metamodel of the megamodel), has attached a set of actions that are accessible to the user by means of a contextual menu. In addition to the view/edit action on the model element, other operational actions are available, such as opening a model browser to explore the content of a registered model or triggering the execution of a transformation. Operational actions, when executed, must update the megamodel to record the changes; for instance, when executing a model transformation on a set of source entities, apart of creating/updating the target entities, the new models must be registered in the megamodel as well as a transformation record must be added to record the execution. Discovery functionality is also available in order to automatically registered a set of modeling artifacts in the underlying megamodel. Moreover, manual creation of elements is also available. When the user registers a modeling artifact to the megamodel, the corresponding modeling elements are created and some metadata associated to the element is automatically recorded, such as its name and location, and the metamodel that a model conforms to, among others.

It is important to notice that, in the AM3 tool, there is a single megamodel embracing all the modeling artifacts and their relationships that are available in the model repository or that are reachable in the modeling environment. To define a new megamodel implies to create a new model element conforming to the corresponding metaclass and adding the links to its constituting elements. By this means, all megamodels are flatten in the megamodel underlying the modeling environment. This issue was discussed in Section §2.3.1.

The AM3 tool is structured in three set of Eclipse plug-ins [Mod09]:

- The *core* plug-ins provide the implementation of the top-level metamodel for the megamodel, namely *AM3Core*, the core functionality of the runtime environment of the tool, the Application Programming Interfaces (APIs) for extension support, and visual navigators and editors to explore megamodels.

- The *utilities* plug-ins implement basic model-repository solutions and provides extraction facilities for Apache ANT script models and Eclipse launch configuration models.

- The *extension* plug-ins consist of several extensions to the core metamodel *AM3Core*, each providing support for a particular modeling technology. The responsibility of an extension plug-in is to extend the metamodel of the megamodel including new constructs (which should be specializations of the existing constructs), the corresponding extension to the editor pages of the existing constructs, and contextual actions. In addition, an extension-specific API should be provided so as to enable extensions to the extension.

The current developed extensions for the AM3 tool mainly target those technologies also provisioned by the AtlanMod team that are available in the Eclipse environment. While these technologies are used for modeling in the small, their corresponding AM3 extension aims to leverage their usage in an environment with support for modeling in the large. The set of extensions available is illustrated in Figure 2.15. The *AM3Core* is the root module, providing a minimal metamodel and the core APIs for the extensions. *GlobalModelManagement* is actually implemented as an extensions to AM3. It extends the metamodel and facilities of the minimal environment provided by *AM3Core*, providing general-purpose technology-agnostic constructs of modeling artifacts and relationships. We use these two modules, *AM3Core* and its extension *GlobalModelManagement*, to define our metamodel for megamodels in Section §2.3.1. The other extensions are actually defined as extensions of the *GlobalModelManagement*. We briefly describe them in what follows. We refer the reader to [Mod09] for a thorough description of the extensions, and mainly their metamodels.

GMM4ATL: This extension introduces constructs that are specific for representing and executing AtlanMod Transformation Language (ATL) modeling artifacts. It defines ATLModel, further categorized as ATLModule and ATLLibrary, to represent *transformation models* implemented in ATL. The corresponding relationship class ATLTransformation is introduced as a specific case of *model to model transformation*.

GMM4AMW: This extension introduces constructs that are specific for representing AtlanMod Model Weaver (AMW) modeling artifacts. It defines AMWModel to represent *weaving models* expressed in AMW, i.e. conforming to the AMW metamodel or to an extension of it. The corresponding relationship class AMWWeaving is introduced as a specific case of *model weaving*.

GMM4ASM: This extension provides constructs that are specific of the ATL virtual machine, particularly, for compiled ATL model transformations. Also, its API provides utilities that are used by the *GMM4ATL* and *GMM4AMW* extensions.

GMM4TCS: This extension introduces constructs that are specific for representing and

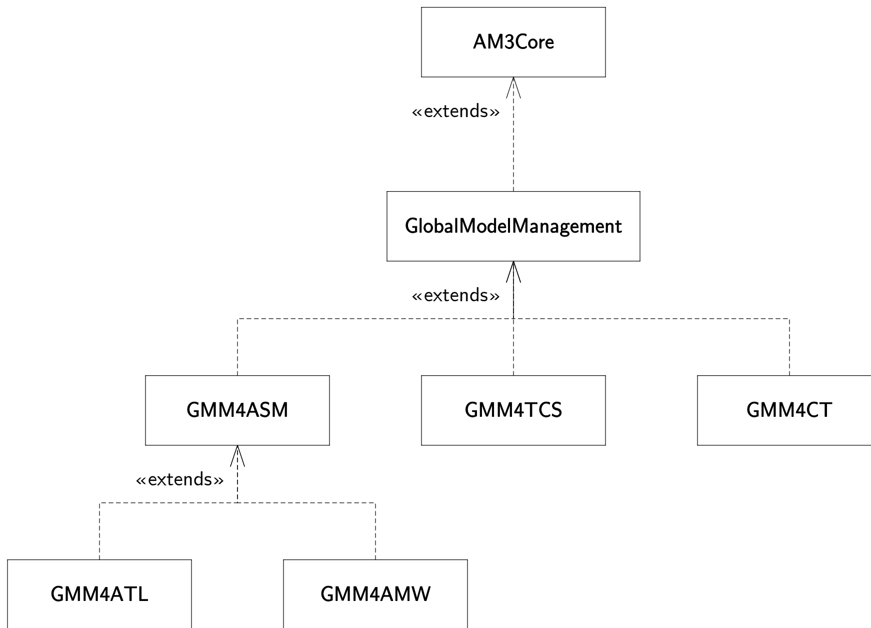


Figure 2.15: *AM3 support to the GMM approach to megamodeling.*
This figure illustrates the main modules and extensions of the AM3 tool support for the Global Model Management approach to megamodeling.

executing Textual Concrete Syntax (TCS) modeling artifacts. *GMM4TCS* defines TextualEntity to represent text files in the megamodel. As text files are non-model artifacts, TextualEntity is a sub-metaclass of Entity. In the metamodel we define in Section §2.3.1, we introduce the general metaclass ExternalEntity to refer to all non-model artifacts in the environment. Then, TextualEntity can be regarded as a specific kind of *external entity*. *GMM4TCS* also defines TCSModel as a specific kind of TransformationModel to represent the concrete models implemented in TCS[1]. TCSInjection and TCSExtraction are defined as a *external to model* and as a *model to external transformation*, respectively, that refer to a TCMModel as its *transformation model*. While TCSInjection represents the construction of a model from a text file being parsed accordingly to the TCSModel, TCSExtraction represents the creation of a text file from a model which is pretty-printed accordingly to the TCSModel.

GMM4CT: This extension introduces constructs for defining and executing composite transformations. By this means, the AM3 implementation of the GMM approach provides a mechanism for defining composite transformations in the megamodel itself, instead of using a transformation language specific for this purpose. Although this might make it simple for developers to create composite transformations as no additional tool is required neither to express them nor to execute them, this is detrimental to knowledge encapsulation, sharing and reuse, in addition to the limitations of the expressiveness of the constructs introduced by the extension; for instance, recursion is not supported and parameters are forcibly entities and hence not executable.

[1]Although TCSModel is depicted as a sub-metaclass of ModelTransformation in [Mod09], page 32, it is clearly stated in the report that TCSModel are actually concrete models. Hence, it is a sub-metaclass of TransformationModel.

As we discussed in Section §2.3.1, we use the conceptual metamodel we defined as a means for the model-based interpretation of Software Architecture Description and Decision. Then, this interpretation is technology-independent and can be used with any of the existing or still-to-be-developed extensions targeting specific technologies. However, the application of our interpretation to the case study must rely on specific technologies in order to be applied in a development environment. To this end, we actually use AM3 extensions when building the case study.

Chapter 3

Model-Based Software Architecture Description

The architecture of a software system constitutes what is essential about that system and its relation to its environment. It conforms the fundamental characterization of the system pertaining to its constituents or elements, their arrangement or interrelation, the principles of its organization or design, and the principles governing its development and evolution over its life cycle. An architecture description is the actual work product that is built and used for capturing, documenting and communicating the architecture of the system of interest. An architecture description is built by means of an architecture design method, whose activities are generally part of a broader system development or evolution effort. We work on architecture design in Chapter §4. Building the architecture description is not an optional or an after-the-fact activity of the architecture design. Quoting P. Clements et al. from [CBB+10], "even the best architecture, most suited for the job, will be essentially useless if the people who need to use it do not know what it is, cannot understand it well enough to apply it, or misunderstand it and apply it incorrectly." The goal of an architecture description is therefore threefold. It has the educational purpose of introducing new team members and external stakeholders to the system. Also, it serves as the primary means for communication among all stakeholders on the system's foundations and governing principles. Finally, it is used as the basis for analyzing system properties and stakeholder expectations fulfillment, and for the construction and evolution of the system. An architecture description is a complex work product consisting of several interrelated work products. It is organized in terms or architecture views, each of them representing the system from a particular perspective and dealing with a specific set of concerns on the system. In turn, views are structured as an aggregation of models. An architecture model is used to capture the impact of architecture decisions and acts as a container for applying architecture patterns or styles to express the fundamental structures and behaviors within views.

In the current state of the art, architecture knowledge on architecture description is large and heterogeneous. The research and practitioners community have developed a plethora of techniques and languages for describing the architecture of systems and families of systems, both general and for particular application domains. Such knowledge is available in the form of books and articles, both printed and published in the Internet, and educational

Chapter Contents

3.1 Standard on Software Architecture Description **71**

3.1.1 Architecture Descriptions 74

3.1.2 Architecture Frameworks & Languages 79

Architecture Frameworks 79

Architecture Description Languages 80

3.2 Model-Based Architecture Description **83**

3.2.1 Definition of the semantic functions 85

(i) Syntactic domain 86

(ii) Semantic domain 88

(iii) Semantic function 98

(iv) Semantic equations 101

3.2.2 Semantics of Architecture Descriptions 104

Concerns 105

Stakeholders 108

Model kinds 111

Architecture Models 119

Architecture Viewpoints 121

Architecture Views 127

Correspondences 128

Correspondence Rules 133

Architecture Descriptions 136

Architectures & Systems-of-interest 145

3.2.3 Semantics of Frameworks and Languages 146

Architecture Frameworks 147

Architecture Description Languages 149

3.2.4 Communicating Architecture Descriptions 152

The Tree-Forces Approach to Architecture Description 153

The Three-Forces Model-Based Approach 154

3.3 Contributions & Discussion **156**

courses are included in engineers curricula, facilitating the learning curve of newcomers and the practice of active architects. Architecture viewpoints capture the knowledge on ways of looking at systems and specify the conventions, with respect to a particular set of concerns, that are needed for constructing views to capture, document and communicate the architecture decisions from a given perspective. Viewpoints use architecture model kinds to define the notations and semantics for building architecture models of the views governed by

the viewpoint. Architecture frameworks is a widely used mechanism for capturing common practices on architecture description, usually by grouping an interrelated set of viewpoints. Architecture description languages is a more focused mechanism used to capture particular notations, semantics and analysis tools framed to a cohesive set of concerns. These mechanisms encapsulate knowledge on architecture description aimed to be shared and reused in multiple architecture design scenarios.

The ISO/IEC/IEEE 42010:2011 standard [ISO11] provides a conceptualization of the architecture description practice. It specifies the consensus on the requirements on architecture descriptions, frameworks and description languages, somehow ruling on how to organize, describe and communicate architecture knowledge on architecture description. However, to the best of our knowledge, the community has not reached a consensus on how to express such architecture knowledge. There is no homogeneous means agreed upon on how to represent the knowledge in order to make it reusable, shareable and mainly tool friendly. We argue that this lack of a uniform means is hindering the growth of the discipline and the pace of its adoption in industry. The mechanisms that researchers and practitioners rely on today do not encourage, promote or facilitate collaboration or reuse.

We address this problem by applying Model-Driven Engineering techniques as the homogeneous means for explicitly capturing architecture knowledge on architecture description. We define how the concepts pertaining the architecture description practice can be captured or interpreted in terms of modeling artifacts, particularly by those that can be defined and organized following the Global Model Management approach explained in Section §2.3. We use a denotational semantic approach to formally define this interpretation. Then, Model-Driven Engineering constructs conform the homogeneous medium for knowledge shareability and reuse.

This chapter is structured as follows. Section §3.1 reviews and discusses the conceptual model for architecture descriptions, architecture frameworks and architecture description languages, as defined by the ISO/IEC/IEEE 42010:2011 standard [ISO11]. Section §3.2 defines the model-based interpretation of the conceptual models in terms of the Model-Driven Engineering constructs captured by the Global Model Management approach that we introduced in Section §2.3. Section §3.3 concludes this chapter with contributions and discussion.

3.1 Standard on Software Architecture Description

In September 2000, the IEEE Standard Board approved the *IEEE Std 1471:2000, Recommended Practice for Architectural Description of Software-intensive Systems* [IEE00], developed by the IEEE Architecture Working Group under the sponsorship of the IEEE Software Engineering Standards Committee. This standard was later adopted by ISO and was published in July 2007 as *ISO/IEC 42010:2007, Systems and software engineering – Recommended practice for architectural description of software-intensive systems* [ISO07]. The contents of both standards were identical. As the result of a joint ISO and IEEE revision on these earlier standards, in November 2011 the *ISO/IEC/IEEE 42010:2011, Systems and*

software engineering – Architecture description [ISO11] was published, superseding the earlier edition. At the time of writing this thesis report, [ISO11] is the latest edition of the standard.

The goal of the original IEEE Std 1471 was to facilitate the expression and communication of architectures through the standardization of concepts and practices for architecture description [IEE00]. It was intended to reflect generally accepted trends in the software architecture community and to provide a framework for further evolution in the area. The standard established a conceptual model for architecture description and defined the requirements that had to be met by the content of such descriptions.

The conceptual model captured the key notion of reusable *architecture viewpoints* as a means to codify best practices for the creation and use of architecture *views* within an architecture description. This concept was being developed by the community since the mid-1990s [EH09]. Additionally, the concept of *library viewpoint* was introduced to represent *viewpoints* that were not directly defined in the architecture description, but that were publicly available elsewhere. This concept intended to allow practitioners to capture existing approaches and to reuse them in their architecture descriptions. For instance, the 4+1 view model of P. Kruchten [Kru95] and ISO/IEC Reference Model of Open Distributed Processing (RM-ODP) [ISO98] were available library viewpoints at the time of the standard. In the years that followed, other approaches that accommodate the notion of library viewpoint were developed, such as Siemens' four views [HNS99], the approach of J. Garland and R. Anthony [GA02] for large scale software architecture, the viewpoints and perspectives of N. Rozanski and E. Woods [RW05] for information systems, and SEI's Views & Beyond approach to documenting architecture [CGB+02].

The specified requirements set by the standard on the content of architecture descriptions of systems includes:

- identification and overview information, such as date of issue and status, issuing organization, change history, summary, scope, context, glossary and references;
- identification of the system stakeholders and their concerns considered to be relevant to the architecture;
- specification of each viewpoint that has been selected to organize the description of the architecture and the rationale for those selections;
- one or more architecture views conforming to their governing viewpoint (which must be also included in the architecture description);
- an analysis of consistency and a record of all known inconsistencies among the architecture views; and
- the rationale for the architectural concepts selected, evidence of the consideration of alternative architectural concepts and the rationale for the choices made.

The ISO/IEC/IEEE 42010:2011 standard was built upon the specification of the IEEE Std 1471. In addition to a refinement of the conceptual model of architecture descriptions, the standard introduces the definition and requirements for the concepts of architecture frameworks and architecture description languages. Although both concepts date back to before the 1990's and their influence was pervasive, their impact on the IEEE Std 1471 was

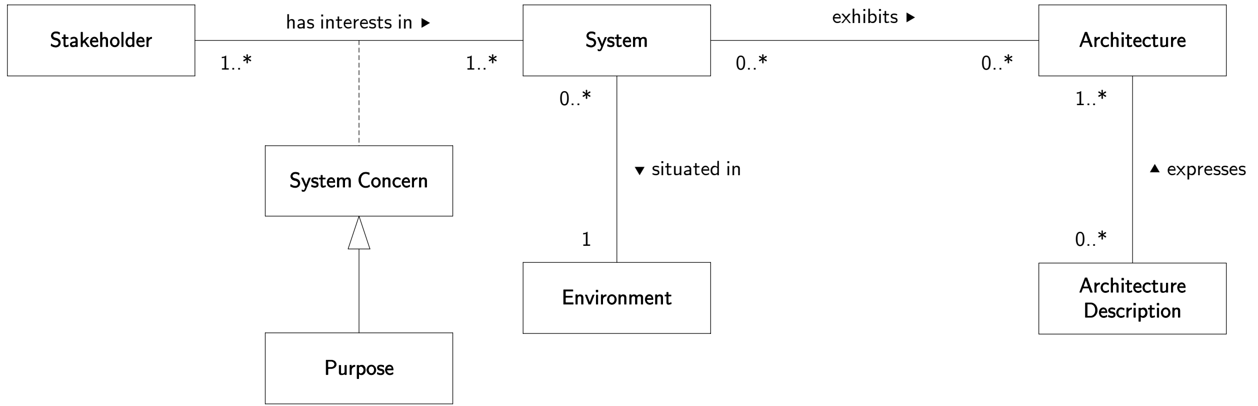


Figure 3.1: *Context of the software architecture description practice.*
The figure illustrates the main concepts pertaining to the context of the practice of software architecture description, as it is conceived by the ISO/IEC/IEEE 42010:2011 standard [ISO11].

minimal to none. The key mechanism for knowledge encapsulation and reuse in IEEE Std 1471 was that of viewpoint and viewpoint library. Architecture frameworks and architecture description languages are widely used mechanisms for capturing common practices. An architecture framework groups a set of interrelated viewpoints and correspondence rules on the participating models. An architecture description language is a more focused mechanism that captures particular notations, semantics and analysis tools framed to a cohesive set of concerns. The current edition of the standard conceptualizes these mechanisms with respects to the conceptual model of architecture description, and defines the requirements for capturing and communicating them.

The standard identifies the key concepts pertaining to the context of the practice of architecture description. Figure 3.1 illustrates this contextual model. The term *system* is used to refer to those entities whose architectures are of interest. The standard focuses on software-intensive systems in which software development and/or integration are dominant considerations, software products and service, and general man-made systems that are configured using hardware, software, data, humans, processes, procedures, facilities, materials and naturally occurring entities. The scope has been widened from the IEEE Std 1471, which only considered software-intensive systems, in order to cover the notion of *systems* as understood by other ISO/IEC standards. The standard does not provide a definition for system and, more importantly, the concepts involved in the standard do not preclude their use for describing the architecture of other kind of entities of interest.

A *system* is situated in one single *environment*, and an environment can contain multiple systems. The environment determines the totality of influences upon the system throughout its life cycle, including the interactions between the system and the environment. The *stakeholders* of a system are parties with interests in that system considered in relation to its environment. Stakeholders' interests on the system are understood as *concerns*, some of them being the *purpose* that stakeholders ascribe to a system. The *architecture* of a system constitutes what is essential about a system in relation to its environment. As there is no single characterization of what is essential or fundamental to a system, more than one architecture might be considered for the same system. The characterization could pertain

to the system constituents or elements, their arrangement or interrelation, the principles of the system's organization or design, and/or the principles governing the evolution of the system over its life cycle. The same architecture could characterize more than one system, for instance, when the architecture describes a family of systems sharing a common architecture. *Architecture descriptions* are used to express the architectures of systems of interest. Several distinct architecture descriptions can be used to express the same architecture, for instance, by employing distinct architecture frameworks. In turn, an architecture description can be used to express one or more architectures. The standard provides no argument or example on this fact. However, we can argue that provided that an architecture description is a representation of an abstract intangible asset (an architecture), the description is the representation of all architectures that can be understood or interpreted from the description itself. An architecture description is actually the description of very similar architectures that vary in the details that are subject to interpretation or that are not explicitly stated in the description. The more precise, unambiguous and detailed the architecture description, the smaller the number of architectures that can be understood from it.

In this context, the ISO/IEC/IEEE 42010:2011 standard defines a conceptual model for architecture description, which in turn can be regarded as the conformance requirements for valid architecture descriptions. In addition, the standard defines architecture frameworks and architecture description languages in the light of the conceptual model. In this section, we overview the conceptual models for architecture descriptions, frameworks and description languages. A thorough analysis is developed in Section §3.2 where we define the interpretation of these concepts in terms of Model-Driven Engineering constructs. In this chapter, we intentionally omit the concepts *architecture rationale* and *architecture decision* as we study them in Chapter §4.

3.1.1 Architecture Descriptions

An *architecture description* is an actual work product of a system development project or an architecture design activity. It conforms the representation of an architecture, which is actually an abstract intangible conception of the essential characterization of a system of interest. The ISO/IEC/IEEE 42010:2011 standard specifies requirements on architecture descriptions, not on architectures, systems or their environments. It does not assume any architecture model, notation or technique to produce architecture descriptions and it is intended to be usable for a range of approaches including document-centric, model-based and repository-based techniques.

The standard uses a conceptual model to describe all the concepts pertaining the practice of architecture description. Although not explicitly stated in the standard, its FAQs page [ISO13] indicates that this conceptual model might be regarded as a metamodel establishing the key concepts and terms for talking about architectures and architecture descriptions. We argue that this is an inaccurate assertion that can be misleading. The conceptual model is not a metamodel, it is a *domain model* of the practice as it captures the most important types of entities or concepts, and their relationships, in a particular domain. We discuss this issue further later in Section §3.2. Figure 3.2 illustrates the concepts and their relationships. In the figure, we intentionally changed the layout (while preserving the meaning)

Figure 3.2: *Conceptual model of architecture descriptions.*

The figure illustrates the conceptual model of the ArchitectureDescription concept as conceived by the ISO/IEC/IEEE 42010:2011 standard [ISO11]. We intentionally changed the layout of the concepts, with respect to the corresponding diagram in the standard, in order to separate real-world concepts at the left column from representation concepts at the center and right columns.

from that used in the standard and most citing publications. By this means, we can clearly distinguish in the figure three columns of concepts. While the left column refers to real-world concepts, the center and right columns refer to concepts representing the constructs of architecture descriptions. We use this separation of the concepts in Section §3.2 to argue why the conceptual model cannot be regarded as a metamodel for architecture descriptions.

The conceptual model in Figure 3.2 is significantly different from the contextual model in Figure 3.1. Apart from the decomposition of the concept ArchitectureDescription, they differ in (a) the architecture description describes a single Architecture of a single SystemOfInterest, (b) the Environment concept is missing, and (c) Concerns cannot be not further characterized as Purpose.

As stated in [ISO11, Section 4.2.2] regarding (a), the conceptual model includes "the concepts pertaining the practice of architecture description when applying the standard to produce one architecture description expressing one architecture for one system of interest." Also, "the term *system-of-interest* is used in this context to refer to the [single] system whose

architecture is under consideration in the preparation of an architecture description." We claim that this limitation in the conceptual model is too restrictive and not really necessary. The standard might be applied to build a family or line of products, and, in these cases, the architecture description uses variability constructs to describe different architectures of different systems of the family or product line. Regardless of this conceptual restriction imposed on the conceptual model, the practical application of the standard to families or lines of products is possible anyway. This restriction does not affect our model-based interpretation either.

Regarding the removal of the concept Environment (b), the standard states that "the environment of a system is bounded by and understood through the identification and analysis of the system's stakeholders and their concerns." The *stakeholders* of a system have *concerns* with respect to the system-or-interest considered in relation to its environment, and the same concern could be held by one or more stakeholders. Stakeholders are not only those to which the system is intended to, stakeholders are all those participating in the development project, i.e. all those that hold any concern on the system. Concerns appear throughout the life cycle of the system, not only at the beginning of the project, and could manifest as stakeholders' needs, goals, expectations, responsibilities, requirements, design constraints, assumptions, dependencies, quality attributes, pre-existing architecture or design decisions and risks [ISO11]. Hence, the Environment is fully characterized by the definition of the complete set of concerns pertaining the system, and it is not relevant to preserve it as a separate concept.

The removal of the concept Purpose (c) is not actually commented on the standard. Stakeholders ascribe various purposes of a system, i.e. the ultimate intention the stakeholders have for the system. Being Purpose a special kind of concern, it somehow imposes a classification criterion for concerns; for instance, those that are stakeholders' purposes and those that are not. Concerns can be classified and prioritized in several ways, and how to catalog concerns actually depends on the practitioner community or organization, the skills of the development team, the clarity on the concerns, among others. It is actually an architecture decision to choose how to catalog concerns. Then, it is fair to omit this concept in the conceptual model, leaving the classification details to the representation of concerns in the architecture description.

According to the conceptual model in Figure 3.2, an *architecture description* expresses the *architecture* exhibited by the *system of interest* identified. As we mentioned before, we can actually state that the *architecture description* represents the *architecture* of a system, family or line of systems that exhibit the architecture. Provided that an architecture description cannot be a complete representation including all conceivable aspects of the architecture, in strict rigor, the architecture description actually represents all the architectures that can be understood or interpreted from the description. The *architecture description* identifies the system stakeholders having architecturally significant concerns. The identified stakeholders may include users, operators, acquires, owners, suppliers, developers, maintainers, among others. Significant concerns are those that may impact and drive architecture design.

An *architecture description* is organized in terms of *architecture viewpoints* and *architecture views*. This is denoted in the model by means of the aggregate associations. An

architecture viewpoint is a way of looking at systems. It specifies the conventions with respect to a set of concerns, that are needed for constructing certain kind of *architecture view*. The conventions take the form of types of models, notations, and languages. Viewpoints are complex work products that can be applied to many systems as their definitions do not refer to any particular system, including the *system of interest*. This reusable nature of architecture viewpoints highlights their utility as a mechanism for capturing architecture knowledge on architecture descriptions. The standard provides a guide for documenting architecture viewpoints in [ISO11, Annex B].

An *architecture view* is the result of the application of an *architecture viewpoint* to a system. Each architecture view in an architecture description represents the whole system of interest from the perspective of the system's concerns framed by its governing viewpoint. A view conforms to its viewpoint in the sense that it uses the viewpoint conventions to describe the architecture. An *architecture view* is an aggregation of *architecture models*. By means of its set of architecture models, a view addresses all of the corresponding concerns. An architecture model captures the impact of architecture decisions and acts as a container for applying architecture patterns or styles to express the fundamental structures and behaviors within views. Each architecture model adheres to a governing *model kind*. A model kind specifies the conventions for building architecture models. An *architecture viewpoint* uses model kinds to define accepted notations and semantics for building architecture models in the corresponding architecture view.

Then, the standard defines three levels of representation of the architecture. First, the architecture description is a complex and composed work product that represents the architecture of a system of interest. Second, each architecture view is a complex and composed work product that represents the architecture of the system with respect to the specific set of concerns framed by its governing architecture viewpoint. And third, each architecture model is a work product that represents the aspects of the architecture of the system that can be expressed by the conventions and purpose of its governing model kind.

The notation used to depict the conceptual model of the standard is based on ISO/IEC 19501:2005 which is actually the Unified Modeling Language (UML) in its version 1.4.2. Then, it is important to remark that the standard conceives an *architecture description* as an aggregation of *architecture viewpoints* and *architecture views*. The aggregation is denoted by the white diamond in Figure 3.2, and it must be distinguished from the notion of composition denoted by a black diamond. By aggregation is must be understood that the composing elements (i.e. architecture viewpoints and views) can compose any number of composed element (i.e. architecture descriptions). In the case of architecture viewpoints, the aggregation reifies their essential characteristic: to capture and encapsulate reusable knowledge on how to describe the architecture of a system from a particular perspective to address a specific set of concerns. The case of architecture views as a reusable assets is not actually discussed in the standard. However, we can argue that a particular architecture view can actually describe two similar but different architectures, provided that any of the other architecture views or architecture models actually differ. The standard limits an architecture description to describe a single architecture. When describing an architecture that is similar to a second architecture already described, it can be expected that some of the architecture views in the second architecture description also apply as in the architecture being described. In

turn, architecture viewpoints and views are conceived as an aggregation of *model kinds* and *architecture models*, respectively. Following the same line of reasoning as before, model kinds capture and encapsulate reusable knowledge and particular architecture models can be used to describe certain aspects of the architecture of more than one system. In the case of architecture viewpoints and model kinds, their encapsulation and reusable nature is fundamental in order to conceptualize the notion of architecture framework and architecture description language, as we discuss later. In the case of architecture views and architecture models, the standard is actually promoting reuse. Particularly for architecture models, the standard states that using the same architecture model in more than one view allows an architecture description to frame distinct but related concerns without redundancy or repetition of the same information in multiple views, reducing also the possibilities for inconsistencies. This is a very important fact that can be overlooked from the conceptual model (see Figure 3.2) and unfortunately is only mentioned as a side note in the standard. As a consequence of this discussion, we derive that architecture descriptions, architecture viewpoints, architecture views, model kinds and architecture models, are complex work products that should be encapsulated in order to facilitate their reusability in different contexts. This has a direct impact in our formalization as the semantics must reflect this encapsulation and reuse capability.

Finally, an architecture description consists also of an aggregation of correspondences and correspondence rules. Provided that the architecture description consists of several views organized in several architecture models also organized in several elements, it is to be expected that certain relations of interest need to be captured and/or enforced on them. As they represents different aspects of the same actual system, they are probably interrelated and, such interrelations are also part of the architecture description. Examples of such relationships are composition, refinement, consistency, traceability, dependency, constraint and obligation. A *correspondence* represents a relation between *architecture description elements* (ADEs). A *correspondence rule* represents a rule or assertion that must hold on the interrelated ADEs. Correspondences are generally captured by means of a table identifying the ADEs that are interrelated. Correspondence rules are generally captured as means of assertions that are expected to hold on the architecture description. Figure 3.3 illustrates the conceptual model for correspondences and correspondence rules. The figure considers the conceptual model as defined in the standard, and extends it by including an association between CorrespondenceRule and ArchitectureDescriptionElement. This association is not actually part of the standard, by it is included in the summary of the conceptual model available at the standard's resources site [ISO14]. This association is crucial if we want the architecture description to include a correspondence rule (assertion) on a correspondence (relation) between architecture description elements that need not to or cannot be defined by extension, i.e. a relation in which there is no need to record all the participating tuples. These kinds of implicit correspondences are actually based on matching properties of the participating elements, such as their name.

From the domain modeling perspective, we argue that the concept of *architecture description element* is not properly defined in the standard. An ADE is defined as any construct in the architecture description, including all the constructs in the conceptual models (such as concerns, stakeholders, viewpoints, views, model kinds, architecture models, architecture description themselves) and those introduced by the specifications of the viewpoints and model kinds in use in the architecture description (i.e. any element of any kind used within viewpoints, model kinds, views and architecture models). Although the intuition is easy to grasp,

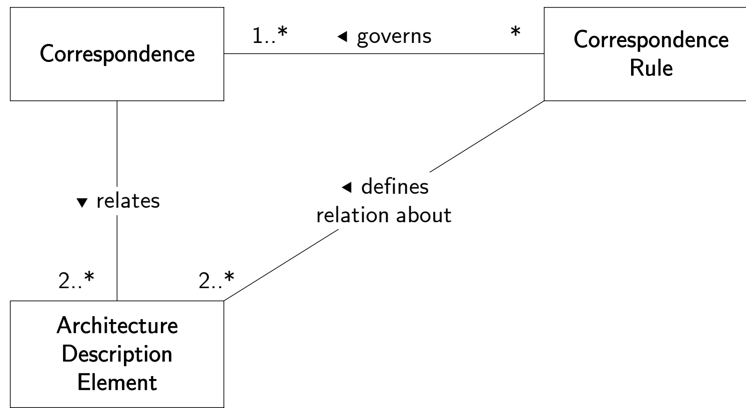


Figure 3.3: *Conceptual model of correspondences.*
The figure illustrates the conceptual model of the Correspondence and CorrespondenceRule concepts among architecture description elements as conceived by the ISO/IEC/IEEE 42010:2011 standard [ISO11]. The figure combines the model in the standard with the corresponding model in the FAQ page [ISO13] of the standard.

to include this concept in the conceptual models is not straightforward, if even possible. As there is no unifying concept in the conceptual model that can be used for this purpose, the standard introduced ADE. We might have expected a super-class of which all other concepts are specializations. However, this super-class does not address the issue of elements within the concepts (i.e. within views, models, etc.). The standard does not provide any modeling approach to represent the internal organization of views and models and hence, it cannot formally include the notion in the conceptual models. This poses a high level of complexity in the conceptualization that is propagated to our model-based interpretation and that we need to deal with.

3.1.2 Architecture Frameworks & Languages

Architecture frameworks and architecture description languages are widely used mechanisms for capturing and communicating common practices on architecture description. These mechanisms enable the encapsulation, sharing and reuse of architecture knowledge on notations, languages and construction and analysis tools for architecture descriptions. The ISO/IEC/IEEE 42010:2011 standard [ISO11] provides a conceptualization of these mechanisms with respect to the concepts pertaining the conceptual model for architecture description. We review both mechanisms in what follows.

Architecture Frameworks

The concept of architecture framework dates back to at least the 1970's, being J. Zachman's Enterprise Architecture Framework [Zac87] often cited as inspiration for current architecture frameworks. The impact of architecture frameworks on the IEEE 1471 edition of the standard was minimal, even though the pervasive influence they had on the architecture description practice at that time [EH09]. The standard emphasizes the concept of viewpoint library in

order to capture these existing approaches, reflecting the current practice in the mid-1990's to define reusable viewpoints. The sole mention of architecture frameworks in the standard was done in a single paragraph in Annex B, in which organizations were invited to produce an architecture framework for a particular domain by specifying a set of viewpoints and making it normative for architecture descriptions on that domain. The expectations of the standard was to promote that existing architecture frameworks were aligned to the standard in this manner, being the targeted frameworks the ISO/IEC Reference Model of Open Distributed Processing (RM-ODP) [ISO98], J. Zachman's Enterprise Architecture Framework [Zac87] and the first edition of L. Bass et al.approach to software architecture [BCK98].

In the years that followed, the architecture practice evolved in two main directions [EH09]. On the one hand, the enterprise architecture line of work evolved by building on J. Zachman's framework, resulting in the recent frameworks as the US Department of Defense Architecture Framework (DODAF) [DoD07, DoD09], UK Ministry of Defence Architecture Framework (MODAF) [MoD08a] and The Open Group Architectural Framework (TOGAF) [OG11]. On the other hand, the research and practitioner community developed and captured many viewpoints and viewpoint sets, such as P. Kruchten's 4+1 view model [Kru95], Siemens' four views [HNS99], the approach of J. Garland and R. Anthony [GA02] for large scale software architecture, the viewpoints and perspectives of N. Rozanski and E. Woods [RW05] for information systems, and SEI's Views & Beyond approach to documenting architecture [CGB+02].

The ISO/IEC/IEEE 42010:2011 standard [ISO11] captures the concept of *architecture framework* by providing a definition, establishing its relationships to the others concepts in the conceptual model of architecture description, and stating the requirements for capturing, documenting and sharing existing and future frameworks. For the standard, an architecture framework is the conventions and common practices for architecture description established within a specific domain or stakeholder community, including practices for creating, interpreting, analyzing and using architecture descriptions. A framework can be understood as a prefabricated structure that can be used to organize an architecture into a set of interrelated complementary views. An architecture framework identifies a set of *stakeholders*, a set of their *concerns*, a set of *architecture viewpoints* framing these concerns, and a set of *correspondence rules* that can be enforced on architecture views governed by those architecture viewpoints. We reviewed these related concepts in Section §3.1.1. Figure 3.4 illustrates the conceptual model for *architecture frameworks*.

Architecture Description Languages

Architecture description languages (ADLs) emerged in the 1990s as the result of effort of the research community to determine how to best capture software architectures [TMD09]. The debate on what constitutes a software architecture naturally yields to the proliferation of ADLs. During that period, characterizations and classifications of ADLs were proposed [KC95, MT00], and a wide variety of ADLs were surveyed in [Cle96, MT00]. Examples of these early ADLs are Darwin [MDEK95], Rapide [LKA+95], and Wright [AG97]. The underlying common denominator of early ADLs was their explicit support for modeling components, connectors, interfaces and configurations. Current techniques on architecture description go far beyond these constructs, as we reviewed in Section §3.1.1 when intro-

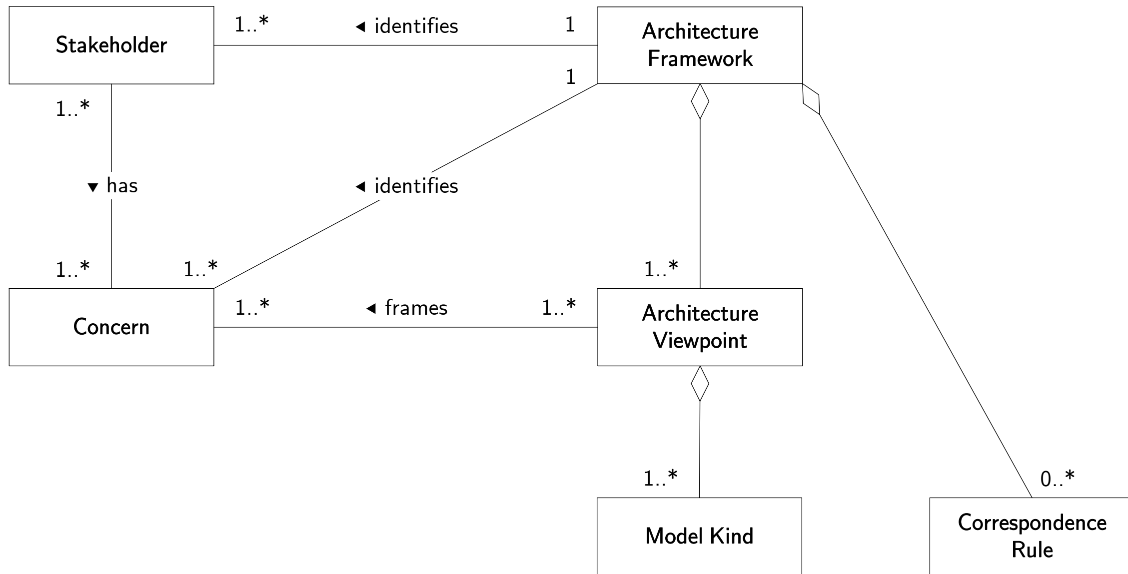


Figure 3.4: *Conceptual model of architecture frameworks.*

The figure illustrates the conceptual model of the ArchitectureFramework concept as conceived by the ISO/IEC/IEEE 42010:2011 standard [ISO11].

ducing viewpoints and model kinds. Those ADLs tended to be semantically precise but lacked breadth and flexibility [TMD09]. With the emergence of the Unified Modeling Language (UML) [OMG97], there was a significant debate in the community about whether it should be considered an ADL or not. While some authors positioned UML as the one-fits-all ADL [SC06], other authors wondered to what extent it can be considered an ADL by itself [GCK02]. For R. Taylor et al. in [TMD09], any language used to capture principal design decisions is effectively an architecture description language, including UML. For us, an ADL is a mechanism for capturing architecture knowledge on architecture descriptions, aimed to define notations, semantics and tool support for languages that are used to build *architecture models*, governed by *model kinds*. Any language intended in that direction should be considered an ADL.

The first generation of ADLs were suitable for describing a wide variety of software system architectures, in many domains and guided by different architecture styles. In the year that followed, domain- and style-specific ADLs emerged. The advantages of this kind of ADLs are that they are targeted to a particular set of stakeholders rather than to developers, they use domain-specific constructs avoiding general ones and reducing verbosity, and the assumptions on the domain can be encoded in the language semantics [TMD09]. Examples of these ADLs are Koala [OLKM00] and AADL [FGH06], both of them in the domain of embedded and real-time systems.

In addition to general-purpose and domain-specific ADLs, there is a third category of ADLs whose main characteristic is to be extensible. The first ADL in this category was Acme [GMW97] in which extensibility is achieved by adding property decorations to any construct available for architecture description. These new properties are used by practitioner-developed tools to enhance analysis and visualization capabilities. The Architecture Description Markup Language (ADML) [Spe00] is an XML-based extensible ADL derived from

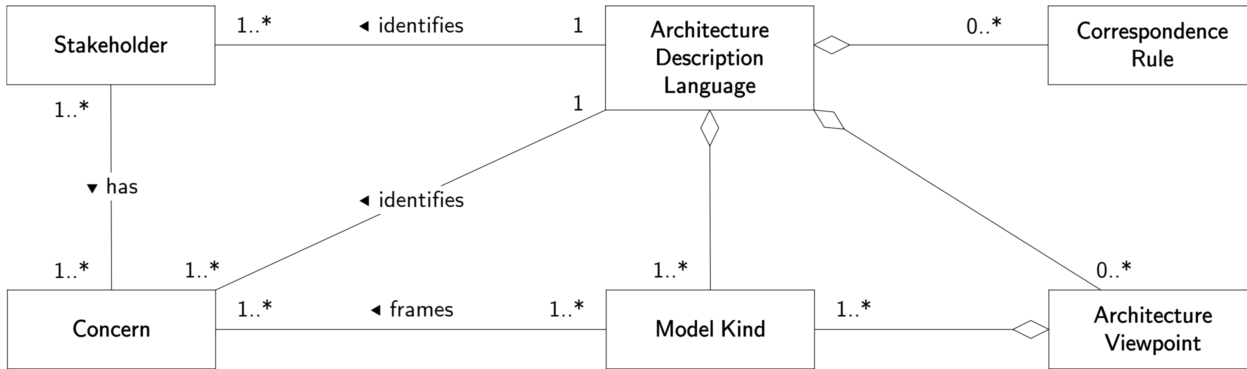


Figure 3.5: *Conceptual model of architecture description languages.*
The figure illustrates the conceptual model of the ArchitectureDescriptionLanguage concept as conceived by the ISO/IEC/IEEE 42010:2011 standard [ISO11].

Acme. Using XML allows practitioners to benefit from the large set of tools available for parsing, manipulating and visualizing XML. xADL [DHT05] is the most recent extensible ADL. It is also based on XML but, in contrast to ADML, it uses XML extensibility mechanisms for its extensions. Also, xADL is not restrictive with respect to the constructs that can be used for architecture description, while ADML is restricted to those defined in Acme.

A non-extensible ADL consists on a rigorously-defined language with companion tools for visualizing, editing, checking and analyzing architecture descriptions based on that language. An extensible ADL, however, provides a framework in which a family of general-purpose and/or domain-specific architecture description languages can be built. Quoting R. Taylor et al. in [TMD09], "effectively, extensible ADLs can be seen as domain-specific ADL factories." This poses additional steps in architecture design in which practitioners have to develop their own ADL extensions when no existing one meets their description needs. This fact has an impact in our model-based interpretation, as we discuss later in Section §3.2.3 and Section §4.2.3. Moreover, the goal and benefits of extensible ADLs is the possibility of having an homogeneous means for capturing architecture knowledge on architecture descriptions, for which XML is the current choice. This fact has a significant intersection with the goal of our work as we discuss at the end of this chapter.

Although ADLs where missing in previous editions of the standard on architecture description practice, the ISO/IEC/IEEE 42010:2011 standard [ISO11] captures this concept by providing a definition, establishing its relationship to the other concepts in the conceptual model of architecture description, and stating the requirements for capturing, documenting and sharing existing and future ADLs. For the standard, an ADL is any form of expression for use in architecture descriptions. An ADL provides one or more *model kinds* as a means to frame some *concerns* for its audience of *stakeholders*. An ADL can have a narrow focus and provide a single model kind, or it can have a wide focus and provide several model kinds, optionally organized into *architecture viewpoints*. An ADL can also define a set of *correspondence rules* that can be enforced in the architecture models resulting from using the model kinds. Figure 3.5 illustrates the conceptual model for *architecture description languages*.

3.2 Model-Based Architecture Description

The emergence and evolution of the modeling discipline have made models and modeling significantly pervade several disciplines in the context of Software Engineering. This is also the case for Software Architecture, but mainly in the context of the architecture description. As we discuss later in Chapter §4, architecture design is still underusing the potential of model-based approaches. In the original IEEE Std 1471 standard [IEE00] dated on 2000, the notion of *model* was explicitly used as part of architecture descriptions. In the standard, an *architecture description* and an *architecture view* were conceived as an aggregation of *models*, and an *architecture viewpoint* was said to establish the methods for the models in the views conforming to the viewpoint. In the ISO/IEC/IEEE 42010:2011 standard [ISO11], the conceptual model introduces the concept of *model kind*. As we reviewed in Section §3.1, *model kinds* allow a more accurate definition of viewpoints and views, and the definition of *architecture frameworks* and *architecture description languages*. Thus, we argue that the modeling discipline is playing an increasing role in the practice of architecture description.

Model-based approaches are also being used for the definition of the standards themselves. Every edition of the standard uses conceptual models to describe the context and practice of architecture description. The conceptual model is actually a *domain model* [Fow97], in the sense that it captures the most important types of entities or concepts, and their properties and relationships, within the domain of the practice of architecture description. Even though it is not explicitly established in the standard, the FAQs pages [ISO13] for the standard state that *"these ideas [concepts in the domain] can be captured via a conceptual model or, metamodel, establishing the key concepts and terms for talking about architectures and architecture descriptions."* We argue that this assertion is inaccurate and might be misleading. The standard does provide a conceptual model capturing with it the key concepts and terms for talking about the architecture description practice, i.e. a domain model of the practice. However, this conceptual model cannot be understood as a *metamodel* for architectures or architecture descriptions because: (1) concepts in the conceptual model do not follow the 3+1 organization of the metamodeling approach, and (2) architecture descriptions require modeling-in-the-large techniques. We explain both arguments next.

The metamodeling approach uses a 3+1 organization to separate the real-world system, concepts or entities (the '1'), from the three layer hierarchical structure of model conformance (the '3'). As Figure 2.2 in page 44 illustrates, the *system* (what is being modeled) pertains to the real-world (known as M0 level), while in the modeling-world, terminal models at M1 level conform to metamodels at M2 level that conform to a self-conforming metametamodel at M3 level. For the standard's conceptual model to be a metamodel for architecture descriptions, it must include concepts pertaining the M2 level exclusively. However, from our understanding, the concepts in the conceptual model pertain to multiple levels, particularly M0 and M2. In the illustration of the conceptual model in Figure 3.2, we lay out the concepts of the conceptual model to distinguish three columns. Although this layout differs from the one used in the standard, the meaning remains unchanged. The concepts at the left column represent real-world concepts while the concepts at the center and right columns represent constructions for describing architectures. An *architecture description* (in the center column) is a representation of an *architecture* of a *system-of-interest* (both at the left column). Then,

a *system-of-interest*'s *architecture* is what is being modeled (i.e. the system or real-world entities at M0 level) and any *architecture description* is a model (at M1 level) of an architecture. The concepts at the center and right columns, including *architecture description*, represent the constructs that can be used to build a description, and hence, they are at the M2 level. The case of *stakeholders* and *concerns* is harder as the conceptual model may be using them to represent (a) the real-world concepts, or (b) the stakeholders and concerns identified, captured and documented in an architecture description. These cases are actually different as case (a) states that the concepts refer to what is being modeled and case (b) states that they refer to the model. For the conceptual model to be a metamodel, case (b) should be the standard's intention for these concepts, and not (a). However, to this end, the association between *architecture description* and these concepts need to be defined as an aggregation, i.e. using a white diamond at the *architecture description* association-end. Notice that aggregations are used to model every other concept that conforms the architecture description, for instance the architecture viewpoints and architecture views. Then, the absence of an aggregation leads to understand the concepts of *stakeholders* and *concerns* as real-world concepts, i.e. case (a) seems to be the standard's intention. If that is the case, conceiving the conceptual model as a metamodel requires these concepts to be removed and the corresponding concepts for case (b) to be included.

Even if the discussed adaptations to the conceptual model are made, the resulting model cannot be understood as a metamodel for architecture descriptions. An *architecture description* is a complex artifact which is generally composed of multiple artifacts, many of them being models. For instance, an architecture description is an aggregation of several *architecture viewpoints* and *architecture views*, which in turn are aggregations of *model kinds* and *architecture models*, respectively. These artifacts are expressed in different languages, with their own notation and semantics. It is hard to conceive a single language to represent them all. In fact, as we discussed in Section §3.1.2, extensible Architecture Description Languages are used today to allow architects modeling different aspects of an architecture. As we explained in Section §2.3, there is a distinction between modeling-in-the-small and modeling-in-the-large. While the former embraces techniques for working on single models, the latter is aimed to scenarios involving a large number of modeling artifacts. Building an architecture description using model-based techniques must not be regarded as modeling-in-the-small, due to the size and complexity of what needs to be modeled. On the contrary, the architecture description practice is actually modeling-in-the-large as it involves building dozens of modeling artifacts to describe a single architecture. Moreover, in order to achieve encapsulation of reusable architecture knowledge on architecture descriptions, it is imperative to use separate artifacts for separate purposes.

An adapted version of the conceptual model in which real-world concepts are omitted, would be actually modeling the different kinds of artifacts, their properties, relationships and governing rules, that are required to build architecture descriptions. If we consider these artifacts to be conceived in the context of Model-Driven Engineering, i.e. to conceive them as modeling artifacts, then such a conceptual model would represent a *metamodel* for domain-specific megamodels of architecture descriptions.

This is actually our model-based solution to achieve a homogeneous, shareable, reusable and tool-friendly mechanism for representing architecture knowledge of architecture descrip-

tion: to use a domain-specific global model management approach. We understand an architecture description as a *domain-specific megamodel* capturing every modeling artifact that is required to describe an architecture. Such a domain-specific megamodel needs to be reified by means of the current model-driven techniques. This poses the question: which are all the modeling artifacts that are needed to build an architecture description using a model-based approach? Technically, how do we understand the software architecture domain-specific megamodel approach in terms of the general-purpose megamodel approach.

In order to formally answer these questions, we use denotational semantics to define the interpretation of the concepts in the domain-specific megamodel in terms of the concepts of general-purpose megamodels as defined by the Global Model Management approach we explained in Section §2.3. The interpretation is defined by means of semantic functions. We use the conceptual model of the ISO/IEC/IEEE 42010:2011 standard [ISO11] itself as the set of terms and concepts that need to be interpreted. The adaptations that we discussed before that are required to conceive the conceptual model as the metamodel for the domain-specific megamodels are reflected in the decisions on how our denotational semantics actually interprets these concepts. Then, the semantic functions map the concepts described in Section §3.1, and instances of these concepts, to modeling artifacts as described by the megamodeling approach defined in Section §2.3.1.

In this section we define the semantic function for our model-based interpretation of architecture descriptions. First, we formalize the function domain, its codomain, and the notation we use to express the mapping. Then, we define the semantic equations of the function for concepts pertaining architecture descriptions, architecture frameworks and description languages, in separate subsections. We conclude this section analyzing how to use the interpretation to *communicate* the architecture description to stakeholders.

3.2.1 Definition of the semantic functions

Formal semantics is concerned with rigorously specifying the meaning and behavior of concepts or entities, either abstract or real-world ones. It makes the distinction between what is being formalized (the syntax) and the meaning (the semantics). Formal semantics has been mostly applied to programs and pieces of hardware as it allows to reveal ambiguities and subtle complexities in apparently crystal clear technical documentations [NN92]. Also, it can form the basis for system implementations, for tool development, and for the analysis and verification of properties on the concepts or entities being formalized. Several formalization approaches have been proposed and used during the last decades in the Formal Methods discipline to specify formal semantics, being operational, axiomatic and denotational semantics, the most widely used. In our work, we apply the denotational semantics approach as it emphasises on the definition of a correspondence between the constructs in a syntactic domain to the constructs in the semantic domain.

The denotational semantics approach maps the concepts or entities directly to their meaning, called their denotation [Sch86]. The denotation is expressed in a target language of which there is an unambiguous and precise understanding, usually mathematical entities such as numbers, sets, or functions. The application of a denotational semantics approach consists

of the definition of:

(i) the syntactic categories or structure of the domain being formalized,

(ii) the value or semantic domain being the semantical target of the formalization,

(iii) the semantic functions conforming the signatures for mappings from the syntactic to the semantic domain, and

(iv) the semantic equations or clauses conforming the rules of the mappings of the semantic functions.

We apply the denotational semantics approach to the formalization of the concepts and entities in the practice of architecture description. We define a mapping from the conceptual model defined by the ISO/IEC/IEEE 42010:2011, reviewed in Section §3.1, to the modeling constructs available in the current practice of Model-Driven Engineering. This formalization contributes with a rigorous interpretation of the constructs pertaining the architecture knowledge, into a domain that allows us to provide a homogeneous means for knowledge representation, sharing and reuse in the software architecture discipline. In this chapter we define our mapping only on the practice of architecture description. In Chapter §4 we extend our mapping to cover also architecture design.

(i) Syntactic domain

In the application of the denotational semantics approach, the syntactic structures and categories characterize the existing objects, concepts or entities in the domain being formalized [NN92]. The syntactic domain is expressed in terms of a language, i.e. the objects in the syntactic domain are those that can be expressed using the constructs of the language. For textual languages, their definition usually involves a concrete syntax and an abstract syntax. The concrete syntax is specified using context-free grammars expressed in Backus-Naur Form (BNF) and it represents the surface of the language. The abstract syntax takes the form of a tree of terms and represents the deep structure of the language in which all potential ambiguities of the concrete syntax are resolved, considering the precedence and associativity of the language constructs. The abstract syntax defines the decomposition of language phrases into their sub-phrases, recursively [SK95]. The application of the denotational semantics approach is compositional, i.e. the denotation of a language construct is defined in terms of the denotation of its sub-phrases. For visual languages the scenario is more complex. There is still a distinction between a concrete and an abstract syntax. The former refers to the icons and shapes used by the language, while the latter refers to the internal structure of these interconnected shapes. Generally, the abstract syntax abstracts geometrical details such as size, position and shape on the concrete syntax (up to topological equivalence), and it is conceived as a directed labeled multi-graph [Erw98]. In the case of modeling languages, a language is defined by means of a metamodel which defines the type of entities, their properties and interrelationships, without imposing any concrete syntax to represent the language constructs. A visual concrete syntax is generally defined for modeling languages in order to allow modelers to work with the models and to communicate them. However, the concrete syntax is used to build diagrams of model elements within models,

not to build models themselves. A metamodel models the language constructs that can be used to model a given domain, and hence, it can be used as a syntactic domain. In the cases of visual and modeling languages, the fact that their abstract syntax can be regarded as a labeled multi-graph imposes additional challenges when trying to preserve the compositionality of the denotational semantics. Being a graph, the syntax provides no tree structure to guide the decomposition. We discuss this issue when we define the semantic function later in (iii).

We have two candidates to be considered the syntactic domain of the semantic function: (1) the conceptual model proposed in the standard, and (2) the metamodel for the domain-specific megamodel of architecture descriptions that we obtain from adapting the conceptual model. As we discussed in the introduction to Section §3.2, the conceptual model cannot be understood as a metamodel for architecture descriptions as it is. By adapting it through removing the real-world concepts *system-of-interest* and *architecture* and considering *stakeholders* and *concerns* as the part of the description that captures the corresponding real-world elements, we obtain a metamodel for a domain-specific megamodel. One might argue that candidate (1) is not actually valid as the syntactic domain as it does not describe a language. On the contrary, it is a valid syntactic domain. Our argument states that it is not a valid metamodel for architecture description, but it is still a valid domain model for the architecture description practice. As such, it defines a language that allow us to express facts in the practice. In the case of candidate (2), even though it is the most appropriate one, the fact that it misses some concepts that pertain the standard might lead to confusion or might be consider incomplete. As a consequence, we opt to use the conceptual model (1) as the syntactic domain of our denotational semantics specification. As every concept in (2) is included in (1), by choosing (1) we are also covering (2). Moreover, when we define the semantic equations for each concept and concept instance later in Section §3.2.2, it can be observed that the real-world concepts *system-of-interest* and *architecture* contribute with no modeling artifact at all, and that *stakeholders* and *concerns* contribute as if they were the elements being captured by an architecture description. This is the exact effect that is expected if we would have interpreted the metamodel (2) directly.

Thus, we use the conceptual model defined in Section §3.1 as the model of the language which provides the constructs for representing architecture knowledge in the architecture description practice. We rely on set theory and set notation to specify the syntactic domain. We introduce and explain the definitions in what follows. Table 3.1 provides a condensed summary of these definitions.

We define $\mathcal{A}^{\mathsf{C}}$ as the set of all the concepts defined in the conceptual model of the ISO/IEC/IEEE 42010:2011 standard. $\mathcal{A}^{\mathsf{C}}$ is a finite set defined by extension with the terms or concepts depicted in Figures 3.2, 3.4 and 3.5. For instance, $\mathsf{Architecture}$, $\mathsf{Concern}$, $\mathsf{ArchitectureDescription}$ and $\mathsf{ModelKind}$ are some of the elements of $\mathcal{A}^{\mathsf{C}}$. We use $\mathsf{X} : \circ$ to denote $\mathsf{X} \in \mathcal{A}^{\mathsf{C}}$, i.e. that X is a concept defined in the conceptual model.

We define $\mathcal{A}_{\mathsf{c}}$ as the set of all elements or instances of the concepts in the conceptual model. Each element in $\mathcal{A}_{\mathsf{c}}$ is an instance of a single concept which determines its properties and relationships, according to the governing structure defined in the conceptual model. $\mathcal{A}_{\mathsf{c}}$ is an infinite set. Given $\mathsf{X} : \circ$, we use $\mathsf{x} :_{\circ} \mathsf{X}$ to denote both that $\mathsf{x} \in \mathcal{A}_{\mathsf{c}}$ and that it is an

$\mathcal{A}^{\mathsf{C}}$	$\overset{\text{def}}{=}$	$\{x \mid x \text{ is a concept from the conceptual model}\}$
$\mathsf{X} : \circ$	$\overset{\text{def}}{=}$	$\mathsf{X} \in \mathcal{A}^{\mathsf{C}}$
$\mathcal{A}_{\mathsf{c}}$	$\overset{\text{def}}{=}$	$\{x \mid x \text{ is an instance of a concept from the conceptual model}\}$
$\mathsf{x} :_\circ \mathsf{X}$	$\overset{\text{def}}{=}$	$\mathsf{X} : \circ \quad \wedge \quad \mathsf{x} \in \mathcal{A}_{\mathsf{c}} \quad \wedge \quad \mathsf{x} \text{ is an instance of the concept } \mathsf{X}$
$\mathcal{A}$	$\overset{\text{def}}{=}$	$\mathcal{A}^{\mathsf{C}} \;\cup\; \mathcal{A}_{\mathsf{c}}$
$\mathcal{A}^*$	$\overset{\text{def}}{=}$	$\mathcal{P}(\mathcal{A})$

Table 3.1: *Syntactic domain of the semantic functions.*

instance of the concept X. For instance, $\mathsf{c} :_\circ \mathsf{Concern}$ states that c is an element of $\mathcal{A}_{\mathsf{c}}$ (i.e. c is an instance of concept of the conceptual model), and particularly, that c is an instance of the concept Concern.

We define $\mathcal{A}$ as the set of all concepts and concept instances in the conceptual model, formally $\mathcal{A} \overset{\text{def}}{=} \mathcal{A}^{\mathsf{C}} \cup \mathcal{A}_{\mathsf{c}}$. We also define $\mathcal{A}^*$ as the set of all possible sets of concepts and concept instances, formally $\mathcal{A}^* \overset{\text{def}}{=} \mathcal{P}(\mathcal{A})$.

We use the sets $\mathcal{A}^{\mathsf{C}}$, $\mathcal{A}_{\mathsf{c}}$, $\mathcal{A}$ and $\mathcal{A}^*$, as the syntactic domain of the different semantic functions that we define later in (iii).

We use dot notation on concept instances to refer to their attribute values and to the reachable (navigable) instances through their links. The available attributes and properties are those defined in the conceptual model. For instance, given $\mathsf{c} :_\circ \mathsf{Concern}$, we use c.stakeholders to denote the set of stakeholders $\{\mathsf{s_i} :_\circ \mathsf{Stakeholder}\}$ that are linked to the concern c through the association between Concern and Stakeholder specified in the conceptual model of Figure 3.2. We also use OCL [OMG12] to build expressions on values, collections and elements of a model. For instance, given $\mathsf{avp} :_\circ \mathsf{ArchitectureViewpoint}$, we use avp.concerns.stakeholders→flatten() to denote the set of stakeholders that are linked to a concern in the set of concerns linked to the architecture viewpoint avp. This expression is actually an OCL shorthand notation for the OCL expression avp.concerns→collect($c \mid c$.stakeholders)→flatten().

(ii) Semantic domain

In the application of the denotational semantics approach, the definition of the semantic domain usually relies on mathematical entities, mainly using set theory or category theory for this purpose. Our position is that for our specification, using a pure mathematical specification as the target of the mapping would require a level of detail that might hinder the readability and the understanding of the semantical mapping. Such a strict level of formalization is out of the scope of this work, as our goal is to use the level of rigor that provides preciseness but still facilitates understandability. Also and even more important, the goal of our semantic specification is to allow the interpretation of architecture knowledge in terms of Model-Driven Engineering (MDE) constructs. Then, instead of set or category theory, we

use the modeling approach itself as the definition of the target language of the mapping. Our semantic domain is the realm of modeling artifacts built in the context of MDE techniques. We use the Global Model Management approach to megamodeling as a conceptualization of the target of the interpretation of architecture knowledge. Our work assumes that the MDE technical space is unambiguously and precisely understood, being the formalization of its semantics out of the scope of this work. Formal semantics for MDE is being studied by the research community, both for specific constructs or languages (e.g. model transformations) and for the foundations of the modeling technical space. Particularly, Z. Diskin et al. provide in [DM12] a formalization of MDE in terms of category theory.

We define a precise syntactic notation for representing the modeling artifacts, their kind, properties and relationships, as governed by the metamodel for megamodeling defined in Section §2.3.1. The precise notation is based on set theory and uses assertions to state facts on the semantic elements in order to further characterize the denotation. Similarly to the syntactic domain, we also use OCL-like expressions to navigate the models in the semantic domain. The metamodel for megamodels determines the attributes and properties that can be used in the expressions. While we use natural language to explain and aid understanding, we rely on the precise notation for the definition of the semantic equations. Such preciseness avoids ambiguity in definitions. We discard the possible alternative of using object diagrams to illustrate the model elements instantiating the metamodel constructs. The main reason is that object diagrams are not suitable for expressing optionality or quantification, and might result confusing when showing many interconnected elements in a single diagram.

Model repository. As we explained in Section §2.2, modeling artifacts are stored and preserved in model repositories. From a technological point of view, a model repository can be a file system, a database, a software service archiving and publishing artifacts, among others [Mod08b]. However, from a conceptual point of view, a model repository can be regarded as a set of modeling artifacts, independently of both the underlying technology used to store and provide access to them, and any physical or logical structure (e.g. folders) used to organize the repository. The specific modeling environment and tools in use are responsible for providing technological support for the repository and its contained modeling artifacts. Modeling artifacts are resources that can be uniquely identified. Given any two modeling artifacts a_1 and a_2, it is possible to determine whether the assertion $\mathsf{a}_1 = \mathsf{a}_2$ holds, being $=$ the equality relation among artifacts that is reflexive, symmetric, anti-symmetric and transitive. The underlying tool support is responsible for providing an identification mechanism and to enforce the equality relation. We use $\mathsf{a}_1 \neq \mathsf{a}_2$ as a shorthand for $\neg(\mathsf{a}_1 = \mathsf{a}_2)$. Provided that a model repository is a set, we can then use set relations such as inclusion, and set operations such as union, intersection and difference, to express assertions on model repositories.

We define a *model repository* as a finite set of modeling artifacts. We define the assertion $\mathsf{R} : \blacksquare$ to state that R is a model repository. Then, for any R such that $\mathsf{R} : \blacksquare$, the assertion $\mathsf{x} \in \mathsf{R}$ states the fact that the modeling artifact x is stored in the model repository R. We define $\mathcal{R}$ as the infinite set of all conceivable modeling artifacts. We define $\mathcal{R}^*$ as the power set of $\mathcal{R}$ denoted by $\mathcal{P}(\mathcal{R})$, i.e. the set of all sets of modeling artifacts. Then we have that $\mathsf{R} : \blacksquare \Leftrightarrow \mathsf{R} \subset \mathcal{R} \Leftrightarrow \mathsf{R} \in \mathcal{R}^*$.

$\mathcal{R}$	$\stackrel{\text{def}}{=}$	$\{x \mid x \text{ is a modeling artifact}\}$
$\mathcal{R}^*$	$\stackrel{\text{def}}{=}$	$\{x \mid x \text{ is a finite set of modeling artifacts}\}$
	$\equiv$	$\{x \mid x \text{ is a model repository}\}$
	$\equiv$	$\mathcal{P}(\mathcal{R})$
$\mathsf{R} : \blacksquare$	$\stackrel{\text{def}}{=}$	R is a model repository
	$\equiv$	R is a finite set of modeling artifacts
	$\equiv$	$\mathsf{R} \subset \mathcal{R} \quad \wedge \quad \mathsf{R}$ is finite
	$\equiv$	$\mathsf{R} \in \mathcal{R}^* \quad \wedge \quad \mathsf{R}$ is finite
$\mathcal{R}^{\mathsf{E}}$	$\stackrel{\text{def}}{=}$	$\{x \mid x \text{ is a kind of modeling artifact}\}$
	$\equiv$	$\{x \mid x \text{ is a metaclass in the metaclass hierarchy of } \mathsf{Entity}\}$
	$\equiv$	$\{\mathsf{Entity}, \mathsf{Model}, \mathsf{ExternalEntity}, \mathsf{ReferenceModel}, \mathsf{Metametamodel}, \mathsf{Metamodel}, \mathsf{TerminalModel}, \mathsf{TransformationModel}, \mathsf{WeavingModel}, \mathsf{Megamodel}\}$
$\mathcal{R}^{\mathsf{C}}$	$\stackrel{\text{def}}{=}$	$\{x \mid x \text{ is a concrete kind of modeling artifact}\}$
	$\equiv$	$\{x \mid x \text{ is a concrete sub-metaclass of } \mathsf{Entity}\}$
	$\equiv$	$\{\mathsf{ExternalEntity}, \mathsf{Metametamodel}, \mathsf{Metamodel}, \mathsf{TerminalModel}, \mathsf{TransformationModel}, \mathsf{WeavingModel}, \mathsf{Megamodel}\}$
$\mathsf{X} : \bullet$	$\stackrel{\text{def}}{=}$	$\mathsf{X} \in \mathcal{R}^{\mathsf{E}}$
$\mathsf{x} :_{\bullet} \mathsf{X}$	$\stackrel{\text{def}}{=}$	$\mathsf{x} \in \mathcal{R} \quad \wedge \quad \mathsf{X} : \bullet \quad \wedge \quad \mathsf{x}$ is a modeling artifact of kind X

Table 3.2: *Semantic domain: model repositories.*

The kind of modeling artifacts that can be preserved in a model repository depends on the technological support of the modeling environment and tools. However, as we discussed in Section §2.3, the Global Model Management (GMM) approach provides a conceptualization of all kinds of modeling artifacts that are available for use. Using this conceptualization to structure the semantic domain allows us to keep the interpretation technology-independent. We use the top-level metamodel defined in Section §2.3.1 as the categorization and classification of the modeling artifacts that can be stored in a model repository. This technology-independent metamodel provides a coherent and consistent foundation that can be extended to cope with technology-specific constructs. As we discussed in Section §2.3.2, several extensions are already available, together with the corresponding tool support. Then, although we structure the semantic domain by means of the GMM conceptualization of modeling artifacts, practitioners must rely on specific technological support when applying the interpretation in real-world projects.

GMM distinguishes two kinds of elements in the metamodel for megamodels, namely Entity and Relationship. As we explained in Section §2.3.1, Entity is the abstract metaclass that represents actual modeling artifacts stored in the model repository or that are reachable in the modeling environment, and Relationship represents a semantic relationship between elements and does not represent actual modeling artifacts in the repository. Given a repository R and

a modeling artifact x stored in R, i.e. $\mathsf{R} : \blacksquare \ \wedge \ \mathsf{x} \in \mathsf{R}$, the kind of the modeling artifact x is Entity, and particularly, one of the concrete sub-metaclasses of Entity. We define $\mathcal{R}^{\mathsf{E}}$ as the set of all the metaclasses that represent entities, i.e. Entity and all of its sub-metaclasses. We define $\mathcal{R}^{\mathsf{C}}$ as the set of all concrete sub-metaclasses of Entity. Then, by definition we have that $\mathcal{R}^{\mathsf{C}} \subset \mathcal{R}^{\mathsf{E}}$.

We define the assertion $\mathsf{X} : \bullet$ to state that $\mathsf{X} \in \mathcal{R}^{\mathsf{E}}$, i.e. X is a kind of modeling artifact, either abstract or concrete. Given $\mathsf{X} : \bullet$, we define the assertion $\mathsf{x} :_{\bullet} \mathsf{X}$ to denote that x is a modeling artifact of kind X. For instance, the assertion $\mathsf{m} :_{\bullet}$ TerminalModel states the fact that m is a modeling artifact, and that in particular, it is a *terminal model*. It is important to notice that for every modeling artifact x, its kind is always a single concrete metaclass in $\mathcal{R}^{\mathsf{C}}$ and at the same time, every concrete or abstract super-metaclass in $\mathcal{R}^{\mathsf{E}}$ of that concrete metaclass, as defined by the metaclass hierarchy of the metamodel for megamodels. For instance, provided that $\mathsf{m} :_{\bullet}$ TerminalModel holds, we can derive that $\mathsf{m} :_{\bullet}$ Model and $\mathsf{m} :_{\bullet}$ Entity also hold. Table 3.2 provides a condensed summary of these definitions for the semantic domain.

Conforms-to assertion. The *conforms-to* relationship introduced in [Béz05b] relates a model m to a reference model rm that models the modeling language used to express the model m. This relationship is captured in the metamodel for megamodels by the association between Model and ReferenceModel. We define the assertion $\mathsf{m} \lhd \mathsf{rm}$ to state that the modeling artifact m of kind Model *conforms-to* the modeling artifact rm of kind ReferenceModel. The assertion holds when $\mathsf{m} :_{\bullet}$ Model, $\mathsf{rm} :_{\bullet}$ ReferenceModel, and the OCL expression m.conformsTo = rm evaluates to **true**. Table 3.3 defines the *conforms-to* assertion. J. Favre et al. in [FN05] represent this assertion using the symbol χ.

In the Global Model Management approach to megamodeling [Mod09], the core metamodel packages provide no treatment for entities that are not models. It is the extension package for Textual Concrete Syntax (TCS) that introduces the concept of textual entity. In our definition, we include a base metaclass for all kinds of entities that are not actually models, namely ExternalEntity. External entities do not comply to the Model-Driven Engineering approach in the sense that they are expressed in languages whose structure is not guided by a known metamodel. Hence, general-purpose model transformation languages can rarely be used to read, create or manipulate external entities, due to their particular technology-specific representation. Instead, special purpose-specific tools are required to deal with them. However, external entities are artifacts of interest in development projects and hence, they have to be dealt with when applying Model-Driven Development techniques. As we explained in Section §2.3.1, in our metamodel for the megamodels we also introduce the concept of ExternalEntityKind as a means for specifying the purpose of the external entities. Given an external entity $\mathsf{e} :_{\bullet}$ ExternalEntity, its ExternalEntityKind k determines the kind of content defined in the external entity e. It is important to notice that k is not a modeling artifact and hence it is not preserved in a model repository. The kind k is used to further characterize external entities, like e, and this information may be preserved in a megamodel if the modeling artifact e is represented in the megamodel. Symmetrically to the *conforms-to* relationship between a model and its reference model, we define in our metamodel an *is-kind-of* relationship between an external entity and its kind; Figure 2.11 illustrated this relationship. Then, we define the assertion $\mathsf{e} \unlhd k$ to state that the artifact e is of kind k. The assertion

$\mathsf{x}_1 \lhd \mathsf{x}_2 \overset{\text{def}}{=} \mathsf{x}_1 :_\bullet \mathsf{Model} \;\wedge\; \mathsf{x}_2 :_\bullet \mathsf{ReferenceModel} \;\wedge\; \mathsf{x}_1.\mathsf{conformsTo} = \mathsf{x}_2$
$\mathsf{x} \unlhd k \overset{\text{def}}{=} \mathsf{x} :_\bullet \mathsf{ExternalEntity} \;\wedge\; k \text{ is an } \mathsf{ExternalEntityKind} \;\wedge\; \mathsf{x}.\mathsf{kind} = k$
$\mathsf{x} \lessdot y \overset{\text{def}}{=} (\mathsf{x} :_\bullet \mathsf{Model} \Rightarrow \mathsf{x} \lhd y) \;\wedge\; (\mathsf{x} :_\bullet \mathsf{ExternalEntity} \Rightarrow \mathsf{x} \unlhd y)$
$\equiv \mathsf{x} \lhd y \;\vee\; \mathsf{x} \unlhd y$
$\mathsf{x} \multimap \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ for $n \in \mathbb{N}$
$\overset{\text{def}}{=} \mathsf{x} :_\bullet \mathsf{Metamodel} \;\wedge$
$\forall \mathrm{i}.\; \mathsf{x}_\mathrm{i} :_\bullet \mathsf{Metamodel}$ for $0 < \mathrm{i} \leq n \;\wedge$
$\mathsf{x}.\mathsf{requires} = \mathbf{Set}\{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$

Table 3.3: *Semantic domain: conforms-to, is-kind-of and requires assertions.*

holds when $\mathsf{e} :_\bullet \mathsf{ExternalEntity}$, k is an ExternalEntityKind, and the OCL expression $\mathsf{e.kind} = k$ evaluates to **true**. Table 3.3 defines the *is-kind-of* assertion.

The *conforms-to* and *is-kind-of* assertions are different as they are applied on modeling artifacts (*entities*) of different nature. While *conforms-to* is used for *models*, *is-kind-of* is used for *external entities*. However, both assertions have the same purpose: to provide classification and characterization of modeling artifacts. We define the *is-characterized-by* assertion for modeling artifacts, denoted by $\lessdot$. The assertion $x \lessdot y$ for $x :_\bullet \mathsf{Entity}$ states that x is characterized by y. When x is a model, then the assertion states that x *conform-to* y, where $y :_\bullet \mathsf{ReferenceModel}$ must hold. When y is an external entity, then the assertion states that x *is-kind-of* y, where y is an ExternalEntityKind. Formally, $x :_\bullet \mathsf{Model} \Rightarrow x \lhd y$ and $x :_\bullet \mathsf{ExternalEntity} \Rightarrow x \unlhd y$. Table 3.3 defines the *is-kind-of* assertion.

Requires assertion. The *requires* relationship used for ontological metamodeling relates a metamodel mm to a possibly empty finite set of metamodels whose constructs are used to properly define mm. This relationship provides a modularization mechanism for metamodels facilitating reuse. Additionally, it allows the definition of a base metamodel acting as a framework for multiple metamodels that specialize (require) the base metamodel. We define the assertion $\mathsf{mm} \multimap \{\mathsf{mm}_1, \ldots, \mathsf{mm}_n\}$ for $0 < \mathrm{i} \leq n, n \in \mathbb{N}$, to state that the metamodel mm *requires* the metamodels mm_i. Particularly, for the case $n = 0$ we have $\mathsf{mm} \multimap \varnothing$ indicating that mm requires no metamodel and hence it is self-contained. We use the shorthand notation $\mathsf{mm} \multimap \mathsf{mm}'$ when the set of required metamodels is a singleton (i.e. $n = 1$). Table 3.3 defines the *requires* assertion.

Is-defined-in assertion. Provided that most of the target elements in the semantic domain are actually models, i.e. modeling artifacts of kind Model, we need a mechanism to express assertions on the definitions made within these models. Particularly, we need to assert the existence of certain model elements in a model, together with their attribute values and their relationships to other model elements within the model. We define the *is-defined-in* relationship, denoted by ε between a model element e and a modeling artifact $\mathsf{m} :_\bullet \mathsf{Model}$, to state the fact that the model element e is defined in the model m, i.e. e is a model element

$\mathrm{e} \varepsilon \mathsf{m}$	$\stackrel{\text{def}}{=}$	$\mathsf{m} :_{\bullet} \mathsf{Model} \quad \wedge \quad$ e is a model element defined in the model m
$\mathcal{F}_{\mathsf{RM}}$	$\stackrel{\text{def}}{=}$	$\{x \mid x$ is a model fragment based on the reference model $\mathsf{RM}\}$
	$\equiv$	$\left\{ \begin{array}{l} x \mid x \text{ is formed by instances of metaclasses of the reference model RM,} \\ \quad \text{defining values for some of their attributes and links among them,} \\ \quad \text{without enforcing invariants or structural constraints on cardinality} \end{array} \right\}$
$\mathsf{f} \in \mathcal{F}_{\mathsf{RM}}$	$\stackrel{\text{def}}{=}$	$\mathsf{RM} :_{\bullet} \mathsf{ReferenceModel} \quad \wedge$ f is a model fragment based on the reference model RM
$\mathsf{f} \lesssim \mathsf{m}$	$\stackrel{\text{def}}{=}$	$\mathsf{RM} :_{\bullet} \mathsf{ReferenceModel} \quad \wedge \quad \mathsf{m} :_{\bullet} \mathsf{Model} \quad \wedge \quad \mathsf{m} \lhd \mathsf{RM} \quad \wedge \quad \mathsf{f} \in \mathcal{F}_{\mathsf{RM}} \quad \wedge$ $\forall \mathrm{e} \, \varepsilon \, \mathsf{f}. \; \exists \mathrm{e}' \, \varepsilon \, \mathsf{m}. \; \mathrm{e}'$ is the matching model element of e

Table 3.4: *Semantic domain: model fragments.*

of the model m. The *is-defined-in* assertion can be used with any kind of model, and it is particularly useful in our semantic function to state which are the model elements populating the resulting megamodels. We use OCL expressions to refer to attributes and to navigate associations, and we use Boolean OCL expressions to express facts on the model elements. Table 3.4 defines the *is-defined-in* assertion. J. Favre et al. in [FN05] represent this assertion using the symbol ϵ.

Model fragment assertion. We introduce the notion of *model fragment* to group definitions of model elements and some of their attribute values and links. A model fragment actually consists of a fragment of a model, i.e. it can be formed by model elements, values for some or all of their attributes, and links between these elements. Specifically, given a reference model RM, a model fragment based on RM is formed by model elements that are instances of the metaclasses defined in RM, and whose attribute values and links are also governed by the reference model. A model fragment is not necessarily a model by itself. Although it is structurally governed by a reference model, a model fragment does not need to satisfy the structural constraints imposed by the cardinality of attributes and association ends. Moreover, a model fragment can break any or all of the invariants imposed by the reference model. In other words, while a model fragment might be ill-formed with respect to structural constraints or invariants, a model must not.

We define $\mathcal{F}_{\mathsf{RM}}$ as the infinite set of all conceivable model fragments for the reference model RM. Then, the assertion $\mathsf{f} \in \mathcal{F}_{\mathsf{RM}}$ states that f is a model fragment of the reference model RM. We extend the *is-defined-in* relationship to relate a model element to a model fragment in which it is defined. Then, given a model fragment $\mathsf{f} \in \mathcal{F}_{\mathsf{RM}}$, the assertion $\mathrm{e} \, \varepsilon \, \mathsf{f}$ states that the model element e is defined in the model fragment f. When appropriate, we include the reference model as a subscript of a model fragment to enhance readability; for instance, we might write the previous assertion as $\mathrm{e} \, \varepsilon \, \mathsf{f}_{\mathsf{RM}}$.

Given $\mathsf{f} \in \mathcal{F}_{\mathsf{RM}}$ and a particular modeling artifact $\mathsf{m} :_{\bullet} \mathsf{Model}$ such that $\mathsf{m} \lhd \mathsf{RM}$ holds, we define the *is-fragment-of* assertion $\mathsf{f} \lesssim \mathsf{m}$ to state that (a) each model element defined in the model fragment has a corresponding model element in the model, (b) the value for

each defined attribute of the model element in the fragment is equal to the value of the corresponding attribute of the model element in the model, and (c) for all links for the model element in the fragment, the corresponding model element in the model has a corresponding link through the same association. In other words, all definitions in the model fragment are also defined in the model. It is important to notice that the model might have model elements that have no corresponding model element in the model fragment, and that the model elements in the model might present additional attribute values and links that are not present in the model fragment. We assume that there is a mechanism to uniquely identify model elements within the model fragment and the model, and that this mechanism can be used to make the correspondence between model elements. Table 3.4 summarizes these definitions.

Representation-of functions. In the modeling discipline, there is a distinction between what is being modeled and the model itself. The former is generally called *system* and the latter is a *model* of the system. This aspect was described in Section §2.2.2 as the mapping criterion and was illustrated in Figure 2.2. This relation between a system and a model is called *representation-of* in [Béz04] and is denoted by the symbol μ in [FN05]. Actually, given a particular real-world domain, system or entity, there are multiple ways to define a model of it. Using different modeling languages to build the model clearly renders different models of the same system. However, even using a single modeling language, there may be multiple ways to express the same system in terms of the constructs of this language, each of these ways actually representing the system. Consider for instance using different levels of detail or different levels of refinement. In practice, for the same system under study, different modelers generally come up with different models, even in the same working context.

Then, for a particular real-world domain and a particular modeling language, there are many and different ways to reify the *representation-of* relation. If we conceive the corresponding *representation-of* function that takes a particular system or domain under study and produces its representation in a target modeling language, each way of representing the system conforms a valid function. As a consequence, for a given domain X and a given reference model RM, there is a family (set) of functions, each of them encapsulating a modeler's decision or intention on how to represent each element in the domain X in terms of the constructs of the reference model RM. We denote by Θ^X_{RM} the family of representation-of functions from the particular domain X to the particular reference model RM. Each function θ^X_{RM} of the family Θ^X_{RM} consists of a partial function $\theta^X_{\mathsf{RM}} : X \hookrightarrow \mathcal{F}_{\mathsf{RM}}$ that maps elements in the domain X to a model fragment based on the reference model RM.

Our model-based interpretation relies on representation-of functions to provide a means to the practitioners community to customize <u>how</u> to proceed to populate the actual models with model elements in order to represent architecture concepts and concept instances. By this means we avoid fixing the interpretation because otherwise it would not succeed to encapsulate the wide spectrum of the current architecture knowledge, and it would impose one particular representation. We denote by μ^X_{RM} the particular representation function <u>selected</u> by the practitioner for the customization of the interpretation. Then, the assertion $\mu^X_{\mathsf{RM}} \in \Theta^X_{\mathsf{RM}}$ holds by definition. Then, given a element $x \in X$ and a model m such that $\mathsf{m} \triangleleft \mathsf{RM}$, the assertion $\mu^X_{\mathsf{RM}}(x) \lesssim \mathsf{m}$ states that the selected representation of x renders a model fragment

of the model m. The semantic equations for our semantic function includes several assertions of this kind. Whenever the domain and the reference model of the representation-of function μ can be inferred from the context, we use subscript as a shorthand for the application of the selected function of the family. For example, given a concept instance c of a concept C and a model m conforming to the reference model rm, formally $\mathsf{c} :_{\circ} \mathsf{C} \wedge \mathsf{m} :_{\bullet} \mathsf{Model} \wedge \mathsf{rm} :_{\bullet} \mathsf{ReferenceModel} \wedge \mathsf{m} \triangleleft \mathsf{rm}$, we express the assertion $\mu^{\mathsf{C}}_{\mathsf{rm}}(\mathsf{c}) \lesssim \mathsf{m}$ simply as $\mathsf{c}_{\mu} \lesssim \mathsf{m}$ when the domain C and the reference model rm can be inferred from the context.

In our formalization, we repeatedly use a particular family of representation-of functions for mapping modeling artifacts to model elements defined in a megamodel. It is important to notice that when we refer to *modeling artifact*, we refer to the actual resource stored in a model repository, and not to any model element in a megamodel representing the modeling artifact – recall that there is a distinction between the modeling artifact in the model repository (system) and a representation of such an artifact in a megamodel (model). For instance, given $\mathsf{R} : \blacksquare$, $\mathsf{m} :_{\bullet} \mathsf{TerminalModel}$, and $\mathsf{mgm} :_{\bullet} \mathsf{Megamodel}$, we use $\mathsf{m} \in \mathsf{R}$ to state that the modeling artifact m is stored in the model repository R. However, the assertion $\mathsf{m} \in \mathsf{mgm}$ is ill-formed as mgm is not a set of modeling artifacts, it is conformed by model elements representing modeling artifacts, not by modeling artifacts themselves. In order to state that the model m is represented in the megamodel mgm, we need a *representation-of* function to produce the model fragment that represents m in terms of megamodeling constructs, and the *is-fragment-of* assertion to state that the produced model fragment is a fragment of mgm.

Let MgMM be the metamodel for megamodels defined in Section §2.3.1, we define the family of *representation-of* functions $\Theta^{\mathcal{R}^*}_{\mathsf{MgMM}}$ that takes a set of modeling artifacts and renders the model fragment defining the model elements that represent the modeling artifacts in the set. Provided that our model-based interpretation produces its denotation in terms of modeling artifacts directly characterized by the modeling constructs of MgMM, we can select the function $\mu^{\mathcal{R}^*}_{\mathsf{MgMM}} : \mathcal{R}^* \rightarrow \mathcal{F}_{\mathsf{MgMM}}$ that is total and renders the most possibly detailed representation for every modeling artifact. We use ν to denote this function. The resulting model fragment for a set of modeling artifacts defines a model element for every modeling artifact being mapped, together with all the known properties and relationships that can be derived from the assertions that characterize the modeling artifacts. The function ν is eager as it includes in the model fragment as much information as possible from the set of assertions that are known to hold. Also, ν attaches *source information* to every model element it defines in the resulting model fragment. The source information of a modeling artifact is the *concept* that it is denoting and the name or identifier of the *concept instance* it is denoting, if any. This information is captured by means of metadata attached to the model elements, using the key-value pairs defined for each model element as we studied in Section §2.3.1 and illustrated in Figure 2.5. We do not define the function ν explicitly as it can be straightforwardly derived from the definitions and assertions used in the semantic equations.

Continuing with the example above, given $\mathsf{R} : \blacksquare$, $\mathsf{m} :_{\bullet} \mathsf{TerminalModel}$, $\mathsf{R} = \{\mathsf{m}\}$, and $\mathsf{mgm} :_{\bullet} \mathsf{Megamodel}$ which is not in R in this example, we can now use the assertion $\nu(\{\mathsf{m}\}) \lesssim \mathsf{mgm}$ to state the fact that there is a model element in mgm representing the terminal model m, and that the known assertions on m are also recorded in mgm. For the application of ν to a singleton set, as shorthand notation we either omit the curly braces indicating a set, or we

Θ^X_{RM} for a domain $X \quad \wedge \quad \mathsf{RM} :_\circ \mathsf{ReferenceModel}$
$\stackrel{\text{def}}{=} \{\theta : X \hookrightarrow \mathcal{F}_{\mathsf{RM}} \mid \theta$ is a *representation-of* function of X using $\mathsf{RM}\}$
$\mu^X_{\mathsf{RM}} : X \hookrightarrow \mathcal{F}_{\mathsf{RM}} \quad \in \quad \Theta^X_{\mathsf{RM}}$
$\stackrel{\text{def}}{=}$ the selected *representation-of* function embodying the modeler's intention on how to customize the model-based interpretation when representing X using RM
$\nu : \mathcal{R}^* \hookrightarrow \mathcal{F}_{\mathsf{MgMM}}$
$\stackrel{\text{def}}{=} \mu^{\mathcal{R}^*}_{\mathsf{MgMM}}$
$\equiv$ the selected function of the family $\Theta^{\mathcal{R}^*}_{\mathsf{MgMM}}$ that is total, eager, and renders the most possibly detailed representation for every modeling artifact

Table 3.5: *Semantic domain: representation-of functions.*

use ν as subscript. For example, we use $\nu(\mathsf{m})$ or m_ν as a shorthand for $\nu(\{\mathsf{m}\})$.

Table 3.5 summarizes these definitions.

Model weavings. A WeavingModel is a specific kind of modeling artifact that can be preserved in a model repository. We use the assertion $\mathsf{wm} :_\bullet \mathsf{WeavingModel}$ to state the fact that wm is a weaving model. However, by solely stating this fact we are omitting important information that further characterizes the weaving model. Particularly, we are missing which is the *model weaving* relationship that the weaving model is actually reifying. In the metamodel for megamodels, ModelWeaving is the specific kind of Relationship that captures the relation between the models being woven and the model that weaves them. We need a specific kind of construct or expression that allows us to denote *model weavings* and a specific kind of assertion that allows us to state that a *weaving model* realizes or reifies a model weaving.

We define the construct $\mathsf{mw}\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle$ to describe or denote a *model weaving* mw on the models m_i, for $1 \le \mathrm{i} \le n, 1 \le n$. The construct mw is neither a *modeling artifact* nor a *model element* defined in a megamodel, it is a descriptor of a model weaving relationship between a set of models. We define *Weavings* as the set of all conceivable model weaving descriptors. Then, we have that $\mathsf{mw} \in \mathit{Weavings}$ holds.

We define the *realizes* relationship to capture the fact that *weaving models* realize or reify *model weavings*. Given the modeling artifact $\mathsf{wm} :_\bullet \mathsf{WeavingModel}$ and the model weaving descriptor $\mathsf{mw}\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle \in \mathit{Weavings}$, we define the assertion $\mathsf{wm} \blacktriangleright \mathsf{mw}$ to state that weaving model wm realizes the model weaving mw. When possible, we use the full form $\mathsf{wm} \blacktriangleright \mathsf{mw}\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle$ to skip defining each of them separately.

In order to capture model weavings in megamodels, we extend the representation-of function ν to also produce the representation of model weaving descriptors. To this end, we redefine the domain of ν to be $\mathcal{R}^* \cup \mathit{Weavings}$, i.e. ν maps modeling artifacts and descriptors to model fragments based on the metamodel of megamodels. Then, $\nu(\mathsf{mw})$ denotes the model fragment of the metamodel for megamodels that represents the model weaving, and

we can use assertions of the form $\nu(\mathsf{mw}) \lesssim \mathsf{mgm}$ for any $\mathsf{mgm} :_\bullet \mathsf{Megamodel}$. Provided that we have defined the representation-of function ν to be eager with respect to the assertions known to hold on the arguments, having $\nu(\mathsf{wm}) \lesssim \mathsf{mgm}$ implies that $\nu(\mathsf{mw}) \lesssim \mathsf{mgm}$.

Transformations. Proceeding analogously to the case for model weaving relationships, we need a specific kind of construct or expression that allows us to denote transformations and a specific kind of assertion that allows us to state that a transformation model realizes or reifies a transformation. We define the construct $\mathsf{t}\,\langle \overline{x_1}, \ldots, \overline{x_m}, y_1, \ldots, y_n \twoheadrightarrow z_1, \ldots, z_l \rangle$ to describe or denote a *transformation* on the *reference models* or *external entity kinds* x_i, y_j, z_k, for $0 < \mathrm{i} \leq m, 0 < j \leq n, 0 < k \leq l$ with $m, n, l \in \mathbb{N}$. The construct t is neither a *modeling artifact* nor a *model element* defined in any megamodel, it is a descriptor of a transformation relationship on a set of reference models and external entity kinds. The parameters before the symbol $\twoheadrightarrow$ represent In parameters and those after represent Out parameters. The parameters with a line over their name can only occur before $\twoheadrightarrow$, and represent InOut parameters. It is important to notice that m, n and l can be 0, separately or at the same time, representing a transformation with no InOut, In and Out parameters, respectively. However, at least one of them must be greater than 0, i.e. $m+n+l > 0$. In the case that a transformation has no InOut and no In parameters (i.e. $m + n = 0$) we use an empty set $\varnothing$ before the arrow, and in the case that the transformation has no Out parameters (i.e. $l = 0$) we use an empty set after the arrow. The concrete kind of transformation, as classified in the metamodel for megamodels in Figures 2.11, 2.12, 2.13 and 2.14, can be inferred from the nature of the parameters x_i, y_j and z_k. We define *Transformations* as the set of all conceivable transformation descriptors. Then, we have that for any t of the form defined before, it holds that $\mathsf{t} \in \mathit{Transformations}$.

We define the *realizes* relationship to capture the fact that *transformation models* realize, reify or implement *model transformations*. Given the transformation model tm and the transformation descriptor $\mathsf{t} \in \mathit{Transformations}$, we define the assertion $\mathsf{tm} \blacktriangleright \mathsf{t}$ to state that the transformation model tm implements the transformation t. When possible, we use the full form $\mathsf{tm} \blacktriangleright \mathsf{t}\,\langle \overline{x_1}, \ldots, \overline{x_m}, y_1, \ldots, y_n \twoheadrightarrow z_1, \ldots, z_l \rangle$ to skip defining each of them separately.

In order to capture transformations in megamodels, we extend the representation-of function ν to also produce the representation of transformation descriptors. Analogously to the case of model weavings that we discussed before, we redefine the domain of ν to be $\mathcal{R}^* \cup \mathit{Transformations}$ and can use assertions of the form $\nu(\mathsf{t}) \lesssim \mathsf{mgm}$ and for any $\mathsf{mgm} :_\bullet \mathsf{Megamodel}$.

Transformation records. We proceed analogously for *transformation records* of *transformations*; transformation records were illustrated in Figure 2.11. We define the construct $\mathsf{r} \downarrow \mathsf{t}\,\langle \mathsf{e}_1, \ldots, \mathsf{e}_n \twoheadrightarrow \mathsf{f}_1, \ldots, \mathsf{f}_l \rangle$ to describe the transformation record r that uses the entities e_i as In parameter values and obtains f_j as Out parameter values when applying the transformation t, for $0 < \mathrm{i} \leq n, 0 < j \leq l$ with $n + l > 0$. The parameter values e_i and f_j are actual modeling artifacts that must match the parameters of the transformation t. Each parameter value of r is either a *model* that conforms to the *reference model* of its corresponding parameter of t, or an *external entity* that is of kind of the *external entity kind* of its corresponding parameter. The correspondence between parameter values and parameters is determined by the

position (index) in the definition of r and t. We define *Records* as the set of all conceivable transformation record descriptors.

A transformation record r is neither a *modeling artifact* nor a *model element* in any megamodel, it is a descriptor that records the effect (i.e. the output entities) of executing a *transformation* on the input entities. There is no corresponding *realizes* relationship for transformation records as the metamodel for megamodels defines no entity to this purpose. However, as transformation records can be captured in megamodels, we extend the representation-of function ν to also produce the representation of a transformation record descriptor. The domain of ν is redefined to be $\mathcal{R}^* \bigcup$ *Records*, and we use assertions of the form $\nu(\mathsf{r} \downarrow \mathsf{t}\,\langle \mathsf{e}_1, \ldots, \mathsf{e}_n \twoheadrightarrow \mathsf{f}_1, \ldots, \mathsf{f}_l \rangle) \lesssim \mathsf{mgm}$ for $\mathsf{mgm} :_\bullet \mathsf{Megamodel}$ to state that the transformation record r is represented in the megamodel mgm.

Finally, we define *Descriptors* to be the set of all conceivable descriptors, i.e. every model weaving in *Weavings*, every transformation in *Transformations* and every transformation record in *Records*. Then, the representation-of function ν is actually redefined to accept elements in the domain $\mathcal{R}^* \bigcup$ *Descriptors*.

Table 3.6 summarizes these definitions.

(iii) Semantic function

In the application of the denotational semantics approach, semantic functions are used to map elements in the syntactic domain to elements in the semantic domain. These functions are defined by their signature, precisely by their domain and codomain, and by the semantic equations that specify how the functions act on each pattern in the syntactic domain [SK95]. More than one semantic function may be needed when the syntactic domain is structured or organized by means of more than one domain, for instance, a union of different sub-domains. We define the signatures next; the semantic equations are defined later in (iv).

We define the semantic function $\mathcal{M} : \mathcal{A} \longmapsto \mathcal{R}^*$ as the top-level semantic function that maps any element from the syntactic domain to the semantic domain. The domain of $\mathcal{M}$ is $\mathcal{A}$ defined in (i) as the set of all concepts and concept instances in the practice of architecture description. The codomain of $\mathcal{M}$ is $\mathcal{R}^*$ defined in (ii) as the set of all conceivable model repositories storing modeling artifacts classified by means of the metamodel for megamodels defined in Section §2.3.1. The characterization of the modeling artifacts is specified by means of the assertions defined in (ii). Then, $\mathcal{M}$ defines the interpretation of the conceptual model of the architecture description practice in terms of Model-Driven Engineering constructs, particularly as defined by the Global Model Management approach. As the syntactic domain $\mathcal{A}$ is defined in Table 3.1 as $\mathcal{A}^{\mathsf{C}} \bigcup \mathcal{A}_{\mathsf{c}}$, we define the semantic function $\mathcal{M}$ by means of two auxiliary semantic functions, one for each sub-domain.

We define the semantic function $\mathcal{M}^{\mathsf{C}} : \mathcal{A}^{\mathsf{C}} \longmapsto \mathcal{R}^*$. The domain of $\mathcal{M}^{\mathsf{C}}$ is $\mathcal{A}^{\mathsf{C}}$ defined as the finite set of all *concepts* of the conceptual model of the practice of architecture description. The codomain is $\mathcal{R}^*$ defined as the set of all conceivable model repositories. The denotation of a single concept in $\mathcal{A}^{\mathsf{C}}$ may consist of zero, one or many modeling artifacts. These resulting

Weavings	$\stackrel{\text{def}}{=}$	$\left\{ \begin{array}{l} x \mid x \text{ is a construct of the form } \mathsf{mw}\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle, \\ \quad \text{where } \mathsf{mw} \text{ is an identifier and } \forall \mathrm{i}, 1 \le \mathrm{i} \le n, n \ge 1.\ \mathsf{m_i} :_\bullet \mathsf{Model}, \\ \quad \text{that describes a model weaving relationship between the models } \mathsf{m_i} \end{array} \right\}$

$\mathsf{wm} \blacktriangleright \mathsf{mw}\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle$ for $n \in \mathbb{N}, n \ge 1$

$\stackrel{\text{def}}{=} \mathsf{wm} :_\bullet \mathsf{WeavingModel} \quad \wedge \quad \mathsf{mw}\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle \in \mathit{Weavings} \quad \wedge$

$\mathsf{mw.weavingModel} = \mathsf{wm}$

Transformations	$\stackrel{\text{def}}{=}$	$\left\{ \begin{array}{l} x \mid x \text{ is a construct of the form } \mathsf{t}\langle \overline{x_1}, \ldots, \overline{x_m}, y_1, \ldots, y_n \twoheadrightarrow z_1, \ldots, z_l \rangle, \\ \quad \text{where } \mathsf{t} \text{ is and identifier and } n, m, l \in \mathbb{N} \text{ such that } n + m + l > 0, \\ \quad \forall \mathrm{i}, 0 < \mathrm{i} \le m.\ (x_\mathrm{i} :_\bullet \mathsf{ReferenceModel} \vee x_\mathrm{i} \text{ is an } \mathsf{ExternalEntityKind}) \ \wedge \\ \quad \forall j, 0 < j \le n.\ (y_j :_\bullet \mathsf{ReferenceModel} \vee y_j \text{ is an } \mathsf{ExternalEntityKind}) \ \wedge \\ \quad \forall k, 0 < k \le l.\ (z_k :_\bullet \mathsf{ReferenceModel} \vee z_k \text{ is an } \mathsf{ExternalEntityKind}), \\ \quad \text{that describes a transformation relationship with the} \\ \quad \mathsf{InOut} \text{ parameters } x_\mathrm{i}, \mathsf{In} \text{ parameters } y_j, \text{ and } \mathsf{Out} \text{ parameters } z_k \end{array} \right\}$

$\mathsf{tm} \blacktriangleright \mathsf{t}\langle \overline{x_1}, \ldots, \overline{x_m}, y_1, \ldots, y_n \twoheadrightarrow z_1, \ldots, z_l \rangle$ for $m, n, l \in \mathbb{N}$

$\stackrel{\text{def}}{=} \mathsf{tm} :_\bullet \mathsf{TransformationModel} \quad \wedge$

$\mathsf{t}\langle \overline{x_1}, \ldots, \overline{x_m}, y_1, \ldots, y_n \twoheadrightarrow z_1, \ldots, z_l \rangle \in \mathit{Transformations} \quad \wedge$

$\mathsf{t.transformationModel} = \mathsf{tm}$

Records	$\stackrel{\text{def}}{=}$	$\left\{ \begin{array}{l} x \mid x \text{ is a construct of the form } \mathsf{r} \downarrow \mathsf{t}\langle \mathsf{e}_1, \ldots, \mathsf{e}_n \twoheadrightarrow \mathsf{f}_1, \ldots, \mathsf{f}_l \rangle, \\ \quad \text{where } \mathsf{r} \text{ is an identifier, } \mathsf{t} \in \mathit{Transformations} \text{ and } n, l \in \mathbb{N}, \\ \quad \text{that describes a transformation record that uses} \\ \quad \mathsf{e_i} \text{ as } \mathsf{In} \text{ parameter values and obtains } \mathsf{f}_j \text{ as } \mathsf{Out} \text{ parameter values} \\ \quad \text{when applying } \mathsf{t} \text{ with matching parameter types,} \\ \quad \text{for } 0 < \mathrm{i} \le n, 0 < j \le l, n + l > 0 \end{array} \right\}$
Descriptors	$\stackrel{\text{def}}{=}$	$\mathit{Weavings} \cup \mathit{Transformations} \cup \mathit{Records}$
ν	$\stackrel{\text{def}}{=}$	the selected function of the family $\Theta_{\mathsf{MgMM}}^{\mathcal{R}^* \cup \mathit{Descriptors}}$ that is total, eager and renders the most possibly detailed representation for every modeling artifact and descriptor

Table 3.6: *Semantic domain: descriptors of non-entity elements.*

modeling artifacts provides the basic foundation for mapping instances of the concept. For example, mapping a concept $\mathsf{X} : \circ$ may result in a model repository containing $\mathsf{mmm} :_\bullet \mathsf{Metametamodel}$ and $\mathsf{mm} :_\bullet \mathsf{Metamodel}$ such that $\mathsf{mm} \lhd \mathsf{mmm}$, which are later used in the denotation of $\mathsf{x} :_\circ \mathsf{X}$ that results in $\mathsf{m} :_\bullet \mathsf{TerminalModel}$ such that $\mathsf{m} \lhd \mathsf{mm}$. This is a common scenario that occur in the semantic equations that we define later in (iv).

We define the semantic function $\mathcal{M}_\mathsf{c} : \mathcal{A}_\mathsf{c} \longmapsto \mathcal{R}^*$. The domain of $\mathcal{M}_\mathsf{c}$ is $\mathcal{A}_\mathsf{c}$ defined as the infinite set of all conceivable *instances of concepts* of the conceptual model of the practice of architecture description. The codomain is $\mathcal{R}^*$. The denotation of a single concept instance in $\mathcal{A}_\mathsf{c}$ uses the basic foundation provided by the denotation of the concept of the instance. Then, we have that $\forall \mathsf{x} :_\circ \mathsf{X}.\ \mathcal{M}^\mathsf{C}[\![\mathsf{X}]\!] \subseteq \mathcal{M}_\mathsf{c}[\![\mathsf{x}]\!]$. In addition, the denotation of a concept instance may include other modeling artifacts that are specific to the concept instance being denoted.

$\mathcal{M}^{\mathsf{C}} : \mathcal{A}^{\mathsf{C}} \longmapsto \mathcal{R}^*$	is the semantic function that maps concepts to a model repository
$\mathcal{M}_{\mathsf{c}} : \mathcal{A}_{\mathsf{c}} \longmapsto \mathcal{R}^*$	is the semantic function that maps concept instances to a model repository
$\mathcal{M} : \mathcal{A} \longmapsto \mathcal{R}^*$	is the semantic function that maps elements in the syntactic domain to a model repository, where $\mathcal{M}[\![x]\!] \overset{\text{def}}{=} \begin{cases} \mathcal{M}^{\mathsf{C}}[\![x]\!] & \text{if } x : \mathsf{O} \\ \mathcal{M}_{\mathsf{c}}[\![x]\!] & \text{if } x :_{\circ} \mathsf{X}, \text{for any } \mathsf{X} : \mathsf{O} \end{cases}$
$\mathcal{M}^* : \mathcal{A}^* \longmapsto \mathcal{R}^*$	is the semantic function that maps set of elements in the syntactic domain to a model repository, where $\mathcal{M}^*[\![\{x_1, x_2, \dots, x_n\}]\!] \overset{\text{def}}{=} \bigcup_{\mathrm{i}=1}^{\mathrm{i}=n} \mathcal{M}[\![x_{\mathrm{i}}]\!] \quad \text{for } n \in \mathbb{N}$

Table 3.7: *Semantic functions.*

Then, the semantic function $\mathcal{M}$ distinguishes two cases: it maps concepts according to the semantic function $\mathcal{M}^{\mathsf{C}}$, and it maps concept instances according to the semantic function $\mathcal{M}_{\mathsf{c}}$. Formally, we define $\mathcal{M} : \mathcal{A} \longmapsto \mathcal{R}^*$ as:

$$\mathcal{M}[\![x]\!] \overset{\text{def}}{=} \begin{cases} \mathcal{M}^{\mathsf{C}}[\![x]\!] & \text{if } x : \mathsf{O} \\ \mathcal{M}_{\mathsf{c}}[\![x]\!] & \text{if } x :_{\circ} \mathsf{X}, \text{for any } \mathsf{X} : \mathsf{O} \end{cases}$$

We define the semantic function $\mathcal{M}^* : \mathcal{A}^* \longmapsto \mathcal{R}^*$ that maps finite sets of *concepts* and *concept instances* to model repositories. This function results in the union of mapping each of the elements in the set. In the particular case of an empty set of concept and concept instances, $\mathcal{M}^*$ produces an empty model repository. Formally, we define $\mathcal{M}^*$ as:

$$\mathcal{M}^*[\![\{x_1, x_2, \dots, x_n\}]\!] \overset{\text{def}}{=} \bigcup_{\mathrm{i}=1}^{\mathrm{i}=n} \mathcal{M}[\![x_{\mathrm{i}}]\!] \quad \text{for } n \in \mathbb{N}$$

In order to improve readability, when applying the semantic function to a syntactic element x, we augment the argument to the assertion that characterizes whether the argument is a concept or an instance of a concept. Then, we use $\mathcal{M}[\![\mathsf{X} : \mathsf{O}]\!]$ to denote $\mathcal{M}[\![\mathsf{X}]\!]$ that is equivalent to $\mathcal{M}^{\mathsf{C}}[\![\mathsf{X}]\!]$ as $\mathsf{X} : \mathsf{O}$ holds. We use $\mathcal{M}[\![\mathsf{x} :_{\circ} \mathsf{X}]\!]$ to denote $\mathcal{M}[\![\mathsf{x}]\!]$ that is equivalent to $\mathcal{M}_{\mathsf{c}}[\![\mathsf{x}]\!]$ as $\mathsf{x} :_{\circ} \mathsf{X}$ holds. We are not actually mapping assertions, we are just augmenting the notation to ease the understanding of our specification. Also, in order to avoid cluttering in the definitions, we introduce a shorthand notation for the application of the semantic function. We omit the function name and the curly braces (denoting a set) to denote the application of $\mathcal{M}^*$. For instance, we use $[\![\mathsf{x} :_{\circ} \mathsf{X}]\!]$ as a shorthand for $\mathcal{M}^*[\![\{\mathsf{x}\}]\!]$, that is equivalent to $\mathcal{M}[\![\mathsf{x}]\!]$, and particularly to $\mathcal{M}_{\mathsf{c}}[\![\mathsf{x}]\!]$ as $\mathsf{x} :_{\circ} \mathsf{X}$ holds. As another example, $[\![\mathsf{x} :_{\circ} \mathsf{X}, \mathsf{Y} : \mathsf{O}]\!]$ as a shorthand for $\mathcal{M}^*[\![\{\mathsf{x}, \mathsf{Y}\}]\!]$, that is equivalent to $\mathcal{M}[\![\mathsf{x}]\!] \cup \mathcal{M}[\![\mathsf{Y}]\!]$, and particularly to $\mathcal{M}_{\mathsf{c}}[\![\mathsf{x}]\!] \cup \mathcal{M}^{\mathsf{C}}[\![\mathsf{Y}]\!]$ as $\mathsf{x} :_{\circ} \mathsf{X}$ and $\mathsf{Y} : \mathsf{O}$ hold.

Table 3.7 summarizes these definitions.

Compositionality. The definition of the semantic function is compositional, i.e. the denotational semantics of a syntactic construct is expressed in terms of the denotational semantics

of its sub-constructs. The reason for pursuing compositionality are twofold [SK95]. First, the meaning (denotation) of a construct is defined as the contribution to the meaning of the whole, it is formulated as a function of the denotations of its sub-constructs. As a result, when two constructs have the same denotation, one can be replaced by the other without changing the meaning of the whole, i.e. it supports the substitution of semantically equivalent constructs. Second, since the denotational definition parallels the syntactic structure, properties of constructs in the syntactic domain can be verified by structural induction. The structure allows the individual syntactic constructs to be analyzed and evaluated in relative isolation.

For textual languages in which the abstract syntax is structured as a tree, the compositional definition of the semantic function follows this tree structure. For visual and modeling languages in which the abstract syntax is a graph, compositionality is not straightforward. The denotational semantics of a construct still can be defined in terms of the denotation of its connected constructs. However, having cycles in the definition not only prevent proofs by structural induction, but also it complicates any potential automation of the semantics due to infinite recursion. In order to avoid or reduce the impact of having a graph structuring the syntactic domain, we can decide a traversal on the graph that renders a directed acyclic graph. Although the rendered graph is not a tree, it still present compositionality, as we avoid circularity in the definition of the semantic equations.

(iv) Semantic equations

In the application of the denotational semantics approach, semantic functions are defined by means of semantic equations or clauses [SK95]. A semantic function is well-defined if for every element in the syntactic domain, there is a semantic equation that specifies how such an element is denoted in the semantic domain. A semantic equation may specify the denotation of a single syntactic element, or a category of elements of the same form.

In (iii) we formally defined the semantic function $\mathcal{M}^*$ on set of syntactic elements by means of the semantic function $\mathcal{M}$. In turn, $\mathcal{M}$ was formally defined by means of the semantic functions $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$. These two functions were not defined in (iii). Then, the semantic equations that we need to define fall in two cases: (a) the specification of how a particular concept $\mathsf{X} : \mathsf{o}$ in $\mathcal{A}^{\mathsf{C}}$ is mapped to a set of modeling artifacts, and (b) how a particular instance $\mathsf{x} :_{\mathsf{o}} \mathsf{X}$ is mapped. Cases (a) and (b) correspond to the equations of $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$, respectively, and correspond also to the two cases of the definition of the semantic function $\mathcal{M}$. A semantic equation has the general form:

$$[\![x]\!] \stackrel{\mathsf{def}}{=} \mathsf{R} \text{ such that } \psi$$

where x is a concept or concept instance, and its denotational semantics is a set of modeling artifacts R on which the assertion ψ holds. It is important to notice that we use the shorthand notation for the application of the semantic functions, as we explained in (iii). Whether the equation corresponds to $\mathcal{M}^{\mathsf{C}}$ or $\mathcal{M}_{\mathsf{c}}$ can be inferred from the kind of the element x.

As the semantic functions are compositional, the semantic equations may recursively use the semantic functions on the sub-constructs of the construct being defined. In the case that the construct x being mapped is terminal, i.e. its semantics is not defined in terms of any sub-construct, the set of modeling artifacts R is expressed by extension as $\{\mathsf{a_1}, \ldots, \mathsf{a_k}\}$ where $k \geq 0$. The case $k = 0$ is the case where the construct is not mapped to any modeling artifact, i.e. it is mapped to the empty set of modeling artifacts. In the case that the construct x being mapped is non-terminal, i.e. its semantics is defined in terms of the semantics of its sub-constructs, the set of modeling artifacts R takes the form of the union of the sets resulting from applying the function to the sub-constructs with (a possibly empty) set of modeling artifacts that is the contribution of the construct by itself. Then, the refined general form for a semantic clause is:

$$[\![x]\!] \stackrel{\text{def}}{=} [\![x_1]\!] \ \cup \ \ldots \ \cup \ [\![x_n]\!] \ \cup \ \{\mathsf{a_1}, \ldots, \mathsf{a_k}\} \text{ such that } \psi$$

where $n, k \geq 0$, being the case $n = 0$ for terminal constructs and $n > 0$ for non-terminal, and the case $k = 0$ for constructs that do not contribute any modeling artifact by themselves. Each x_i is not necessarily a single concept or concept instance, it can actually consist of a set as $x_\mathrm{i} \in \mathcal{A}^*$.

The assertion ψ is a conjunction of assertions on the modeling artifacts conforming the resulting model repository. As we explained in (ii) when we defined the semantic domain for the semantic functions, there are several kinds of assertions that allow the characterization of the modeling artifacts in terms of the Global Model Management approach. In the specification of every semantic equation, we use an assertion for each resulting modeling artifact to state its kind; these assertions take the form $\mathsf{a} :_\bullet \mathsf{A}$ as defined in Table 3.2. In order to enhance the readability of the equations, we embed this particular kind of assertions in the enumeration of modeling artifacts in the result. The additional assertions that may be required are stated as part of the assertion ψ. Then, the refined general form for a semantic clause is:

$$\begin{aligned}
[\![x]\!] \stackrel{\text{def}}{=} \ & [\![x_1]\!] \ \cup \ \ldots \ \cup \ [\![x_n]\!] \ \cup \\
& \{\mathsf{a}_1 :_\bullet \mathsf{A}_1, \\
& \ \ \ldots, \\
& \ \ \mathsf{a}_k :_\bullet \mathsf{A}_k\} \text{ such that} \\
& \psi_1 \ \wedge \\
& \ldots \ \wedge \\
& \psi_l
\end{aligned}$$

where $n, k, l \geq 0$. The assertion specifies that the *concept* or *concept instance* x (depending on whether $[\![x]\!]$ has the form $[\![\mathsf{X} : \circ]\!]$ or $[\![\mathsf{x} :_\circ \mathsf{X}]\!]$) is denoted by the modeling artifacts rendered by the sub-constructs $x_1, \ldots, x_n$ of x, and the modeling artifacts $\mathsf{a}_1, \ldots, \mathsf{a}_k$ of kind $\mathsf{A}_1, \ldots, \mathsf{A}_k$ respectively, on which the assertion $(\psi_1 \wedge \ldots \wedge \psi_l)$ holds.

Customization of the equations. The goal of our model-based interpretation is to provide a precise specification of which modeling artifacts must be built to represent architecture knowledge expressed in terms of the conceptual model on the architecture description prac-

tice, their properties and relationships. However, the goal of the interpretation is neither to define nor restrict how these resulting modeling artifacts must be actually built. Each semantic equation states that a concept or concept instance (in the syntactic domain) is represented by a set of modeling artifacts (in the semantic domain) on which certain assertions hold, but it does not state how these modeling artifacts must be defined. We specify the semantic equations in a way that they are not dependent on any particular architecture framework, architecture description language, practitioners' skills, or description or design method. Our goal is to allow the representation of any of these techniques that are part of the architecture knowledge in terms of our semantic domain by guiding what actually needs to be built, leaving the how to the actual intention of the architect, team or community.

We encode the intention on how to build, define or develop particular modeling artifacts by means of a *customization* function denoted by ρ. When appropriately, the specification of the semantic equations applies the customization function to specify that the practitioner must decide how to develop a particular modeling artifact. The semantic equation states (forces) that the modeling artifact exist in the resulting model repository, but it is the practitioner's responsibility to determine how to build the modeling artifact. Then, each application of the customization function ρ acts as a placeholder in the semantic equations for the practitioner to fill in.

The customization function ρ is parameterized on the kind of modeling artifact that must be built. This kind is any concrete sub-metaclass of Entity, i.e. an element of $\mathcal{R}^{\mathsf{C}}$ defined in Table 3.2. The customization function is also parameterized on an identifier that allows to distinguish applications of the function from each other. We define *Identifiers* as the set of all conceivable identifiers that might be required or used to identify and distinguish from each other every placeholder in the application of the semantic function to a particular set of elements of the syntactic domain. Each identifier in *Identifiers* can take any form as it is convenient to capture the intuition on the purpose of the placeholder. For instance, an identifier might be a concept in $\mathcal{A}^{\mathsf{C}}$, a concept instance in $\mathcal{A}_{\mathsf{c}}$, a tuple of elements, an arbitrary name, or any combination that fits the purpose. The result of applying the customization function is a single modeling artifact. Then, the signature of the customization function is $\rho : \mathcal{R}^{\mathsf{C}} \times \mathit{Identifiers} \rightharpoonup \mathcal{R}$. The function ρ is partial as not every combination of metaclass and identifier must be defined by the practitioner. The function is well-defined when for all the combinations that are actually required by the denotation of a particular set of elements of the syntactic domain, we have that $\rho(\mathsf{X}, \mathrm{id}) :_{\bullet} \mathsf{X}$, with $\mathsf{X} \in \mathcal{R}^{\mathsf{C}}, \mathrm{id} \in \mathit{Identifiers}$.

When applying the customization function, we use the metaclass in the subscript as a shorthand notation. Then, we use $\rho_{\mathsf{X}}(\mathrm{id})$ as a shorthand for $\rho(\mathsf{X}, \mathrm{id})$. Whenever the metaclass of the resulting modeling artifact can be inferred from the context, we omit the metaclass and uses the function name as a subscript for the identifier. Then, provided that we know that the placeholder is of a given kind X, we use id_{ρ} as a shorthand for $\rho(\mathsf{X}, \mathrm{id})$. This is a recurring scenario in the semantic equations as the resulting modeling artifact is directly stated in an assertion determining its kind, i.e. we have assertions of the form $\mathrm{id}_{\rho} :_{\bullet} \mathsf{X}$, and, in this case, the kind can be directly inferred from the assertion itself.

Our model-based interpretation does not provide the clauses for defining the customization function ρ, these clauses are to be defined by the architect, the development team

or the practitioner community; this is actually the purpose of customization functions. It is important to notice that the *customization* function is not a *representation-of* function. Representation-of functions yields model elements within a model fragment, not modeling artifacts. The *customization* function is actually a kind of semantic function as it maps identifiers (in the *Identifiers* syntactic domain) to a modeling artifact in the semantic domain. In contrast to the semantic function $\mathcal{M}$ that deals with the external properties and relationships of the modeling artifacts, the customization function ρ deals with the definitions that are internal to the modeling artifact.

3.2.2 Semantics of Architecture Descriptions

We define the semantics of software architecture descriptions by providing the semantic equations or clauses of the semantic functions defined Section §3.2.1. We define the equations for those concepts of the conceptual model that are directly involved with the ArchitectureDescription concept. We reviewed these concepts in Section §3.1.1 and we illustrated them in Figure 3.2. The semantic equations coping with architecture frameworks and architecture description languages are presented later in Section §3.2.3.

In order to define the semantic equations in a way that they preserve the compositionality of the semantic function, we decide a traversal on the concepts of the conceptual model that yields a directed acyclic graph. In Figure 3.6 we superpose arrows on associations to illustrate this traversal. Even though the concept SystemOfInterest is the only node that has no inbound arrows, it is not meant as the root node of the traversal. The application of the semantic function can be actually applied to any concept directly disregarding all those concepts or concept paths that directs to the concept. For instance, the semantic function can be applied directly to ArchitectureDescription without even considering the concepts Architecture or SystemOfInterest. Outbound arrows are those that impose compositionality, i.e. when mapping any concept X of the conceptual model, the denotation of every concept that can be reached from X following the arrows are also considered in the denotation.

We define the semantic equations for the semantic functions $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$, dealing with concepts and concept instances of the conceptual model, respectively. As we explained in Section §3.2.1 (iii), the semantic function $\mathcal{M}$ is defined by case analysis on the syntactic element to denote, being $\mathsf{X} : \mathsf{O}$ and $\mathsf{x} :_{\circ} \mathsf{X}$ the two cases. We decide the same traversal of the conceptual model for both cases. However, in the case of the semantic function $\mathcal{M}_{\mathsf{c}}$ for concept instances, the composition also traverse to the concept level. In other words, every semantic equation for a concept instance $\mathsf{x} :_{\circ} \mathsf{X}$ is compositionally defined in terms of the semantic equations for the concept instances that can be reached from x following the traversal of the graph <u>and</u> in terms of the semantic equation for the its concept X.

Given that the set of concepts $\mathcal{A}^{\mathsf{C}}$ is finite, it can be defined by extension. Hence, we define a semantic rule for each of the concepts in the conceptual model. However, the set of concept instances $\mathcal{A}_{\mathsf{c}}$ is infinite; there are infinite concept instances of the concepts in the conceptual model. In this case, the semantic function cannot be defined by extension. We define a semantic equation for each kind of concept instance. The set of resulting modeling artifacts from a concept instance $\mathsf{x} :_{\circ} \mathsf{X}$ are indexed by x, in order to allow us to distinguish

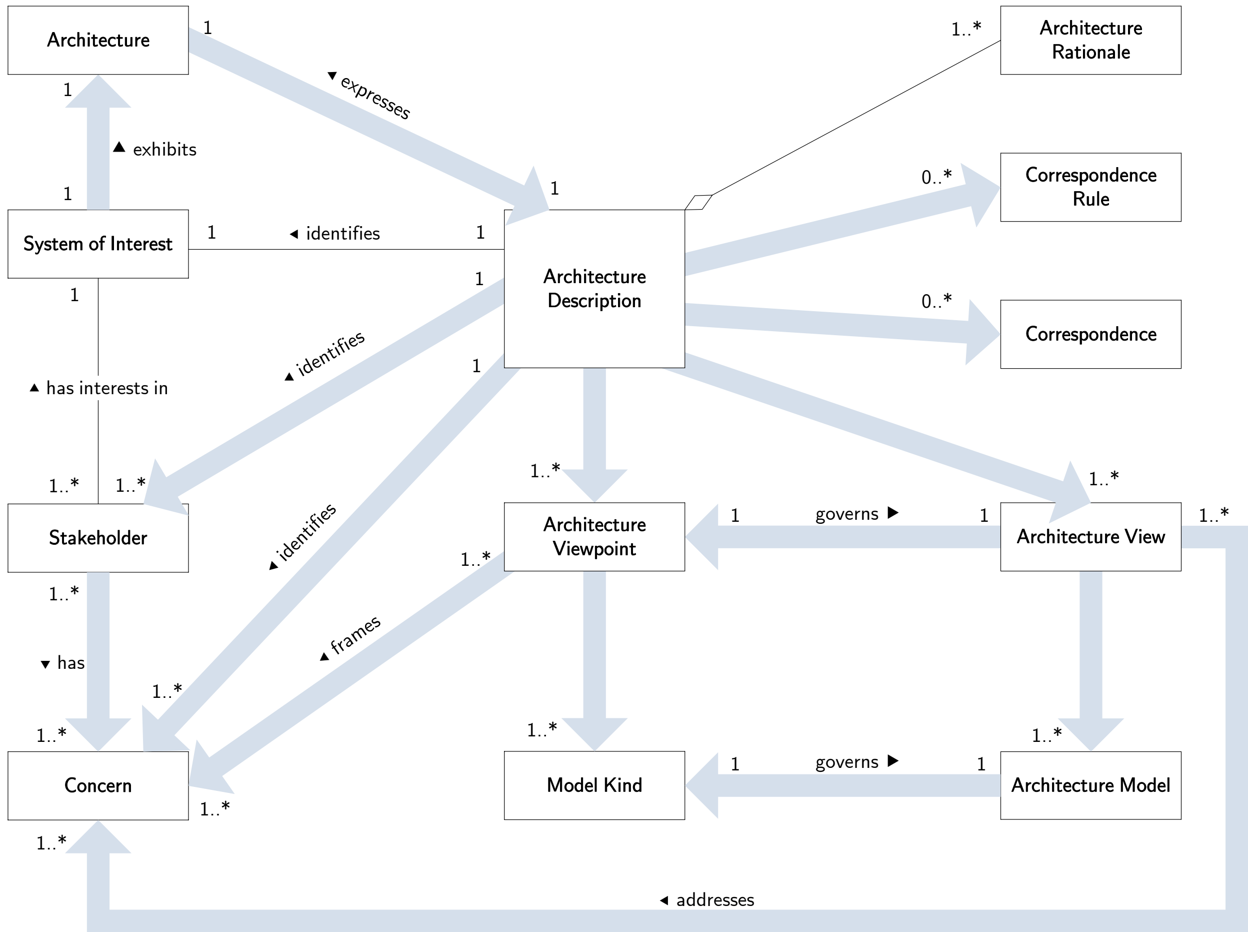


Figure 3.6: *Conceptual model traversal for semantic function compositionality.*
The figures illustrates how the conceptual model for architecture descriptions is traversed by the semantic equations of the semantic function $\mathcal{M}$ in order to yield a directed acyclic graph. The traversal is used to provide compositionality in the definition of the semantic equations. The underlying conceptual model is presented in Figure 3.2.

modeling artifacts corresponding to different instances of the same concept.

We define the equations for concepts and concept instances in what follows. We use a bottom-up approach to present the equations. We follow the inverse order of compositionality, starting with the leaves of the directed acyclic graph, upwards through the composition paths.

Concerns

In the Software Engineering discipline, *concerns* refer to any significant topic of interest pertaining the system. Concerns can manifest as stakeholders needs, expectations, goals, system responsibilities, requirements, design constraints, assumptions, dependencies, quality attributes, pre-existing architecture or design decisions and risks [ISO11]. In the practice of architecture description, architecturally significant concerns drive the architecture design. The architecture description must identify and capture these concerns, defining where (in which views) and how (structure, behavior and rationale) they are addressed. Also, con-

cerns are used to classify and characterize reusable architectural assets such as architecture viewpoints, model kinds, architecture frameworks and architect description languages, as we explained in Sections §3.1.1 and §3.1.2.

Provided the tight relationship and impact of concerns on other concepts of the architecture description practice, concerns tend to be captured together with the description of those concepts. For instance, the standard provides a template for guiding architecture viewpoint documentation in [ISO11, Annex B], which includes an specific section for listing the concerns that are framed by the viewpoint. Whether the concerns are recorded in a purpose-specific artifact or as a part (section) of other artifacts might be regarded as a subjective matter that is finally up to the architect team to decide. We argue that embedding the description of concerns in several artifacts bring some problems. First, it may imply a certain level of redundancy if practitioners intend to describe the concerns or to capture their interrelationships. Concern definitions end up tangled with other definitions and scattered across several artifacts. Second, it may introduce inconsistencies in the set of artifacts due to poor choice of names, typos, or unperformed updates. Third, while the traceability from artifacts to concerns would be explicitly captured, to trace from a given concern to the related artifacts requires to span over the whole set of artifacts. We claim that it is important to separate (a) the information pertaining concerns from (b) the information pertaining the relation of the concerns to any given artifact. Case (a) might include a definition, a set of properties, catalog information such as a refinement hierarchy of concerns or a classification in categories, and interrelationships such as one favoring or constraining another, among others. Case (b) might include their feasibility, their prioritization, whether they are framed or cannot be framed (anti-concerns), among others. In our semantic specification, we follow the strategy of using a single purpose-specific model. We take advantage of the inherent transverse and reusable characteristic of concerns by applying modularization and encapsulation. We are inclined to take this modularization approach for all concepts whenever it is possible and appropriate.

Our semantic specification defines a separate *modeling artifact*, a *concerns model*, to capture any information in case (a) in the particular working context. This model provides a coherent and cohesive medium for concern specification. We do not impose any particular internal structure for representing concerns; it is up to customization how the concerns are actually represented. Then, for each architecture description there is a single terminal model representing the concerns that are significant for the architecture description. We proceed analogously for each architecture framework and architecture description language, as we discuss later in Section §3.2.3. The relationship between concerns and other concepts such as stakeholders, architecture viewpoints and model kinds, are captured as model annotations of this single concerns model. Model annotations takes the form of model weavings of arity 1.

Provided that reusable architectural assets are independently defined and packaged, we need a mechanism for combine concerns models. For instance, the architect's *decision* of using a given architecture framework in the architecture description the architect is working on, requires all the concerns in the architecture framework to be merged into the concerns model of the architecture description. We use this mechanism later in Section §4.2.3 when we formally specify this kind of architecture decision. Provided that concerns are captured in an separate model, this mechanism can be reified by a model transformation that takes two concerns models and merge the definitions in the second into the first one.

$$
\begin{array}{rl}
1 & [\![\mathsf{Concern} : \circ]\!] \stackrel{\text{def}}{=} \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \\
2 & \quad \mathsf{CMM}_\rho :_\bullet \mathsf{Metamodel}, \\
3 & \quad \mathsf{CM} :_\bullet \mathsf{TerminalModel}, \\
4 & \quad \mathsf{TMM}_\rho :_\bullet \mathsf{Metamodel}, \\
5 & \quad \mathsf{CMergeTM} :_\bullet \mathsf{TransformationModel}\} \text{ such that} \\
6 & \quad \mathsf{CM} \lhd \mathsf{CMM}_\rho \ \wedge \ \mathsf{CMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge \\
7 & \quad \mathsf{CMergeTM}_\rho \lhd \mathsf{TMM}_\rho \ \wedge \ \mathsf{TMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge \\
8 & \quad \mathsf{CMergeTM}_\rho \blacktriangleright \mathsf{CMergeT} \left\langle \overline{\mathsf{CMM}_\rho}, \mathsf{CMM}_\rho \twoheadrightarrow \varnothing \right\rangle \\
9 & [\![\mathsf{c} :_\circ \mathsf{Concern}]\!] \stackrel{\text{def}}{=} [\![\mathsf{Concern} : \circ]\!] \text{ such that} \\
10 & \quad \mathsf{c}_\mu \lesssim \mathsf{CM}
\end{array}
$$

Table 3.8: *Semantic equations:* Concern.

Table 3.8 defines the semantic equations for the concept Concern (1) and its instances (9). The concept Concern is denoted by four modeling artifacts that provide the foundation for representing concerns in architecture descriptions.

The metametamodel MMM_ρ (1) is the metametamodel that determines the particular modeling technical space that is used in the specification. In order to provide the practitioner the ability to decide the technical space, we use the *customization* function ρ to specify the placeholder for the particular metametamodel. The same placeholder is used along all the semantic equations, establishing that the same technical space is to be used in the particular context the practitioners are working. Then, the metametamodel MMM_ρ defines the modeling language that is used to express the metamodels.

The metamodel CMM_ρ (2) is the customized metamodel defining the modeling language to build models that capture and document concerns. By using a customized metamodel, our specification enables practitioners to define the language of their convenience. The metamodel must be expressed in terms of the metametamodel MMM_ρ (6). The customized metamodel is responsible of identifying which information and interrelations on concerns is interesting in the working context or in the practitioners community.

The terminal model CM (3) is the model that actually captures and preserves the representation of the concerns of interest in the working context. The terminal model is not customized, it is a specific model that conforms to the customized metamodel CMM_ρ (6). Weaving models for annotating the concerns model CM are not part of the denotation of the concept Concern or its instances. These weaving models are part of the denotation of those concepts or concept instances that are related to concerns.

The purpose of the transformation CMergeT (8) is to merge into a concerns model the definitions of a second concern model. The type of both parameters is the metamodel CMM_ρ, being the first parameter InOut as its corresponding parameter value is the one that is updated by the transformation. The transformation CMergeT is realized by the transformation model $\mathsf{CMergeTM}_\rho$ (8). The transformation model is implemented (expressed) in the model transformation language defined by the metamodel TMM_ρ (7). We use a customized metamodel TMM_ρ (4) as a placeholder for the desired model transformation language, and we use

the same metamodel across all semantic equations. This is actually a simplification. Different model transformation languages can be used by the practitioners for developing different transformation models. To support this variability would require us solely to use different identifiers when naming these metamodels. However, we prefer to use the same identifier in order to favor readability and understanding of the specification.

The denotation of every $\mathsf{c} :_\circ \mathsf{Concern}$ results in the set of foundational modeling artifacts that are provided by the denotation of the Concern concept itself. However, provided that the concern c is part of a composing construct, it must be captured and documented. A representation of the concern c is then recorded in the concerns model CM (10). However, how this representation is actually built depends on the metamodel CMM_ρ. We use c_μ to represent the application of the *representation-of* function $\mu^{\mathsf{Concern}}_{\mathsf{CMM}_\rho}$ to the concern c. c_μ is a model fragment conforming to CMM_ρ. The semantic equation for mapping a concern c results in the same modeling artifacts as mapping the concept Concern, but it states that the resulting concern model CM actually defines the model elements corresponding to the model fragment c_μ.

Stakeholders

The *stakeholders* of a system are parties with interests or concerns in that system. In the architecture description practice, stakeholders having architecturally significant concerns must be identified and recorded. Knowing the stakeholders provides contextualization for concerns, and allows the architect team to keep track of whom to fallback in case of renegotiation when conflicting concerns must be dealt with. Trade-off analysis to make architecture decisions may lead to this kind of renegotiation. As we explained in Sections §3.1.1 and §3.1.2, concerns and stakeholder together are used to classify and characterize reusable architectural assets such as architecture viewpoints, model kinds, architecture frameworks and architect description languages.

As is the case for concerns, stakeholders have also a tight relationship and impact on other concepts of the architecture description practice. Then, whether stakeholders are captured in a purpose-specific artifact or as a part of other artifacts is also a subjective matter. We employ the same arguments as for concerns to argue that (a) the information pertaining stakeholders must be separated from (b) the information pertaining the relationship between stakeholders and any other artifacts. In practice, stakeholders are significant to the architecture when their concerns are. Generally, information in case (b) involves a list of stakeholders that is used to characterize and classify concerns. Then, provided that the architect capture both the relationship between artifacts and concerns, and the relationship between concerns and stakeholders, the relationship between artifacts and stakeholders can be derived by transitivity from those two relationships.

The relationship between concerns and stakeholders is represented in the conceptual model by the many-to-many *has* association, as illustrated in Figure 3.2. In addition to capturing concerns and stakeholders, practitioners must capture the relationship between them. There are several alternatives to this end: a single artifact capturing everything together, two artifacts capturing concerns in one and stakeholders and the relationship in another (or

vice versa), or three separate artifacts capturing them independently. Both concerns and stakeholders are at the same level of abstraction and they are used in combination, as it is illustrated in Figures 3.2, 3.4 and 3.5 for the cases of architecture descriptions, architecture frameworks, and architecture description languages, respectively. Even though architecture viewpoints, model kinds and architecture views are associated to concerns but not to stakeholders in the conceptual model, it is either an omission in the standard, or it is expected to be derived by transitivity from their relationship to concerns. Then, a single artifact is well suited for the purpose. However, we argue that capturing everything together does not favors separation-of-concerns – in the sense introduced by E. Dijkstra in [Dij74]. Also, the lack of modularization and encapsulation when using a single artifact does not take into account the possible evolution of the conceptual model to cases in which concerns or stakeholders are actually of interest independently of each other. Then, in our semantic specification we follow the strategy of using three separate artifacts. We discard the case of two artifacts as it would favor modularization of one of the concepts, but not both. It is important to notice, however, that the greater the number of artifacts the greater the effort to manage them. Also, even though the information to be captured is actually the same in all cases, the practitioners skills and the tool support required in the case of multiple artifacts is higher. We consider the other cases as a variation of our semantic specification, and although not specified in our semantic equations, they can be formally defined following the same technique as ours.

Our semantic specification defines a *stakeholders model* to capture stakeholders' specific information, and a *stakeholders-concerns weaving model* to capture their relationship. The *concerns model* is introduced by the semantic equation of $\mathsf{Concern} : \circ$. As is the case for concerns, we also need a transformation to merge into the models under development the information captured in reusable architectural assets such as architecture frameworks and architecture description languages.

Table 3.9 defines the semantic equations for the concept $\mathsf{Stakeholder}$ (1) and its instances (15). The denotation of $\mathsf{Stakeholder} : \circ$ includes the denotation of $\mathsf{Concern} : \circ$ (2), according to the compositional traversal of the conceptual model illustrated in Figure 3.6, and a set of modeling artifacts (3–9) specifically required to provide the foundation for representing *stakeholders* in architecture descriptions. The metametamodel MMM_ρ (3) determines the technical space used in the specification. The metamodel SMM_ρ is the customized metamodel defining the modeling language to capture and document stakeholder-specific information. SMM_ρ conforms to the metametamodel MMM_ρ (10). The terminal model SM (5) is the model that actually captures and preserves the representation of the stakeholders in the working context. SM is not customized and it conforms to SMM_ρ (10). SCWMM_ρ (6) is the metamodel that defines the modeling language for capturing relationships between stakeholders and concerns. SCWMM_ρ is customized it depends on the architect intention which information need to be captured. The metamodel can simply establish links between stakeholders and concerns (pairs), or it can define a more complex modeling language that enables the architect to capture relevant information on the links. The weaving model SCWM (7) is the weaving model that actually captures the many-to-many relationship between stakeholders in SM and concerns in CM, expressed in terms of the metamodel SCWMM_ρ (11). The weaving model realizes the model weaving SCMW. The model CM is part of the resulting model repository as it is produced by the denotation of $\mathsf{Concern} : \circ$ as defined in Table 3.8. As is the case for *concerns*, a transformation $\mathsf{SMergeT}$ (14) is needed to merge into a working set of models the

$$
\begin{array}{rl}
1 & [\![\mathsf{Stakeholder} : \circ]\!] \\
2 & \quad \stackrel{\text{def}}{=} [\![\mathsf{Concern} : \circ]\!] \;\cup \\
3 & \qquad \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \\
4 & \qquad\; \mathsf{SMM}_\rho :_\bullet \mathsf{Metamodel}, \\
5 & \qquad\; \mathsf{SM} :_\bullet \mathsf{TerminalModel}, \\
6 & \qquad\; \mathsf{SCWMM}_\rho :_\bullet \mathsf{Metamodel}, \\
7 & \qquad\; \mathsf{SCWM} :_\bullet \mathsf{WeavingModel}, \\
8 & \qquad\; \mathsf{TMM}_\rho :_\bullet \mathsf{Metamodel}, \\
9 & \qquad\; \mathsf{SMergeTM} :_\bullet \mathsf{TransformationModel}\} \text{ such that} \\
10 & \qquad \mathsf{SM} \triangleleft \mathsf{SMM}_\rho \;\wedge\; \mathsf{SMM}_\rho \triangleleft \mathsf{MMM}_\rho \;\wedge \\
11 & \qquad \mathsf{SCWM} \triangleleft \mathsf{SCWMM}_\rho \;\wedge\; \mathsf{SCWMM}_\rho \triangleleft \mathsf{MMM}_\rho \;\wedge \\
12 & \qquad \mathsf{SCWM} \blacktriangleright \mathsf{SCMW}\langle \mathsf{SM}, \mathsf{CM}\rangle \text{ with } \mathsf{CM} \in [\![\mathsf{Concern} : \circ]\!] \;\wedge \\
13 & \qquad \mathsf{SMergeTM}_\rho \triangleleft \mathsf{TMM}_\rho \;\wedge\; \mathsf{TMM}_\rho \triangleleft \mathsf{MMM}_\rho \;\wedge \\
14 & \qquad \mathsf{SMergeTM}_\rho \blacktriangleright \mathsf{SMergeT}\left\langle \overline{\mathsf{SMM}_\rho}, \overline{\mathsf{CMM}_\rho}, \overline{\mathsf{SCWMM}_\rho}, \mathsf{SMM}_\rho, \mathsf{CMM}_\rho, \mathsf{SCWMM}_\rho \twoheadrightarrow \varnothing \right\rangle \\
15 & [\![\mathsf{s} :_\circ \mathsf{Stakeholder}]\!] \\
16 & \quad \stackrel{\text{def}}{=} [\![\mathsf{Stakeholder} : \circ]\!] \;\cup\; [\![\mathsf{s.concerns}]\!] \text{ such that} \\
17 & \qquad \mathsf{s}_\mu \lesssim \mathsf{SM} \;\wedge \\
18 & \qquad \forall \mathsf{c} \in \mathsf{s.concerns}.\; \langle \mathsf{s}, \mathsf{c}\rangle_\mu \lesssim \mathsf{SCWM}
\end{array}
$$

Table 3.9: *Semantic equations:* Stakeholder.

definitions of a second set of models. The transformation updates a stakeholders model conforming to SMM_ρ, a concerns model conforming to CMM_ρ and a weaving model conforming to SCWMM_ρ, from a set of models conforming to these same metamodels. The transformation SMergeT is realized by a transformation model $\mathsf{SMergeTM}_\rho$ (13) that encapsulate the knowledge of how the working models on stakeholders, concerns and their relationships are updated when including a set of definitions already taken in a separate set of models. It is expected that the transformation model $\mathsf{SMergeTM}_\rho$ is defined as a composite transformation that uses $\mathsf{CMergeTM}_\rho$ defined in Table 3.8–(5) to combine concerns.

The denotation of every $\mathsf{s} :_\circ \mathsf{Stakeholder}$ (15) results in the set of foundational modeling artifacts provided by the denotation of $\mathsf{Stakeholder} : \circ$ (16). Additionally, provided the compositional traversal we decided (see Figure 3.6), the denotation of s also includes the denotation of every concern c related to s, i.e. every $\mathsf{c} \in \mathsf{s.concerns}$ (16). Although no additional modeling artifact is required to denote $\mathsf{s} :_\circ \mathsf{Stakeholder}$, additional assertions must hold on the resulting artifacts. The representation of s in terms of the modeling language defined by SMM_ρ must be defined in the stakeholders model SM (17). Also, for each related concern c, the relationship between s and c must be captured in the weaving model SCWM (18) according to the language defined by the metamodel SCWMM_ρ. As stated in (18), given a $\mathsf{c} :_\circ \mathsf{Concern}$ with $\mathsf{c} \in \mathsf{s.concerns}$, the customized representation of the relationship $\langle \mathsf{s}, \mathsf{c}\rangle$ is defined in the weaving model SCWM.

Model kinds

A *model kind* specifies the conventions for building a specific kind of *architecture models* and it is conceived as a system-independent reusable asset to be shared across different architecture descriptions. The ISO/IEC/IEEE 42010:2011 standard provides an informative guide in [ISO11, Annex B] on how to capture and document *architecture viewpoints* in a form that it meets the standard requirements. Provided that an architecture viewpoint is an aggregation of model kinds, this guide explains in [ISO11, Section B.2.6] how to capture and document *model kinds*. Although this guide is not normative but informative, it still provides a thorough basis on how the practitioner community is expected or promoted to document model kinds. The standard suggests to use one or a combination of the following methods:

(a) by specifying a metamodel that defines the architecture models' core constructs,
(b) by providing a model template to be filled by architects, or
(c) via a language definition or by reference to an existing modeling language.

In the context of our model-based interpretation, we consider that case (3.2.2) must be mandatory. Metamodeling is the core of Model-Driven Engineering techniques: not only a metamodel provides the specification of a modeling language in which models are expressed, but also a metamodel enables the development of model transformations to automate querying, processing and updating models. However, basing our specification only on case (3.2.2) would not be enough for capturing actual architecture knowledge regarding model kinds. The goal of our specification is to provide support to any combination of the alternatives to capture model kinds. As a consequence, the semantic equations for model kinds result in a set of modeling artifacts in which some of them are optional. Whether to produce the optional modeling artifacts in the denotation of a specific $\mathsf{mk} :_{\circ} \mathsf{ModelKind}$ is decided by the practitioner community generating the model-based specification of the model kind.

Table 3.10 defines the semantic equations for the concept $\mathsf{ModelKind}$ (1) and its instances (14). The denotation of $\mathsf{ModelKind} : \circ$ includes the metametamodel MMM_{ρ} (3) that determines the technical space for all metamodels. It also includes the metamodel TMM_{ρ} (4) for transformation models as the specification of the model transformation language to use. The denotation of $\mathsf{mk} :_{\circ} \mathsf{ModelKind}$ is compositional on the denotation of $\mathsf{ModelKind} : \circ$ (15) which provides the foundational modeling artifacts for the modeling artifacts of any model kind. The rest of the resulting modeling artifacts we define are those required to capture any combination of the documentation methods for model kinds, as guided by the standard. In what follows, we discuss the semantic equations for the concept and its instances at the same time, and we structure the discussion in different aspects that, altogether, cope with the subsections of [ISO11, Section B.2.6].

Concerns. Although the standard does not state an explicit association between *model kinds* and *concerns* in the conceptual models for architecture descriptions and architecture frameworks, as we illustrated in Figures 3.2 and 3.4 respectively, it does consider such an association in the context of architecture description languages as illustrated in Figure 3.5. We claim that the association is useful and should be used in practice. As model kinds are conceived as reusable architectural assets, they are subject to classification in terms of

1 $[\![\mathsf{ModelKind} : \circ]\!]$

2 $\overset{\text{def}}{=} [\![\mathsf{Concern} : \circ]\!] \;\; \cup$

3 $\{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$

4 $\mathsf{TMM}_\rho :_\bullet \mathsf{Metamodel},$

5 $\mathsf{ModelKindCWMM}_\rho :_\bullet \mathsf{Metamodel},$

6 $\mathsf{ModelKindDocMM}_\rho :_\bullet \mathsf{Metamodel},$

7 $\mathsf{ProblemMM}_\rho :_\bullet \mathsf{Metamodel},$

8 $\mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that

9 $\mathsf{TMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge$

10 $\mathsf{ModelKindCWMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge$

11 $\mathsf{ModelKindDocMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge$

12 $\mathsf{ProblemMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge$

13 $\mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho$

14 $[\![\mathsf{mk} :_\circ \mathsf{ModelKind}]\!]$

15 $\overset{\text{def}}{=} [\![\mathsf{ModelKind} : \circ]\!] \;\; \cup \;\; [\![\mathsf{mk.concerns}]\!] \;\; \cup$

16 $\{\mathsf{mkCWM} :_\bullet \mathsf{WeavingModel},$

17 $\mathsf{mkDocM} :_\bullet \mathsf{TerminalModel},$

18 $\mathsf{mkMM}_\rho :_\bullet \mathsf{Metamodel},$

19 $\mathsf{mkcTM}^{\mathsf{i}}_\rho :_\bullet \mathsf{TransformationModel},$

20 $\mathsf{mktPlaceholderMM}^{j}_\rho :_\bullet \mathsf{Metamodel},$

21 $\mathsf{mktTM}^{j}_\rho :_\bullet \mathsf{TransformationModel},$

22 $\mathsf{mkMgM} :_\bullet \mathsf{Megamodel}\}$

23 for $1 \le \mathrm{i} \le n_c, 0 \le n_c, 1 \le j \le n_t, 0 \le n_t$ such that

24 $\mathsf{mkCWM} \lhd \mathsf{ModelKindCWMM}_\rho \;\; \wedge \;\; \mathsf{mkCWM} \blacktriangleright \mathsf{mkCMW}\,\langle \mathsf{CM} \rangle \;\; \wedge$

25 $\forall \mathsf{c} \in \mathsf{mk.concerns}.\;\; \langle \mathsf{mk}, \mathsf{c} \rangle_\mu \lesssim \mathsf{mkCWM} \;\; \wedge$

26 $\mathsf{mkDocM} \lhd \mathsf{ModelKindDocMM}_\rho \;\; \wedge \;\; \mathsf{mk}_\mu \lesssim \mathsf{mkDocM} \;\; \wedge$

27 $\mathsf{mkMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge$

28 $\mathsf{mkcTM}^{\mathsf{i}}_\rho \lhd \mathsf{TMM}_\rho \;\; \wedge \;\; \mathsf{mkcTM}^{\mathsf{i}}_\rho \blacktriangleright \mathsf{mkcMT}^{\mathsf{i}}\,\langle \mathsf{mkMM}_\rho \twoheadrightarrow \mathsf{ProblemMM}_\rho \rangle \;\; \wedge$

29 $\mathsf{mkcsEEK}$ is an $\mathsf{ExternalEntityKind} \;\; \wedge \;\; \mathsf{mkcsEEK} \;\varepsilon\; \mathsf{mkMgM} \;\; \wedge$

30 $\nu(\mathsf{mkcsIT}\,\langle \mathsf{mkcsEEK} \twoheadrightarrow \mathsf{mkMM}_\rho \rangle) \lesssim \mathsf{mkMgM} \;\; \wedge$

31 $\nu(\mathsf{mkcsET}\,\langle \mathsf{mkMM}_\rho \twoheadrightarrow \mathsf{mkcsEEK} \rangle) \lesssim \mathsf{mkMgM} \;\; \wedge$

32 $\mathsf{mktPlaceholderMM}^{j}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge$

33 $\mathsf{mktTM}^{j}_\rho \lhd \mathsf{TMM}_\rho \;\; \wedge \;\; \mathsf{mktTM}^{j}_\rho \blacktriangleright \mathsf{mktMT}^{j}\,\langle \mathsf{mktPlaceholderMM}^{j}_\rho \twoheadrightarrow \mathsf{mkMM}_\rho \rangle \;\; \wedge$

34 $\mathsf{mkMgM} \lhd \mathsf{MgMM}_\rho$

35 $\nu(\{\mathsf{MMM}_\rho, \mathsf{TMM}_\rho, \mathsf{CMM}_\rho, \mathsf{CM}, \mathsf{ModelKindCWMM}_\rho, \mathsf{mkCWM},$

36 $\mathsf{ModelKindDocMM}_\rho, \mathsf{mkDocM}, \mathsf{mkMM}_\rho, \mathsf{ProblemMM}_\rho, \mathsf{mkcTM}^{\mathsf{i}}_\rho,$

37 $\mathsf{mktPlaceholderMM}^{j}_\rho, \mathsf{mktTM}^{j}_\rho\}) \lesssim \mathsf{mkMgM}$

Table 3.10: *Semantic equations:* ModelKind.

concerns (i.e. the set of concerns framed or covered by architecture models governed by the model kind), and transitively, in terms of stakeholders having those concerns. Then, even though the compositional traversal of the conceptual model in Figure 3.6 does not state it explicitly, we define the semantic equation for ModelKind : ○ (1) to be compositional on the semantic equation of Concern : ○ (2). In turn, the denotation of every mk $:_\circ$ ModelKind (14) includes the denotation of every concern c related to mk, i.e. every c ∈ mk.concerns (15). As we discussed in the specification of the semantic equation for concerns, we use model annotations to capture the information that is specific to the relationship between the model kind mk and its related concerns. To this end, the denotation includes the customized metamodel $\mathsf{ModelKindCWMM}_\rho$ (5) that specifies the modeling language for capturing the relationship between concerns and a particular model kind. Also, it includes a weaving model mkCWM (16) that conforms to the metamodel $\mathsf{ModelKindCWMM}_\rho$ (24), and that realizes the unary model weaving mkCMW on the CM (24). The woven model CM is the *concerns model* that is part of the denotation as $\mathsf{CM} \in [\![\mathsf{Concern} : \circ]\!] \wedge [\![\mathsf{Concern} : \circ]\!] \subseteq [\![\mathsf{ModelKind} : \circ]\!]$. For each related concern c to the model kind mk, the representation of the relationship $\langle \mathsf{mk}, \mathsf{c} \rangle$ is defined in mkCWM (25).

Documentation. The denotation of mk $:_\circ$ ModelKind includes the terminal model mkDocM (17) that represents the documentation of the model kind. Which information can be actually documented in this terminal model is determined by the customized metamodel $\mathsf{ModelKindDocMM}_\rho$ (6). Examples of the kind of information that might be captured are (i) administrative data such as name and title, overview, date and version, authors and affiliation, (ii) acronyms, definitions, reference materials and bibliographic references, and (iii) technical information such as usage examples, application scenarios and restrictions, general considerations, operations that are available on models of the kind such as related design techniques, analysis methods, among others. The main purpose of this model is to document all significant information relative to the model kind that is not actually captured in the other modeling artifacts in the denotation of the modeling kind. The metamodel $\mathsf{ModelKindDocMM}_\rho$ (6) defines the language for expressing this documentation (26). The metamodel can provide a precise domain-specific structure for the information to be documented, or it can provide a simple general structure mainly based on text attributes of one or few metaclasses. The decision is up to the practitioner community as the metamodel must be customized. The model mkDocM must be populated with the actual documentation for the model kind. Then, it must contain the representation of mk in terms of the metamodel $\mathsf{ModelKindDocMM}_\rho$ (26).

Metamodel. The denotation of mk $:_\circ$ ModelKind also includes the customized metamodel mkMM_ρ (18). This metamodel specifies the modeling language for expressing *architecture models* governed by the model kind mk. The metamodel must conform to the customized metametamodel MMM_ρ (27). The standard states in [ISO11, Section B.2.6.2] that the metamodel definition includes constraints on the entities, attributes and/or relationships in the models of this kind. In Model-Driven Engineering, the formal definition of any metamodel includes, in addition to the constructs, properties and relationships, the set of constraints that must be satisfied by every model conforming to the metamodel. These constraints are called *invariants* and can be defined in plain English, OCL, or any assertion language ap-

propriate for the purpose. The metametamodel must provide a mechanism for expressing invariants on metamodels. Then, a proper definition of a metamodel (which include the invariants imposing well-formedness constraints) satisfies the expectations of the standard. For the purpose of example, in our definition to the metamodel for megamodels in Section §2.3.1 we use invariants to declare assertions that must hold on all megamodels conforming to our metamodel. Particularly, Figure 2.6 illustrates invariants imposed on the conforms-to relationship on models and reference models.

Selectable constraints. The purpose of an *invariant* is to rule out models that, even though they conform to the structure defined by the metamodel, they are not actually valid models in the sense of the intention (domain) captured by the metamodel. However, during the activity of developing a particular model conforming to a metamodel, additional constraints may arise that are specific to the application scenario the modeler is working on. These constraints are not actually invariants as they must not be imposed on every model conforming to the metamodel. This kind of constraints allows the modeler to capture application-specific knowledge that rules or govern a given model. They are particularly useful when updating or evolving the model as there is an explicit representation of the decided set of constraints on the given model.

In order to illustrate this point, let us use as an example models governed by the a Module structure for styles from SEI's Views & Beyond approach to architecture documentation [CBB+10]. Consider a model ModulesM representing the modular decomposition of a system, and that conforms to the metamodel ModulesMM. The metamodel provides the construct Layer to define a layered organization of the module decomposition, having a many-to-many association between the layers to determine usage dependency between them. As the structure of the metamodel allows circular dependencies between layers, an *invariant* is used to enforce that the dependencies must be acyclical. This is an invariant as circular dependencies would allow a layer to depend on itself, and it is not valid in the domain of module decomposition. Now, when the architect decide to use a layered organization for modules, two alternatives are possible: a relaxed organization in which a layer can depend on any of its sub-layers, or a strict organization in which a layer can depend on a single sub-layer directly below and not on the sub-layers of that sub-layer. Whether to use one alternative or the other depends on the intention of the architect. The metamodel supports both kinds of organization. The problem is that, even if the current modules defined in the model follow a strict organization, we cannot determine whether the organization was decided by the architect and hence must not be broken by any update of the model, or if the architect intention is a relaxed organization although not used in the model yet. Then, we need a means to explicitly capture the constraint imposing an strict organization, that can be checked on the current version of the model under development, and on any future version of the model created during the development, maintenance and evolution of the system.

This kind of constraints in which it is up to the architect to decide whether they must be applied or not, is represented in our specification by means of a model transformation. The transformation encodes the knowledge of how to process the model to determine whether the constraint is being satisfied or not. While the source of the transformation is the model to constrain, we have different alternatives for its target. Provided that the transformation must

determine whether the model satisfies or not a constraint, a Boolean result would be enough. Transformations of this form are called queries as they return primitive types. In terms of our metamodel for megamodels, a query on a model is a ModelToExternal transformation, being the external entity a primitive value. The drawback of this approach is that no detailed feedback is provided to the modeler on why the constraint fails, which is the nature of the problem, which model element or elements are involved in the problem, etc. To address this issue, we use the $\mathsf{ProblemMM}_\rho$ metamodel that defines the constructs to provide detailed error information that result from any given action. This metamodel usually defines a *Problem* metaclass which attributes provide detailed information about the problem. It is important to notice that models conforming to this metamodel may be actually weaving models and the problems can be considered as annotations of the model being constrained. A model conforming to this metamodel is a set of model elements of type *Problem*, and possibly a link to the model elements involved in the problem. Then, the model transformation takes the source model and yields a possibly empty *problems model*. An empty problems model represents the successful scenario and a non-empty problems model provides detailed information on the error or errors found. Our model-based interpretation uses a customization for this metamodel in order to provide flexibility on how problems are actually documented. It is important to notice that this metamodel can be used in any scenario when we need to report the result of an action. Also, we can define a query transformation that yields a Boolean value indicating whether any problem was found or not. Then, the denotation of $\mathsf{ModelKind} : \circ$ includes the customized metamodel $\mathsf{ProblemMM}_\rho$ (7), and the denotation of every $\mathsf{mk} :_\circ \mathsf{ModelKind}$ includes a possibly empty set of transformation models $\mathsf{mkcTM}^\mathsf{i}$ (19) that realize the transformations mkcT^i from mkMM_ρ to $\mathsf{ProblemMM}_\rho$ (28). By defining a transformation (i.e. $n_c > 0$) for a model kind we are stating a constraint that can be applied on *architecture models* governed by the model kind mk; whether it is applied or not is up to the architect to decide. The transformation models are implemented in the model transformation language defined by the metamodel TMM_ρ.

Language. As we stated before, for a given $\mathsf{mk} :_\circ \mathsf{ModelKind}$, the metamodel mkMM_ρ (18) defines the modeling language for *architecture models* governed by mk. This metamodel covers method (3.2.2) of the standard's guide. Method (3.2.2) proposes to define the model kind via a language definition or a reference to an existing modeling language. In [ISO11, Section B.2.6.4] the standard states that an existing notation or model language must be identified or one must be defined, and that it must include syntax, semantics and tool support, as needed. The documentation model mkDocM (17) for mk can be used to capture this kind of definitions. For instance, bibliographical references to existing documentation on concrete syntax, semantics and tool support might be included in the model. However, we can further apply Model-Driven Engineering techniques to capture some of this knowledge in terms of modeling constructs.

Provided that our specification requires a mandatory metamodel mkMM_ρ (18) for defining the modeling language, we already count with a definition of the abstract syntax of the language. The concrete syntax of the language can be either textual, visual or both, and the defined concrete constructs might differ slightly of significantly from the constructs defined by the metamodel. Then, the artifact expressed in terms of the concrete syntax differs from the modeling artifact expressed in terms of the metamodel, even though they

are both representing the same entity. In term of modeling constructs, the concrete-syntax version of a model is actually an external entity. Then, the process of building a model from its concrete syntax is known as *parsing* while the reverse is known as *pretty-printing*. We define two transformations to formally capture these two processes. Let mkcsEEK (29) be the ExternalEntityKind representing the concrete syntax of the language defined by the model kind mk. Then, we use the injector mkcsIT (30) to capture the knowledge on how to parse an external entity of kind mkcsEEK to generate a model conforming to mkMM_ρ. mkcsIT is an ExternalToModelTransformation. Symmetrically, we use an extractor mkcsET (31) to capture the knowledge on how the definitions in a model conforming to mkMM_ρ are expressed in terms of the concrete syntax of the language in an external entity. mkcsET is a ModelToExternalTransformation. As the injector and extractor may not be implemented by means of a model transformation language, we do not include in the denotation the transformation models realizing these transformations. It is important to notice that as transformations are relationships, they are not part of the resulting set of modeling artifacts. Instead, they are recorded as relationships in the megamodel mkMgM (30, 31). The megamodel mkMgM (22) for a model kind is introduced later.

There are several mechanisms for capturing the semantics of a language, either informal, formal, or any intermediate level of rigor. Model-Driven Engineering techniques can barely assist on this task. Approaches like model transformations (queries) to provide an operational semantics, or weaving models to relate the constructs in the metamodel (source) to the constructs in other metamodel (target) of which there is a precise unambiguous understanding might be used. However, we claim that to suggest or force this practice in the context of our model-based interpretation for architecture description is excessive due to the required effort of such a formalization. We consider the semantics of the language, if present, to be captured in the documentation model mkDocM, possibly as a bibliographical reference to the semantic specification.

There is a wide variety of potential tool support for languages, and the standard provide no further classification or categorization of them. Current tool support, when available, vary in the purpose, the functionality, the usage scenario (attended or unattended), the development environment they operate in (if any), the underlying technology or platform they work with, among others. The semantic equations does not include any modeling artifact to represent tool support; we propose to capture the set of tools in the documentation model mkDocM. However, in practice, Model-Driven Engineering techniques can actually assist or perform the work of current tool support and, counting with a model-based formalization of architecture description may promote their development. We consider the inclusion of modeling artifacts representing tool support as a variation of our specification. The following examples describe transformations that can be defined for deal with some common scenarios:

- An analysis tool with the ability to work on models can be developed as a model-to transformation from the model kind metamodel mkMM_ρ to a given result (either a model or a primitive type):

  $\mathsf{t}\left\langle \mathsf{mkMM}_\rho \twoheadrightarrow \mathsf{Result} \right\rangle$

- A updating or processing tool with the ability to work on models can be developed as a model transformation on an InOut model conforming to the model kind metamodel:

  $\mathsf{t}\left\langle \overline{\mathsf{mkMM}_\rho} \twoheadrightarrow \varnothing \right\rangle$

- An external tool that performs analysis can be assisted by a model-to-external transformation that generates the tool-specific input artifact from the model conforming to the model kind metamodel:

  $\mathsf{t}\left\langle \mathsf{mkMM}_{\rho} \twoheadrightarrow \mathsf{ExternalToolInputKind} \right\rangle$

- An external tool that performs any kind of processing or update to a model conforming the model kind metamodel can be assisted by a transformation generating its input, as in the previous case, and by an mixed transformation that performs the updates in the model:

  $\mathsf{t}\left\langle \overline{\mathsf{mkMM}_{\rho}}, \mathsf{ExternalToolOutputKind} \twoheadrightarrow \varnothing \right\rangle$

Templates. The method (3.2.2) of the standard's guide proposes to define a model kind by providing a template or form specifying the format and/or content of models governed by the model kind [ISO11, Section B.2.6.3]. However, the standard makes no further comments on what is involved in a template. We distinguish two possible cases for these templates: document-based or model-based templates.

In the first case, the template consists of a structured textual document describing the sections and content of the model, generally by means of natural language and diagrams. Templates in this category are not actually templates of a model, they are templates of the documentation of a model. The template is not expressed in terms of the concrete or abstract syntax of the language and hence, we cannot provide any assistance on using them to create actual models. In terms of our model-based interpretation, this kind of templates are captured in the documentation model mkDocM (17) of the model kind.

In the second case, the template consists of a model expressed either in a relaxed form of the concrete or abstract syntax of the language of the model kind. This model template presents placeholders that must be replaced by the architect to build an initial working version of the model. The language syntax is relaxed in order to provide flexibility in the definition of these placeholders. Then, the template model may not necessarily be a well-formed model. Once the placeholders are filled-in, the resulting model satisfy the syntax. Then, provided a set of placeholder values and the model template, the architect can generate the initial version of the model. If the model template is expressed using the concrete syntax, an external tool must be used to produce the initial version. This tool is a syntax-specific editor. Later, the injector transformation mkcsIT (30) can be used to generate the model conforming to the metamodel mkMM_{ρ}. In our model-based interpretation, we also use the documentation model mkDocM_{ρ} (17) to preserve this kind of model template (or references to them) as they do not conform valid modeling artifacts. If the model template is expressed using the abstract syntax, the model template may not be a valid model due to placeholders, and hence, it may not be processable by model editors or model transformations. Even if the model template is a valid model, the architect must know how to proceed (and do it manually or tool-assisted) to fill-in the placeholders once the placeholder values were decided. In our model-based interpretation, we use a model transformation to encapsulate the knowledge on how to produce a valid initial version of a model from the set of the decided placeholder values. The denotation for $\mathsf{mk} :_{\circ} \mathsf{ModelKind}$ (14) includes a possibly empty set of transformation models $\mathsf{mktTM}_{\rho}^{j}$ (21) that realizes the transformation mktMT^{j} (33). Each transformation takes

a model defining the values for the placeholders, and produces a model conforming to the model kind metamodel mkMM_ρ. The values for the placeholders are defined in a model conforming to the customized and transformation-specific metamodel $\mathsf{mktPlaceholderMM}^j_\rho$ (20). Our specification uses a superscript index j to denote differently the metamodels of each transformation. However, the same customized metamodel might be used in more than one transformation. The more general the constructs defined by this metamodel, the greater the possibility to reuse it in more than one transformation.

Examples. Although the standard's guide suggests to use examples to illustrate *architecture viewpoints* in [ISO11, Section B.2.9], examples are not suggested for *model kinds*. Provided that architecture viewpoints are aggregations of model kinds, the examples of the former are surely created using examples of the latter. Examples might be reified using a document-based or model-based approach. Moreover, in the latter case, example models are not necessarily valid models with respect to their metamodel. A fragment of a model must be used, hence possibly breaking structural constraints. Notice that the usefulness of an example might no rely on its well-formedness, but in the way a given (possibly known) domain is captured by means of the modeling language.

Then, our model-based interpretation does not provide an special treatment for examples. We use the documentation model mkDocM_ρ (17) to capture and preserve examples. As a variation of our semantic equations, valid example models can be captures as terminal models conforming to the metamodel mkMM_ρ. Then, the denotation of $\mathsf{mk} :_\circ \mathsf{ModelKind}$ should include this possibly empty set of terminal models reifying example models.

Megamodel. Conceptually, a model kind is a work product that specifies the conventions for building a kind of architecture models, and is conceived as reusable and shareable asset. From a model-based perspective, a model kind is a complex work product conformed by several modeling artifacts. As we have analyzed so far, the denotation of $\mathsf{mk} :_\circ \mathsf{ModelKind}$ includes the foundational modeling artifacts provided by the denotation of $\mathsf{ModelKind} : \circ$ and several modeling artifacts that are specific to the model kind mk. In the context of a model repository containing a large number of modeling artifacts required for different purposes, we need the ability to capture and specify which of those artifacts are actually part of the model kind mk and which are not. Moreover, we need a mechanism to explicitly capture and document the properties and relationships among these artifacts, particularly those specified by means of assertions in the semantic equations. We use a megamodel for this purpose. The semantic equation for $\mathsf{mk} :_\circ \mathsf{ModelKind}$ (14) includes a megamodel mkMgM (22) that consists of a complete representation of all the modeling artifacts required to denote mk. The assertions (35–37) use the purpose-specific representation-of function ν defined in Section §3.2.1 (ii) to state that the modeling artifacts are represented in the megamodel mkMgM. It is important to notice that not every modeling artifact in the denotation is represented in the megamodel. For instance, the megamodel mkMgM itself is not represented. Also, from the modeling artifacts produced by the denotation of $\mathsf{Concern} : \circ$, only those directly related to mk are represented; for instance, notice that the transformation model $\mathsf{CMergeTM}_\rho$ defined in Table 3.8–(5) is not included.

The megamodel mkMgM also captures the elements and relationships that are not reified or realized by the modeling artifacts in the resulting denotation. For instance, the external entity kind mkcsEEK (29) of the concrete syntax, and the corresponding injector mkcsIT (30) and extractor mkcsET (31) transformations are represented in the megamodel mkMgM.

The megamodel mkMgM conforms to the metamodel for megamodels MgMM_ρ (34). The metamodel MgMM_ρ (8) is marked as customized because it must be expressed in terms of the customized metametamodel MMM_ρ (13). However, the customization of the metamodel for megamodels must not differ from our definition in Section §2.3.1 as the specification relies on the structure we defined; see Section §3.2.1 (ii) for details.

Architecture Models

An *architecture model* is an abstract or simplified representation of some aspects of an architecture, used to capture, analyze and communicate those aspects of the system. It reflects the impact of architecture decisions and acts as a container for the application of patterns or styles, and it adheres to a governing *model kind* that specifies the conventions for building the architecture model. It is a reusable asset as it can be shared across architecture views, and also it might be reused in different architecture descriptions. However, an architecture model is system-dependent as it represents some aspects of the architecture of a particular system or set of related (similar) systems. In the context of our model-based interpretation, we modularize and encapsulate the denotation of architecture models, making it compositional on the governing model kind only. The denotation of an architecture model does not depend on the definition of any other architecture model or of any architecture view. Then, an architecture model is denoted by a terminal model conforming to the metamodel defined by the model kind, and a set of constraints that have been decided to rule on the architecture model. We also use a documentation model to capture administrative information and reference material.

Table 3.11 defines the semantic equations for the concept ArchitectureModel (1) and its instances (12). The denotation of ArchitectureModel : ○ is compositional on the denotation of ModelKind : ○, and it includes several reference models that provide the foundation for denoting architecture models. The metametamodel MMM_ρ (3) determines the technical space for all metamodels. The customized metamodel $\mathsf{ArchitectureModelDocMM}_\rho$ (4) provides the constructs to capture administrative information on the model, such as name, title, overview, date, version, author and affiliation, and reference material and bibliographic references. We do not enforce any structure for documentation models, and in practice, in the case of architecture models, it can simply define bookkeeping information. The denotation also includes the metamodel TMM_ρ (5) that determines the modeling language for developing transformation models. The customized metamodel $\mathsf{ProblemMM}_\rho$ (6) was introduced before when we discuss the semantic equations for model kinds. It defines the language for providing detailed error information, and it is used as the output of the evaluation of constraints. The metamodel for megamodels MgMM_ρ (7) is also included.

The denotation of am $:_\circ$ ArchitectureModel is compositional on ArchitectureModel : ○ and on the denotation of the single model kind mk governing am. Formally, mk = am.modelKind

1	$\llbracket \mathsf{ArchitectureModel} : \circ \rrbracket$
2	$\quad \overset{\text{def}}{=} \llbracket \mathsf{ModelKind} : \circ \rrbracket \cup$
3	$\quad\quad \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$
4	$\quad\quad \mathsf{ArchitectureModelDocMM}_\rho :_\bullet \mathsf{Metamodel},$
5	$\quad\quad \mathsf{TMM}_\rho :_\bullet \mathsf{Metamodel},$
6	$\quad\quad \mathsf{ProblemMM}_\rho :_\bullet \mathsf{Metamodel},$
7	$\quad\quad \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
8	$\quad \mathsf{ArchitectureModelDocMM}_\rho \lhd \mathsf{MMM}_\rho \quad \wedge$
9	$\quad \mathsf{TMM}_\rho \lhd \mathsf{MMM}_\rho \quad \wedge$
10	$\quad \mathsf{ProblemMM}_\rho \lhd \mathsf{MMM}_\rho \quad \wedge$
11	$\quad \mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho$
12	$\llbracket \mathsf{am} :_\circ \mathsf{ArchitectureModel} \rrbracket$
13	$\quad \overset{\text{def}}{=} \llbracket \mathsf{ArchitectureModel} : \circ \rrbracket \cup \llbracket \mathsf{mk} \rrbracket \cup$
14	$\quad\quad \{\mathsf{amDocM} :_\bullet \mathsf{TerminalModel},$
15	$\quad\quad \mathsf{amM} :_\bullet \mathsf{TerminalModel},$
16	$\quad\quad \mathsf{amcTM}^{\mathsf{i}}_\rho :_\bullet \mathsf{TransformationModel},$
17	$\quad\quad \mathsf{amMgM} :_\bullet \mathsf{Megamodel}\}$
18	for $1 \leq \mathrm{i} \leq n_c, 0 \leq n_c$ with $\mathsf{mk} = \mathsf{am.modelKind}$ such that
19	$\quad \mathsf{amDocM} \lhd \mathsf{ArchitectureModelDocMM}_\rho \quad \wedge \quad \mathsf{am}_\mu \lesssim \mathsf{amDocM} \quad \wedge$
20	$\quad \mathsf{amM} \lhd \mathsf{mkMM}_\rho \quad \wedge$
21	$\quad \mathsf{amcTM}^{\mathsf{i}}_\rho \lhd \mathsf{TMM}_\rho \quad \wedge \quad \mathsf{amcTM}^{\mathsf{i}}_\rho \blacktriangleright \mathsf{amcMT}^{\mathsf{i}} \langle \mathsf{mkMM}_\rho \twoheadrightarrow \mathsf{ProblemMM}_\rho \rangle \quad \wedge$
22	$\quad \mathsf{amMgM} \lhd \mathsf{MgMM}_\rho \quad \wedge$
23	$\quad \nu(\{\mathsf{MMM}_\rho, \mathsf{TMM}_\rho, \mathsf{ProblemMM}_\rho, \mathsf{ArchitectureModelDocMM}_\rho,$
24	$\quad\quad \mathsf{amDocM}, \mathsf{amM}, \mathsf{amcTM}^{\mathsf{i}}_\rho, \mathsf{mkMgM}, \mathsf{mkMM}_\rho\}) \lesssim \mathsf{amMgM}$

Table 3.11: *Semantic equations:* ArchitectureModel.

(18). The denotation of am includes a terminal model amDocM (14) to capture administrative and reference information related to the architecture model. A representation of am expressed in terms of the constructs defined by the metamodel $\mathsf{ArchitectureModelDocMM}_\rho$ is defined in amDocM (19). The denotation also includes the terminal model amM (15) that consists of the actual representation of those aspects of the system's architecture which are the purpose of the governing model kind mk. The terminal model amM conforms to the metamodel mkMM_ρ (20), i.e. the metamodel defined by the denotation of the model kind mk in Table 3.10–(18) that specifies the modeling language for architecture models governed by mk.

As we explained before, a *model kind* defines a possibly empty set of constraints that can be applied to architecture models governed by the model kind as shown in Table 3.10–(19). Then, for every particular architecture model, we need to preserve which are the selected constraints, i.e. the set of constraints that the architect decide to capture in order to rule the current and future versions of the architecture model. To this end, the denotation of $\mathsf{am} :_\circ \mathsf{ArchitectureModel}$ includes a possibly empty set of transformation models $\mathsf{amcTM}^{\mathsf{i}}_\rho$ (16), that are implemented in the transformation modeling language TMM_ρ (21) and that realize the transformations $\mathsf{amcMT}^{\mathsf{i}}$ from mkMM_ρ to $\mathsf{ProblemMM}_\rho$ (21).

From the model-based perspective, an *architecture model* is a complex work product involving several modeling artifacts. Then, we include in the denotation of am the megamodel amMgM (17) to capture a complete representation of all the modeling artifacts required to denote am. The assertions (23–24) use the purpose-specific representation-of function ν defined in Section §3.2.1 (ii) to state which modeling artifacts are represented in amMgM. It is important to notice that the megamodel mkMgM defined for mk is represented in amMgM to capture the fact that mk is the governing model kind for am. However, we also include in the megamodel amMgM the representation of the metamodel mkMM_ρ defined by the denotation of the model kind mk as it is directly related to the modeling artifacts in the denotation of am, namely its terminal model amM and the constraint transformations $\mathsf{amcMT}^\mathsf{i}$.

Architecture Viewpoints

An *architecture viewpoint* specifies the conventions for building a specific kind of *architecture views* and it is conceived as a system-independent reusable asset to be shared across different architecture descriptions. An architecture viewpoint is organized as an aggregation of *model kinds* that specify the conventions for building the *architecture models* conforming the *architecture views* governed by the architecture viewpoint. As we explained before when we analyzed the semantic equations of model kinds, the ISO/IEC/IEEE 42010:2011 standard provides an informative guide in [ISO11, Annex B] on how to capture and document *architecture viewpoints* in a form that it meets the standard requirements. This guide is structured in ten named slots, providing a description of the intended content and guidance for developing that content. In what follows, we discuss how our model-based interpretation captures that content. We structure the discussion in different aspects, that, altogether, cope with all the slots defined in the guide.

Table 3.12 defines the semantic equations for the concept ArchitectureViewpoint (1) and its instances (10). As is the case for the concepts we explained before, the denotation of ArchitectureViewpoint : $\circ$ includes the metametamodel MMM_ρ (3) that determines the technical space for all metamodels. The denotation of avp $:_\circ$ ArchitectureViewpoint is compositional on the denotation of ArchitectureViewpoint : $\circ$ (11) which provides the foundational modeling artifacts for the modeling artifacts of any architecture viewpoint.

Documentation. As is the case for *model kinds*, the denotation of the concept instance avp $:_\circ$ ArchitectureViewpoint includes the terminal model avpDocM (13) that represents the documentation of the architecture viewpoint. The kinds of information that can be preserved are similar to those for documenting model kinds: (i) administrative information, (ii) identification of and reference to external entities and materials, and (iii) technical information. In terms of the information slots from the standard's guide, (i) includes *name* and *overview*, (ii) includes *sources*, and (iii) includes *operations* and *examples*. The *notes* slot can be part of any of these three kinds of information. Our model-based interpretation does not enforce any particular structure to capture and document this information, neither using the three kinds we mentioned, nor using the slots suggested by the standard's guide. The information that can be actually modeled is determined by the customized metamodel $\mathsf{ArchitectureViewpointDocMM}_\rho$ (4). It is up to the metamodel to provide constructs to capture all necessary information to

1	$\llbracket \mathsf{ArchitectureViewpoint} : \circ \rrbracket$
2	$\stackrel{\text{def}}{=} \llbracket \mathsf{Concern} : \circ \rrbracket \ \cup \ \llbracket \mathsf{ModelKind} : \circ \rrbracket \ \cup \ \llbracket \mathsf{CorrespondenceRule} : \circ \rrbracket \ \cup$
3	$\{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$
4	$\mathsf{ArchitectureViewpointDocMM}_\rho :_\bullet \mathsf{Metamodel},$
5	$\mathsf{ArchitectureViewpointCWMM}_\rho :_\bullet \mathsf{Metamodel},$
6	$\mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
7	$\mathsf{ArchitectureViewpointDocMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
8	$\mathsf{ArchitectureViewpointCWMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
9	$\mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho$
10	$\llbracket \mathsf{avp} :_\circ \mathsf{ArchitectureViewpoint} \rrbracket$
11	$\stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureViewpoint} : \circ \rrbracket \ \cup \ \llbracket \mathsf{avp.concerns} \rrbracket \ \cup$
12	$\llbracket \mathsf{avp.modelKinds} \rrbracket \ \cup \ \llbracket \mathsf{avp.correspondenceRules} \rrbracket \ \cup$
13	$\{\mathsf{avpDocM} :_\bullet \mathsf{TerminalModel},$
14	$\mathsf{avpCWM} :_\bullet \mathsf{WeavingModel},$
15	$\mathsf{avpMM}_\rho :_\bullet \mathsf{Metamodel},$
16	$\mathsf{avpMgM} :_\bullet \mathsf{Megamodel}\}$ such that
17	$\mathsf{avpDocM} \lhd \mathsf{ArchitectureViewpointDocMM}_\rho \ \wedge \ \mathsf{avp}_\mu \lesssim \mathsf{avpDocM} \ \wedge$
18	$\mathsf{avpCWM} \lhd \mathsf{ArchitectureViewpointCWMM}_\rho \ \wedge \ \mathsf{avpCWM} \blacktriangleright \mathsf{avpCMW} \langle \mathsf{CM} \rangle \ \wedge$
19	$\forall \mathsf{c} \in \mathsf{avp.concerns}. \ \langle \mathsf{avp}, \mathsf{c} \rangle_\mu \lesssim \mathsf{avpCWM} \ \wedge$
20	$\forall \mathsf{mk} \in \mathsf{avp.modelKinds}. \ \exists \mathsf{c} \in \mathsf{mk.concerns}. \ \langle \mathsf{avp}, \mathsf{c} \rangle_\mu \lesssim \mathsf{avpCWM} \ \wedge$
21	$\forall \mathsf{mk} \in \mathsf{avp.modelKinds}. \ \mathsf{mkMgM}_\nu \lesssim \mathsf{avpMgM} \ \wedge$
22	$\mathsf{avpMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
23	$\forall \mathsf{mk} \in \mathsf{avp.modelKinds}. \ \mathsf{mkMM}_\rho \rightharpoonup \mathsf{avpMM}_\rho \ \wedge$
24	$\forall \mathsf{cr} \in \mathsf{avp.correspondenceRules}. \ \mathsf{crMgM}_\nu \lesssim \mathsf{avpMgM} \ \wedge$
25	$\mathsf{avpMgM} \lhd \mathsf{MgMM}_\rho \ \wedge$
26	$\nu(\{\mathsf{MMM}_\rho, \mathsf{ArchitectureViewpointDocMM}_\rho, \mathsf{avpDocM}, \mathsf{MgMM}_\rho, \mathsf{avpMM}_\rho,$
27	$\mathsf{CMM}_\rho, \mathsf{CM}, \mathsf{ArchitectureViewpointCWMM}_\rho, \mathsf{avpCWM}\}) \lesssim \mathsf{avpMgM}$

Table 3.12: *Semantic equations:* ArchitectureViewpoint.

meet the standard's requirements. The metamodel can be purpose-specific and provide a precise structure to organize the information, or it can be general-purpose and provide a simple structure for textual information. In the latter case, not only the metamodel is simpler, but also it can be used to document other concepts such as model kinds. However, the lack of fine-grained and purpose-specific structure may yield to documentations that are unalike. The decision is up to the practitioner community to find the suitable middle ground. The model avpDocM must be populated with the actual documentation. Then, it must contain the representation of avp in terms of $\mathsf{ArchitectureViewpointDocMM}_\rho$ (17).

Concerns. The conceptual model in Figure 3.2 indicates that every architecture viewpoint *frames* a set of concerns, i.e. an architecture viewpoint formulates or encloses the description techniques to illustrate or capture the set of concerns. In turn, *concerns* and their interested *stakeholders* characterize the purpose of architecture viewpoints. In order to capture this re-

lationship between architecture viewpoints and concerns, we proceed analogously to the case of *model kinds*. The denotation of ArchitectureViewpoint : ○ is compositional on Concern : ○ (2) and avp :$_\circ$ ArchitectureViewpoint is compositional on the denotation of every concern in avp.concerns (11). We use model annotations to capture the information that is specific to the relationship between an architecture viewpoint and its framed concerns. Then, the denotation includes the customized metamodel $\mathsf{ArchitectureViewpointCWMM}_\rho$ (5) that specifies the modeling language for capturing the relationship, and a weaving model avpCWM (14) that conforms to this metamodel (18) and that realizes the unary model weaving avpCMW on the model CM (18). For each framed concern c, the representation of the relationship $\langle \mathsf{avp}, \mathsf{c} \rangle$ is defined in avpCWM (19).

As we explained before when we discussed the semantic equations for *model kinds*, we introduce a relationship between model kinds and concerns both in the context of architecture descriptions and architecture frameworks. As illustrated in Figures 3.2, 3.4 and 3.5, the standard conceive this relationship only in the context of architecture description languages. Then, given that an architecture viewpoint and every one of its aggregated model kinds frames concerns, it is important to determine whether these sets of framed concerns are related (in the sense of set relationships) to each other and how they are related. This fact is not addressed in the standard as in its conceptual model, model kinds do not frame concerns. Let avpcs = avp.concerns be the set of all concerns framed by an architecture viewpoint avp and let avpmkscs = avp.modelKinds.concerns→flatten() be the set of all concerns framed by all models kinds of avp. We expect that $\mathsf{avpcs} \subseteq \mathsf{avpmkscs}$ as avp *uses* its aggregated model kinds to describe the architecture from a particular point of view and this point of view frames the set of concerns avpcs. Having a concern $\mathsf{c} \in \mathsf{avpcs} \ \wedge \ \mathsf{c} \notin \mathsf{avpmkscs}$ implies that the concern c is framed by the viewpoint but no model kind provides conventions for building models to capture the description on the resolution of the concern c. Then, the definition of the architecture viewpoint would not be framing the concern. However, we cannot expect that every concern in avpmkscs is also in avpcs. Model kinds are reusable assets by themselves which can be defined in isolation; for instance, an architecture description language may define model kinds without considering architecture viewpoints. When an architecture viewpoint includes a model kind in its definition, it might be the case that the purpose of the model kind in the architecture viewpoint is not to frame all the concerns the model kind frames, but only some of these concerns. However, at least one concern framed by the model kind must also be framed by the architecture viewpoint, formally, $\forall \mathsf{mk} \in \mathsf{avp.modelKinds}.\ \exists \mathsf{c} \in \mathsf{mk.concerns}.\ c \in \mathsf{avp.concerns}$. This restriction is actually an invariant that should be considered in the conceptual model. In terms of our semantic equations, the impact of this invariant is that for at least one concern framed by the model kind, the relationship between the architecture viewpoint and this concern is represented in the weaving model avpCWM (20). We force this assertion at the denotation level as it is not being enforced at the conceptual level.

Model kinds. Provided that an *architecture viewpoint* is an aggregation of *model kinds*, to capture and document an architecture viewpoint requires to capture and document every model kind in the viewpoint. This fact is stated in the conceptual model in Figure 3.2 and well as in the standard guide for documenting architecture viewpoints [ISO11, Section B.2.6]. We used this section of the guide when we defined and analyzed the semantic equations for model

kinds. Also, considering the compositional traversal of the conceptual model we decided illustrated in Figure 3.6, the denotation of architecture viewpoint is compositional with the denotation of model kinds. Then, $\mathsf{ArchitectureViewpoint} : \circ$ includes $\mathsf{ModelKind} : \circ$ (2) and $\mathsf{avp} :_{\circ} \mathsf{ArchitectureViewpoint}$ includes $[\![\mathsf{avp.modelKinds}]\!]$ (12). By this means, the denotation of a architecture viewpoint avp includes all the modeling artifacts representing each of its aggregated model kinds. Although necessary, this composition is not enough to preserve the relationship between an architecture viewpoint and its model kinds. In the case of a model repository storing the modeling artifacts for several architecture viewpoints, and for all of their model kinds, we would be missing the information of which are the actual model kinds for each architecture viewpoint. In order to capture this relationship, we need to capture these modeling artifacts in a megamodel. As we introduce later, we define a megamodel to capture the set of modeling artifacts that conforms the denotation of an architecture viewpoint. Then, (some of) the modeling artifacts denoting model kinds must be defined (captured, recorded) in the megamodel for the architecture viewpoint. In order to preserve the modularization and encapsulation provided by the semantic equations, we do not include all the modeling artifacts of each model kind, we uniquely include their defining megamodel. Then, we have that for every $\mathsf{mk} \in \mathsf{avp.modelKinds}$, the megamodel mkMgM is defined (represented) in the megamodel avpMgM of the architecture viewpoint avp. It is important to notice that $\mathsf{mkMgM} \in [\![\mathsf{mk} :_{\circ} \mathsf{ModelKind}]\!]$ is the megamodel which purpose is to capture every modeling artifact representing the model kind mk (21).

Metamodel. The standard states as a side note in [ISO11, Section B.2.6.2] that "when a [architecture] viewpoint specifies multiple model kinds it is often useful to specify a single viewpoint metamodel unifying the definition of the model kinds." We claim that the *usefulness* of having a single and unifying metamodel cannot be argued for the general case as it not only depends on the purpose and goal of the architecture viewpoint, but also on the characteristics of the single modeling language defined by the unifying metamodel. The impact of this recommendation is not minor when capturing an architecture viewpoint and its aggregated model kinds, and it must be attended to and considered carefully.

If the modeling language defined by the unifying metamodel is too specific, i.e. it provides a reduced set of constructs that are specific to a particular solution space, the language restricts the expressiveness of the views and models. As an example, consider a modeling language providing constructs for building Pipes & Filters [BMR+96] solutions, enabling practitioners to capture the decomposition of components in terms of this architecture style. In practice, more than one architecture style is applied when deciding and capturing the decomposition of the various components conforming the system. Then, this modeling language is too specific to develop all models showing component decomposition.

If the modeling language is not too specific, we can distinguish two different cases. In the first case, the language provides a large set of constructs enabling practitioners to capture almost every aspect of the system. Such a language is actually a cohesive and coherent aggregation or unification of separate modeling languages, thought and defined to fit altogether. An example of this kind of languages is the Unified Modeling Language (UML) [OMG11c]. The benefit of a general-purpose language is that once mastered by practitioners and stakeholders, it can be used throughout one or even every architecture viewpoint. The main

drawback is that, being general-purpose, domain-specific (i.e. solution-specific) constructs are not available and hence, practitioners cannot use common constructs in the solution space to express the architecture models. This is usually achieved by extension mechanisms, both in UML and in extensible ADLs as we explained in Section §3.1.2. In the second case, the language provides a reduced set of constructs but general enough in order not to restrict the expressiveness of the language with respect to its purpose. Consider the example of component decomposition we mentioned before, a language in this category would provide a set of constructs to refer to components, connectors, and their decomposition in terms of component and connectors [CBB+10]. Then, practitioners are not restricted to a particular architecture style, but still can work in with a specific modeling purpose (decomposition). This kind of language is more oriented to capture a specific point of view of the system, in contrast to general-purpose languages that can be used to capture any point of view. However, the constructs are too abstract to allow practitioners to express the architecture with in terms of solution constructs. For instance, a Component & Connector modeling language does not actually provide solution-specific constructs like Pipe or Filter, and the more abstract constructs must be used to express elements of these specific kinds.

Then, a middle ground between a general-purpose and a domain-specific language is desired. However, it may not be feasible to conceive and define such a middle ground language for every conceivable architecture viewpoint, it can be achieved for some of them. In practice, practitioners use the available extensibility mechanisms of general languages to define domain-specific ones. For instance, UML provides several extension mechanisms such as tagged values and stereotypes, and UML profiles. Also, as we discussed in Section §3.1.2, current ADLs provide a base general language which is extended to cope with particular domains. Our position is that the recommendation for architecture viewpoint developers should be to identify the general-purpose language that acts as the foundation for the languages of the aggregated model kinds, which in turn, should define their own languages as an extension of the viewpoint's language, using the available extension mechanisms. This is the recommendation that we capture in our model-based interpretation.

The denotation of $\mathsf{avp} :_\circ \mathsf{ArchitectureViewpoint}$ includes the metamodel avpMM_ρ (15) that defines the extensible modeling language that provide the foundation for the languages of the aggregated model kinds. Then, for every $\mathsf{mk} :_\circ \mathsf{ModelKind}, \mathsf{mk} \in \mathsf{avp.modelKinds}$ it holds that the metamodel mkMM_ρ of the model kind mk is an *extension* of the metamodel (15). In Section §2.2.2 we discussed the two dimensions of metamodeling, namely linguistic and ontological, and in Section §2.3.1 we explained that the *conforms-to* relationship captures the linguistic dimension, and that we add the *requires* relationship to enable us capture the ontological dimension. So far, in the semantic equations we have only used the *conforms-to* relationship to state that every metamodel is expressed in terms of the metametamodel MMM_ρ. Particularly, we have that $\mathsf{avpMM}_\rho \lhd \mathsf{MMM}_\rho$ (22) and that $\mathsf{mkMM}_\rho \lhd \mathsf{MMM}_\rho$ in Table 3.10–(27). In the case of the metamodels of model kinds, we also need to state the relationship through the ontological dimension. To this end, we state that $\mathsf{mkMM}_\rho \multimap \mathsf{avpMM}_\rho$ (23). It is important to notice that this assertion is actually introduced by the semantic equation for an architecture viewpoint and not for the model kind. The model kind is a reusable asset that can be defined in isolation. However, for an architecture viewpoint to include a model kind it must enforce (ensure) that the *requires* relationship holds.

Correspondence rules. Although the conceptual model do not define an association between *architecture viewpoint* and *correspondence rules* (see Figures 3.2, 3.4 and 3.5), the standard guide suggests in [ISO11, Section B.2.7] to document the correspondence rules pertaining the architecture viewpoint. For the standard guide, "these rules will be 'cross model' or 'cross view' since constraints within a model kind will have been specified as part of the conventions of that model kind." *Cross model* is understood as correspondence rules between different architecture models governed the model kinds aggregated in the architecture viewpoint. *Cross view* implies *cross model*, but allow these architecture models to be defined in different architecture views. We claim that the case for *cross view* correspondence rules cannot be captured at the architecture viewpoint level. In the context of an *architecture description*, for each *architecture viewpoint* in use there is one and only one *architecture view* in the architecture description that is governed by that architecture viewpoint. This is explicitly stated in the conceptual model by the one-to-one association between architecture viewpoints and architecture views. Then, for an architecture viewpoint to include *cross view* correspondence rules, it must refer to views governed by a different architecture viewpoint. This cannot be actually captured as architecture viewpoints are reusable assets to be defined in isolation and not interrelationship between them (such as requires, uses, depends on) is conceived in the conceptual model. For this reason, architecture descriptions, architecture frameworks and architecture description languages includes a set of correspondence rules: to enable practitioners to define and preserve *cross view* relationships.

In order to allow practitioners to follow the standard guide, we consider the case of *cross model* correspondence rules at the architecture viewpoint level in our semantic equations. Then, we define $\mathsf{ArchitectureViewpoint} : \circ$ compositional on $\mathsf{CorrespondenceRule} : \circ$, and $\mathsf{avp} :_\circ \mathsf{ArchitectureViewpoint}$ compositional on $[\![\mathsf{avp.correspondenceRules}]\!]$, even though this is not represented in the traversal of the conceptual model in Figure 3.6. In order to preserve compositionality, the correspondence rules associated to an architecture viewpoint can only work (rule) on elements defined by the architecture viewpoint itself. We introduce the semantic equations for correspondence rules later in this section. By now, consider that the denotation of a correspondence rule includes a megamodel representing the set of modeling artifacts that represent the correspondence rule, and hence, the relationship between the architecture viewpoint and the correspondence rules is preserved in the megamodel of the architecture viewpoint (24), analogously to how we proceeded for the relationship to model kinds.

Megamodel. From the model-based perspective, an architecture viewpoint is a complex work product conformed by several modeling artifacts. As is the case for the complex work products we analyzed before, we use a megamodel to capture the set of modeling artifacts involved in the denotation of the complex work product, as well as their properties and relationships. The semantic equation for $\mathsf{avp} :_\circ \mathsf{ArchitectureViewpoint}$ includes the megamodel avpMgM (16) consisting of the complete representation of all the modeling artifacts required to denote avp. The assertions (26–27) use the purpose-specific representation-of function ν defined in Section §3.2.1 (ii) to state that the modeling artifacts are represented in the megamodel avpMgM. Assertions (21) and (24) states that the megamodel for the composed complex work products of model kinds and correspondence rules, respectively, are also represented in the megamodel avpMgM.

1	$\llbracket \mathsf{ArchitectureView} : \circ \rrbracket$
2	$\quad \stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureViewpoint} : \circ \rrbracket \ \cup \ \llbracket \mathsf{ArchitectureModel} : \circ \rrbracket \ \cup$
3	$\qquad \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$
4	$\qquad \mathsf{ArchitectureViewDocMM}_\rho :_\bullet \mathsf{Metamodel},$
5	$\qquad \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
6	$\qquad \mathsf{ArchitectureViewDocMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
7	$\qquad \mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho$
8	$\llbracket \mathsf{av} :_\circ \mathsf{ArchitectureView} \rrbracket$
9	$\quad \stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureView} : \circ \rrbracket \ \cup \ \llbracket \mathsf{avp} \rrbracket \ \cup \ \llbracket \mathsf{av.architectureModels} \rrbracket \ \cup$
10	$\qquad \{\mathsf{avDocM} :_\bullet \mathsf{TerminalModel},$
11	$\qquad \mathsf{avMgM} :_\bullet \mathsf{Megamodel}\}$
12	$\qquad$ with $\mathsf{avp} = \mathsf{av.architectureViewpoint}$ such that
13	$\qquad \mathsf{avDocM} \lhd \mathsf{ArchitectureViewDocMM}_\rho \ \wedge \ \mathsf{av}_\mu \lesssim \mathsf{avDocM} \ \wedge$
14	$\qquad \forall \mathsf{am} \in \mathsf{av.architectureModels}.\ \mathsf{amMgM}_\nu \lesssim \mathsf{avMgM} \ \wedge$
15	$\qquad \mathsf{avMgM} \lhd \mathsf{MgMM}_\rho \ \wedge$
16	$\qquad \nu(\{\mathsf{MMM}_\rho, \mathsf{ArchitectureViewDocMM}_\rho, \mathsf{avDocM}, \mathsf{MgMM}_\rho, \mathsf{avpMgM}\}) \lesssim \mathsf{avMgM}$

Table 3.13: *Semantic equations:* ArchitectureView.

Architecture Views

An *architecture view* is a representation of a *system-of-interest* from the perspective of an *architecture viewpoint*. It consists of an aggregation of *architecture models* that, altogether, capture and communicate the system's *architecture* from that perspective and enable analysis of the realization of the set of concerns framed by the architecture viewpoint. An architecture view is a reusable asset that can be shared across *architecture descriptions*. However, being a system-dependent work product, its reusability is restricted to the context of a family of systems or a product line. The denotation of an architecture view depends on the denotation of its governing architecture viewpoint and of the set of aggregated architecture models, as stated by the traversal of the conceptual model illustrated in Figure 3.6. The conceptual model depicted in Figure 3.2 includes an association between *architecture views* and *concerns*, being these concerns the ones addressed by the architecture view. Provided that the architecture view is governed by a single architecture viewpoint which frames a set of concerns, it is expected that these framed concerns are those addressed by the architecture view. Formally, for each $\mathsf{av} :_\circ \mathsf{ArchitectureView}$ is holds that $\mathsf{av.concerns} = \mathsf{av.architectureViewpoint.concerns}$. However, this invariant is not explicitly mentioned in the standard [ISO11]. We consider that explicitly capturing this association requires additional effort for recording information that is already present. While it is accurate to include the association in the conceptual level, it is not required to capture it in practice. Then, we do not capture this relationship explicitly as we can derive it by querying the already defined modeling artifacts.

Table 3.13 defines the semantic equations for the concept ArchitectureView (1) and its instances (8). The denotation of $\mathsf{ArchitectureView} : \circ$ is compositional on the denotation of $\mathsf{ArchitectureViewpoint} : \circ$ and $\mathsf{ArchitectureModel} : \circ$. Also, it includes the metametamodel MMM_ρ that determines the technical space for all metamodels, a customized metamodel

ArchitectureViewDocMM$_\rho$ (4) and the metamodel for megamodels MgMM$_\rho$ (5).

The denotation of av :$_\circ$ ArchitectureView is compositional on the denotation of the concept ArchitectureView (9), the denotation of the single architecture viewpoint avp (9) defined as av.architectureViewpoint (12), and the denotation of all the architecture models aggregated by the architecture view av.architectureModels (9). The denotation of av includes a terminal model avDocM (10) to capture administrative and documentation information related to the architecture view. A representation of av, expressed in terms of the constructs defined by the customized metamodel ArchitectureViewDocMM$_\rho$, is defined in avDocM (13).

As an architecture view is a complex work product involving several modeling artifacts, the denotation of av :$_\circ$ ArchitectureView includes the megamodel avMgM (11) to capture a representation of the whole set of participating artifacts. The megamodel includes a representation of the megamodel avpMgM, i.e. the megamodel defining the architecture viewpoint avp that governs the architecture view (16), and the megamodels amMgM included in the denotation of every am $\in$ av.architectureModels (14). Additionally, the documentation model avDocM and all the required reference models are also represented in avMgM (16).

Correspondences

The *architecture description* of the *architecture* of a *system-of-interest* is decomposed in terms of *architecture views* and *architecture models*. These architecture views, and particularly the architecture models, capture and represent only some aspects of the system-of-interest. However, as they refer to the same system-of-interest, it is expected that they are somehow interrelated. *Correspondences* provide a composition mechanism among the architecture views and architecture models. Correspondences serve the purpose of capturing and representing the relationships of interest between the elements populating the architecture description.

Architecture Description Element. As we explained in Section §3.1.1, the conceptual model uses the notion of *architecture description element* (ADE), as illustrated in Figure 3.3, to represent what can be related by a *correspondence*. As we argued before, from the domain modeling perspective, the concept of ADE is not properly defined in the conceptual model. The standard states that the concept of ADE includes (i) every concept instance of the concepts in the conceptual model, such as concerns, stakeholders, architecture viewpoints, architecture views, model kinds and architecture models, and (ii) every element or instance of the kinds of elements defined and introduced by architecture viewpoints and model kinds, that are used to populate architecture views and architecture models. The conceptual model in the standard lacks a unifying concept for this purpose; it introduces the concept of ADE to this end as it is required to conceptualize *correspondences*. However, the conceptual model does not state how the concept ADE is related (by means of associations) to any of the other concepts in the conceptual model, which would be expected to capture (i). Moreover, the conceptual model does not include any construct to represent the kinds of elements defined within architecture viewpoints and architecture models, which is required to capture (ii).

In our model-based interpretation of the conceptual model, our denotation does have a

unifying construct to represent both the concept instances (i) and the instances of the kinds introduced by architecture viewpoints and model kinds (ii). Let us analyze our definitions in order to identify this unifying construct.

(A) In Section §3.2.1 (ii) we defined the semantic domain of our semantic functions as the power set of all conceivable modeling artifacts. The modeling artifacts are *entities* in terms of the Global Model Management approach defined in Section §2.3.

(B) The semantic equations we have defined so far produce *models*, which is the kind of *entity* that conforms the core of model-based approaches. The other kind of *entity*, namely *external entities*, were used in the semantic equations uniquely to define transformations, either as their source (input) or their target (output). These transformations provide interoperability of our model-based interpretation with external (non-model-driven) tools. Then, the modeling artifacts produced by the denotation are *entities* (A), but particularly they are *models*.

(C) We can distinguish two cases in the representation of concept instances. For the instances of the concepts Concern and Stakeholder, we use *model elements*. A *model element* is an instance of a construct defined in a metamodel, and that is defined in a model. In the case of concerns and stakeholders, their concept instances are model elements defined in a *concerns model* and in a *stakeholders model* respectively, as defined in Tables 3.8–(10) and 3.9–(17). For the concept instances that are complex work products, their denotation includes a megamodel (in addition to other required modeling artifacts). This is the case for ArchitectureViewpoint as defined in Table 3.12–(16), ArchitectureView as defined in Table 3.13–(11), ModelKind as defined in Table 3.10–(22) and ArchitectureModel as defined in Table 3.11–(17). We follow this approach consistently in order to provide a single *representative* modeling artifact for each concept instance that is a complex work product.

(D) The association between simple concept instances, namely concerns and stakeholders, is captured by means of a *weaving model*. As we defined in Table 3.9–(7), the weaving model SCWM serves this purpose. The associations between complex concept instances (i.e. those that are complex work products) with simple concept instances are captured using model annotations that are the special case of *weaving model* with a single woven model. This is the case for the concerns framed by an architecture viewpoint and the concerns framed by a model kind, as shown in Tables 3.12–(14) and 3.10–(16) respectively. Weaving models are a special kind of terminal model in which some of its *model elements* are references to model elements in separate models.

(E) In order to capture any association between complex concepts, we consistently follow the approach of including in one's megamodel a representation of the other's megamodel. For instance, the semantic equation for avp $:_\circ$ ArchitectureViewpoint asserts that for every model kind mk aggregated in avp, the megamodel mkMgM in the denotation of mk is represented in the megamodel avpMgM as shown in Table 3.12–(21). We proceed similarly for architecture models as defined in Table 3.11–(24) and for architecture views as defined in Tables 3.13–(14) and 3.13–(16). The *representation* of a megamodel in another megamodel is actually a *model element* that represents the first megamodel in the second megamodel, using the Megamodel construct of the metamodel for megamodels MgMM. Then, for each complex concept instance, there are model elements in its scope (i.e. defined in any of the modeling artifacts of its denotation) that

represents any of the concept instances it is associated to. This association is actually restricted by the direction of the traversal of the conceptual model illustrated in Figure 3.6.

(F) Provided (B), the denotation renders a set of modeling artifacts that are particularly *models*. Provided (C), (D) and (E), the model elements defined in these models are used to capture and represent any concept instance, i.e. any member of (i).

(G) In our model-based interpretation, only *model kinds* can introduce new kinds of constructs to use in an architecture description. As defined in Table 3.10–(18), the denotation of any model kind mk includes the customized metamodel mkMM_ρ that defines the modeling language, i.e. that provides the set of constructs that are available to build architecture models governed by the model kind mk. The denotation of an *architecture viewpoint* avp also includes a metamodel avpMM_ρ as defined in Table 3.12–(15), but it is used only to provide a foundational metamodel for the metamodels of its aggregated model kinds as defined in Table 3.12–(23). Then, in our model-based interpretation, an *architecture view* does not directly represent any aspect of the system-of-interest. An architecture view provides the representation of the system-of-interest by means of its aggregated *architecture models*. As defined in Table 3.11–(15), the denotation of every *architecture model* am includes the terminal model amM that conforms to the metamodel defined by the governing model kind. It is the model amM that serves the purpose of representing some aspects the system's architecture. Such a representation is achieved by means of *model elements* defined in the model. Then, in our model-based interpretation, every new construct is defined in a metamodel of a model kind and any element participating in the representation of some aspect of the system is actually a *model element* defined in a terminal model of an architecture model. Then, *model elements* capture the members of (ii).

(H) Then, provided (F) and (G), we conclude that our model-based interpretation uses *model element* as the unifying construct that represents any *architecture description element* that can participate in a relationship captured by a *correspondence*. Also, every model element is defined in a *terminal model*.

Scope. We define the *scope* of a *correspondence* as the set of potential *architecture description elements* that can be chosen to participate in the *correspondence*. In the standard, *correspondences* are used only in the context of *architecture descriptions* as illustrated in Figure 3.2. Hence, the standard has no need to restrict the scope as correspondences are used at the top-level only. We argue that correspondences are also useful in the context of a single *architecture view*. An architecture view may capture the correspondences between architecture description elements pertaining only to the view, i.e. correspondences between elements of the *architecture models* aggregated in the view. Then, the scope of a correspondence is determined by the containing concept, i.e. an architecture description or an architecture view.

The standard states in [ISO11, Section 4.2.6] that correspondences can be used to define a relation between architecture descriptions. We claim that this is not supported by the conceptual model of the standard as it does not define any association between *architecture descriptions*. Hence, in the context of any ad $:_\circ$ ArchitectureDescription, there is no other architecture description to which establish a correspondence. Then, we argue that the con-

1	$[\![\mathsf{Correspondence} : \circ]\!]$
2	$\stackrel{\text{def}}{=} \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$
3	$\mathsf{CorrespondenceDocMM}_\rho :_\bullet \mathsf{Metamodel},$
4	$\mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
5	$\mathsf{CorrespondenceDocMM}_\rho \lhd \mathsf{MMM}_\rho \quad \wedge$
6	$\mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho$
7	$[\![\mathsf{c} :_\circ \mathsf{Correspondence}]\!]$
8	$\stackrel{\text{def}}{=} [\![\mathsf{Correspondence} : \circ]\!] \ \cup \ [\![\mathsf{c.scope}]\!] \ \cup$
9	$\{\mathsf{cDocM} :_\bullet \mathsf{TerminalModel},$
10	$\mathsf{cWMM}_\rho :_\bullet \mathsf{Metamodel},$
11	$\mathsf{cWM} :_\bullet \mathsf{WeavingModel},$
12	$\mathsf{cMgM} :_\bullet \mathsf{Megamodel}\}$ such that
13	$\mathsf{cDocM} \lhd \mathsf{CorrespondenceDocMM}_\rho \quad \wedge \quad \mathsf{c}_\mu \lesssim \mathsf{cDocM} \quad \wedge$
14	$\mathsf{cWM} \lhd \mathsf{cWMM}_\rho \quad \wedge \quad \mathsf{cWMM}_\rho \lhd \mathsf{MMM}_\rho \quad \wedge$
15	$\mathsf{cWM} \blacktriangleright \mathsf{cMW}\,\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle$ with $1 \le \mathrm{i} \le n, 2 \le n, \mathsf{m}_\mathrm{i} \in [\![\mathsf{c.scope}]\!], \mathsf{m}_\mathrm{i} :_\bullet \mathsf{TerminalModel} \quad \wedge$
16	$\mathsf{cMgM} \lhd \mathsf{MgMM}_\rho$
17	$\nu(\{\mathsf{MMM}_\rho, \mathsf{CorrespondenceDocMM}_\rho, \mathsf{cDocM}, \mathsf{cWMM}_\rho, \mathsf{cWM}, \mathsf{m}_\mathrm{i}\}) \lesssim \mathsf{cMgM}$

Table 3.14: *Semantic equations:* Correspondence.

ceptual model must be refined by introducing association(s) to the *architecture description* concept in order to support correspondences between them.

In terms of our model-based interpretation, the *scope* of a correspondence consists of the potential modeling artifacts of kind *terminal model* which model elements can participate in the *correspondence*. For any concept instance containing correspondences, the potential terminal models are those obtained in the denotation of the concept instance. We have defined the compositional traversal of the conceptual model in a way that coarse-grained concepts are compositionally denoted in terms of fine-grained concepts, as illustrated in Figure 3.6. Associations in the conceptual model depicted in Figure 3.2, mainly those marked as aggregations, were used to distinguish between coarse- and fine-grained. Considering this model-based interpretation of the scope, the semantic equation for an *architecture view* can be extended to include correspondences, provided that these correspondences are restricted to the scope demarcated by the denotation of the architecture view.

Representation. As the standard states in [ISO11, Section A.6], a *correspondence* can be interpreted as the mathematical concept of n-ary relation, and in practice, it is usually captured by means of a table listing the tuples in the relation. In our model-based interpretation, a *correspondence* is denoted by means of a *weaving model* that realizes a *model weaving* on the terminal models participating of the correspondence. The advantage of a denotation in terms of a weaving model is that a purpose-specific metamodel can be defined for the weaving model in order to capture additional information on the relationship.

Table 3.14 defines the semantic equations for the concept Correspondence (1) and its in-

stances (7). The denotation of $\mathsf{Correspondence} : \circ$ is not compositional on any other concept, as defined by the compositional traversal of the conceptual model. It only provides the foundational modeling artifacts that are needed to denote instances of Correspondence: the metametamodel MMM_ρ (2) that determines the technical space, the customized metamodel $\mathsf{CorrespondenceDocMM}_\rho$ (3) and the metamodel for megamodels MgMM_ρ (4).

The denotation of $\mathsf{c} :_\circ \mathsf{Correspondence}$ is compositional on the denotation of the concept Correspondence (8) and on the denotation of the correspondence's scope (8). The scope of the correspondence c, expressed as c.scope in (8) contains all the concept instances that participate in the correspondence c and that are available in the context of the concept instance containing the correspondence. For example, consider that the containing concept instance is $\mathsf{ad} :_\circ \mathsf{ArchitectureDescription}$, then c.scope is the set of concept instances directly or indirectly associated to ad that participates in the correspondence c. While the conceptual model refer to the participating elements as *architecture description elements*, we assume that the syntactic domain is further characterized and the exact set of concept instances participating in the correspondence are known and well-determined. It is important to notice that, as the denotation of concept instances is always compositional on the denotation of the corresponding concepts, the foundational artifacts for the concept instances in the scope are also included in the denotation of c due to $[\![\mathsf{c.scope}]\!]$.

The denotation of $\mathsf{c} :_\circ \mathsf{Correspondence}$ includes the terminal model cDocM (9) to capture administrative information and reference material related to the correspondence. A representation of c, expressed in terms of the constructs defined by the customized metamodel $\mathsf{CorrespondenceDocMM}_\rho$, is defined in cDocM (13).

The denotation of c also includes the customized metamodel cWMM_ρ (10) that defines the modeling language for capturing the relationship. The weaving model cWM (11) captures and preserves the links between the model elements defined in the woven models $\mathsf{m_i}$ using the constructs defined by cWMM_ρ (14). The weaving model cWM realizes the model weaving cMW (15). The woven models $\mathsf{m_i}$ are *terminal models* defined in the *scope* of the correspondence c. As we explained before, these woven models can be any modeling artifact of kind TerminalModel that is produced by the denotation of the containing concept instance. Each woven model $\mathsf{m_i}$ can be actually a *terminal model*, or it can be of one of the special kinds *weaving model* or *megamodel*, as we determined when we analyzed the concept of *architecture description element*.

The metamodel cWMM_ρ can be a general-purpose metamodel for weaving models defining constructs to preserve simple links between model elements in the woven models. This general-purpose metamodel can be reused in more than one correspondence, and usually, it is already available in the supporting modeling environment. However, such a metamodel does not provide constructs to capture additional information pertaining to each link. In order to further characterize the links, practitioners may need to capture administrative information, reference materials, descriptive data, technical properties, classification according to the semantic of the link, among others. By means of a purpose-specific metamodel, the correspondence can preserve any additional information of interest. Then, a purpose-specific metamodel determines a kind of correspondences that can be used to capture actual correspondences on particular terminal models. This notion of *correspondence kind* governing

correspondences is missing in the standard. Conceiving a correspondence kind separately from the correspondences governed by this kind, allows practitioners to separate system-independent architecture knowledge (what information is interesting to capture) from system-dependent architecture knowledge (which links and their information). The standard uses this distinction between system-independent and system-dependent architecture knowledge to separate *architecture viewpoints* from *architecture views* and *model kinds* from *architecture models*. However, the standard is not homogeneous on applying this technique and hence it is not available for *correspondences*. Although the standard defines the concept of *correspondence rule* and states that correspondence rules govern *correspondences*, as we discuss later, *correspondence rules* actually serves a different purpose. In order to keep alignment to the standard's conceptual model, we do not define separate semantic equations for correspondences and correspondence kinds. However, we take advantage of this distinction later when we define the semantics of correspondence rules.

A correspondence is a complex work product involving several modeling artifacts, the denotation of c includes the megamodel cMgM (12) to capture a representation of the whole set of participating artifacts. Particularly, the megamodel includes a representation of each of the terminal models m_i that are related in the correspondence (17).

Correspondence Rules

Correspondence rules are used to enforce relations within an architecture description. A correspondence rule represents a rule or assertion that must hold on interrelated *architecture description elements*. The standard states that a *correspondence rule* governs *correspondences*, as illustrated in Figure 3.3. The purpose of a *correspondence rule* is not to define the structure of the information that must be preserved by the *correspondences* it governs. The purpose of a *correspondence rule* is to characterize or define a relation between architecture description elements, as captured by the *defines-relation-about* association illustrated in Figure 3.3. As we explained in Section §3.1.1, this association is not part of the standard but it is defined in the standard's resources site [ISO14]. While a *correspondence* is used to explicitly capture and preserve the tuples in a relation, a *correspondence rule* is used to capture properties that are expected to be true on *correspondences*. This is the connotation that the standard attributes to the term govern in this particular scenario. Then, a *correspondence rule* is a notoriously different concept of what we introduce as *correspondence kind*.

We claim that a *correspondence* defines a relation between architecture description elements by extension, in the sense of set theory. A correspondence captures the relation by explicitly enumerating every tuple in the relation. However, a *correspondence rule* defines the relation by specifying its intention, again in the sense of set theory. A correspondence rule defines the assertions that represent or define the valid tuples in the relation. Then, a correspondence rule determines not only if a particular tuple in a correspondence is valid, but also if a particular tuple is missing. For example, consider the *is-implemented-by* relation between components and modules captured in separate models. A *correspondence* consists of tuples of the form $\langle c, m \rangle$ that capture and preserve that the component c is implemented by the module m. A *correspondence rule* may state that every component c is implemented by at least one module m, that every module m implements at least one component c, that

if c is implemented by m then every composing component of c is implemented by a module refining the module m, among others.

Correspondence rules are conceived as reusable architectural assets as they are captured in *architecture frameworks* and *architecture description languages*, as illustrated in Figures 3.4 and 3.5 respectively. However, in the context of architecture frameworks and architecture description languages, there are no actual *architecture models* on which define any *correspondence* or on which enforce any *correspondence rule*. Architecture models are part of *architecture descriptions* uniquely, as conceived by the conceptual model defined in the standard. To conceive a *correspondence rule* as a system-independent reusable asset, it must be defined only in terms of system-independent reusable assets (architecture viewpoints and model kinds), and not in terms of system-dependent assets (architecture views and architecture models). This understanding of correspondence rules is aligned to the standard's perspective as in the context of architecture frameworks and architecture description languages, only system-independent reusable assets participate. However, as *correspondence rules* rule on *correspondences*, we need a mechanism to distinguish what in correspondences are system-independent and what is system-dependent. As we introduced before when we analyzed correspondences, *correspondence kinds* capture the system-independent knowledge on correspondences.

In the context of *architecture descriptions*, however, we distinguish two different roles for *correspondence rules*. The architecture description captures the available correspondence rules defined as system-independent reusable assets. These assets are part of the architecture description when it uses and adheres to architecture frameworks or architecture description languages. However, the availability of a correspondence rule does not imply that it must be applied in the architecture description. It is up to the practitioner to decide whether a correspondence rule is applied or not, and to determine on which system-dependent assets (correspondences and architecture models for instance) it must be applied. It is important to notice that correspondence rules are expressed in terms of system-independent assets, such as correspondence kinds and model kinds, and in an architecture description there might be more than one architecture model governed by the same model kind. Then, the architecture description also captures the selected correspondence rules and indicates on which correspondences (defined in the architecture description) it is actually applied. This distinction between *correspondence rule* and *selected correspondence rule* is not considered in the standard. However, we claim that this distinction is important as the selection of the correspondence rules to apply in part of the architecture decisions that shape the architecture. In our model-based interpretation for correspondence rules, the case of *selected correspondence rules* is treated as part of the denotation of *architecture descriptions*, that we discuss later.

Scope. We define the *scope* of a *correspondence rule* as the set of potential *architecture description elements* that can be chosen to participate in the *correspondences* ruled by the *correspondence rule*. According to the conceptual model of the standard, correspondence rules are used in the context of *architecture descriptions*, *architecture frameworks* and *architecture description languages*. However, as we discussed before when we analyzed *architecture viewpoints*, the standard states in [ISO11, Section B.2.7] that correspondence rules can be used to define 'cross model' constraints on *architecture models* governed by the *model kinds*

$$
\begin{array}{rl}
1 & [\![\mathsf{CorrespondenceRule} : \circ]\!] \\
2 & \quad \stackrel{\text{def}}{=} \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \\
3 & \qquad \mathsf{CorrespondenceRuleDocMM}_\rho :_\bullet \mathsf{Metamodel}, \\
4 & \qquad \mathsf{TMM}_\rho :_\bullet \mathsf{Metamodel}, \\
5 & \qquad \mathsf{ProblemMM}_\rho :_\bullet \mathsf{Metamodel}, \\
6 & \qquad \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\} \text{ such that} \\
7 & \qquad \mathsf{CorrespondenceRuleDocMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge \\
8 & \qquad \mathsf{TMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge \\
9 & \qquad \mathsf{ProblemMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge \\
10 & \qquad \mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho \\
11 & [\![\mathsf{cr} :_\circ \mathsf{CorrespondenceRule}]\!] \\
12 & \quad \stackrel{\text{def}}{=} [\![\mathsf{CorrespondenceRule} : \circ]\!] \ \cup \ [\![\mathsf{cr.scope}]\!] \ \cup \\
13 & \qquad \{\mathsf{crDocM} :_\bullet \mathsf{TerminalModel}, \\
14 & \qquad \mathsf{crTM}_\rho :_\bullet \mathsf{TransformationModel}, \\
15 & \qquad \mathsf{crMgM} :_\bullet \mathsf{Megamodel}\} \text{ such that} \\
16 & \qquad \mathsf{crDocM} \lhd \mathsf{CorrespondenceRuleDocMM}_\rho \ \wedge \ \mathsf{cr}_\mu \lessapprox \mathsf{crDocM} \ \wedge \\
17 & \qquad \mathsf{crTM} \lhd \mathsf{TMM}_\rho \ \wedge \\
18 & \qquad \mathsf{crTM}_\rho \blacktriangleright \mathsf{crT}\,\langle \mathsf{MM}_1, \ldots, \mathsf{MM}_n \twoheadrightarrow \mathsf{ProblemMM}_\rho \rangle \\
19 & \qquad\quad \text{with } 1 \le \mathrm{i} \le n, 2 \le n, \mathsf{MM}_\mathrm{i} \in [\![\mathsf{cr.scope}]\!], \mathsf{MM}_\mathrm{i} :_\bullet \mathsf{Metamodel} \ \wedge \\
20 & \qquad \mathsf{crMgM} \lhd \mathsf{MgMM}_\rho \ \wedge \\
21 & \qquad \forall \mathrm{i}, 1 \le \mathrm{i} \le n.\ \nu(\mathsf{MM}_\mathrm{i}) \lessapprox \mathsf{crMgM} \ \wedge \\
22 & \qquad \nu(\{\mathsf{MMM}_\rho, \mathsf{CorrespondenceRuleDocMM}_\rho, \mathsf{crDocM}, \\
23 & \qquad\quad \mathsf{TMM}_\rho, \mathsf{ProblemMM}_\rho, \mathsf{crTM}_\rho\}) \lessapprox \mathsf{crMgM}
\end{array}
$$

Table 3.15: *Semantic equations:* CorrespondenceRule.

of the *architecture viewpoint*. Then, the scope of a correspondence rule is determined by its containing instance, which is actually an instance of any of the previous concepts.

In terms of our model-based interpretation, the scope consists of the potential modeling artifacts defined in the denotation of the containing concept instance, and that is restricted by the compositional traversal of the conceptual model.

Representation. As the standard states in [ISO11, Section A.6], a *correspondence rule* can be interpreted mathematically as the intentional definition of an n-ary relation, and in practice, it is usually captured by means of an assertion expressed in natural language or seldom in semi-formal or formal languages. In our model-based interpretation, a *correspondence rule* is denoted by means of a *transformation model*. The transformation model captures the knowledge on how to process the *terminal models* participating in *correspondences* to determine whether the correspondence rule holds or not.

Table 3.15 defines the semantic equations for the concept CorrespondenceRule (1) and its instances (11). The denotation of CorrespondenceRule : ○ is not compositional on any other concept, as defined by the compositional traversal of the conceptual model. It only pro-

vides the foundational modeling artifacts to denote instances of CorrespondenceRule: the metametamodel MMM_ρ (2) that determines the technical space, the customized metamodel $\mathsf{CorrespondenceRuleDocMM}_\rho$ (3), the metamodel TMM_ρ (4) that determines the model transformation language, the metamodel $\mathsf{ProblemMM}_\rho$ (5) that determines the constructs for providing detailed error information, and the metamodel for megamodels MgMM_ρ (6).

The denotation of cr :$_\circ$ CorrespondenceRule is compositional on the denotation of the concept CorrespondenceRule (12) and on the denotation of the scope of the correspondence rule (12). As is the case for *correspondences*, we assume that the syntactic domain is further characterized and the exact set of concept instances participating in the correspondence rule are known and well-determined. The denotation of cr includes the terminal model crDocM (13) to capture administrative information and reference material related to the correspondence rule. A representation of cr, expressed in terms of the customized metamodel $\mathsf{CorrespondenceRuleDocMM}_\rho$ is defined in crDocM (16).

The denotation of cr includes the transformation model crTM_ρ (14) that realizes the transformation crT (18) and that captures the knowledge on how to validate the assertion of the correspondence rule. We claim that a transformation model is the appropriate representation of the assertion for three reasons. First, a transformation is not attached to any particular terminal model. It is defined to work on any terminal model provided that they conforms to the reference models specified as the parameters of the transformation. The In parameters of the transformation crT are the metamodels MM_i which are available in the scope of the correspondence rule (19). As we discuss later, while a correspondence rule defines an assertion that can be potentially applied, a *selected correspondence rule* defines the application of this assertion to a particular case of terminal models participating in the *correspondence* being ruled. Second, as transformation models are executable, we provide the ability to compute the assertion to determine whether is holds or not on a particular set of terminal models. As we discussed before when we analyzed the constraints on *model kinds*, we use the customized metamodel $\mathsf{ProblemMM}_\rho$ to allow a transformation model to provide detailed error information on the violations of the assertion it encodes. Third, by using a transformation we allow practitioners to define relations that need not to be captured explicitly (i.e. by means of a *correspondence*). In this particular case, there is no *correspondence* and hence no weaving model representing the correspondence. The transformation model uses its own mechanism for matching model elements defined in the participating terminal models (e.g. their name) and validates the assertion according to this matching.

The denotation of cr includes the megamodel crMgM (15) to capture a representation of all the modeling artifacts participating in the definition of the correspondence rule (21–23).

Architecture Descriptions

An *architecture description* is a complex work product that embodies the representation of the *architecture* of a *system-of-interest.* It identifies and captures the set of architecturally significant concerns and the set of stakeholders holding those concerns. An architecture description is organized in terms of *architecture views*, each representing the architecture of the system-of-interest from a particular perspective. These perspectives are defined and

documented by means of *architecture viewpoints* which provide the conventions for building architecture views. Architecture views provide practitioners of a decomposition mechanism for structuring and modularizing the architecture description. An architecture description also captures *correspondences* between the architecture description elements used throughout the architecture description. Correspondences provide the composition mechanism.

Table 3.16 defines the semantic equations for the concept ArchitectureDescription (1) and its instances (11). The denotation of ArchitectureDescription : ○ is compositional on the denotation of its associated concepts, according to the compositional traversal of the conceptual model illustrated in Figure 3.6. Additionally, the denotation of the concept includes the customized metametamodel MMM_ρ (6) that determines the technical space, the customized metamodel $\mathsf{ArchitectureDescriptionDocMM}_\rho$ (7), and the metamodel for megamodels MgMM_ρ (8). The denotation of ad :$_\circ$ ArchitectureDescription is compositional on the denotation of the concept ArchitectureDescription (12) and on the denotation of all the related concept instances according to the compositional traversal of the conceptual model.

The denotation of ad :$_\circ$ ArchitectureDescription includes the terminal model adDocM (17) that captures the documentation of the architecture description. The documentation includes mainly all the information that is not actually captured by the set of modeling artifacts aggregated by the architecture description. The documentation usually includes the customer's name, goals and expectations, the contextual business case, the system's name, goals and technological context, an overview of the main problem to be addressed and of the solution proposed, project information, reference materials, bibliographic references, a glossary, among others. The customized metamodel $\mathsf{ArchitectureDescriptionDocMM}_\rho$ (7) defines the language for capturing and expressing this documentation (20). Even though a metamodel providing a simple reduced set of text-oriented constructs can be used, a customized well-structured fine-grained set of constructs would enable accurate and homogeneous documentations that can be replicated in separate development projects. These constructs can be inspired in documentation practices such as SEI's Views & Beyond [CBB+10], but they should be accommodated to fit the particular scenario, practice and skills or the development organization.

The denotation of ad also includes a megamodel adMgM (19) to capture the complete representation of the modeling artifacts that participate in the architecture description. The megamodel includes a representation of those modeling artifacts directly produced by the semantic equations of (1) and (11), and some of the modeling artifacts obtained by the compositional application of the semantic function. As we proceeded before, the relation of the architecture description ad to other complex work products is captured by including in adMgM only the *representative* megamodel of those work products. In what follows, we discuss different aspects of the denotation of architecture descriptions and the denotation of its composed concept instances.

Concerns & stakeholders. The denotation of ad :$_\circ$ ArchitectureDescription is compositional on the denotation of its concept (12), which in turn, is compositional on the denotation of Concern : ○ (3). As we defined in Table 3.8–(1), the denotation of Concern : ○ includes a terminal model CM, that we called *concerns model*, that captures and preserves the set of significant concerns that are identified in a particular *working context*. Thus, being the

1 $\llbracket \mathsf{ArchitectureDescription} : \circ \rrbracket$

2 $\stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureFramework} : \circ \rrbracket \;\cup\; \llbracket \mathsf{ArchitectureDescriptionLanguage} : \circ \rrbracket \;\cup$

3 $\llbracket \mathsf{Concern} : \circ \rrbracket \;\cup\; \llbracket \mathsf{Stakeholder} : \circ \rrbracket \;\cup$

4 $\llbracket \mathsf{ArchitectureViewpoint} : \circ \rrbracket \;\cup\; \llbracket \mathsf{ArchitectureView} : \circ \rrbracket \;\cup$

5 $\llbracket \mathsf{Correspondence} : \circ \rrbracket \;\cup\; \llbracket \mathsf{CorrespondenceRule} : \circ \rrbracket \;\cup$

6 $\{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$

7 $\mathsf{ArchitectureDescriptionDocMM}_\rho :_\bullet \mathsf{Metamodel},$

8 $\mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that

9 $\mathsf{ArchitectureDescriptionDocMM}_\rho \triangleleft \mathsf{MMM}_\rho \;\wedge$

10 $\mathsf{MgMM}_\rho \triangleleft \mathsf{MMM}_\rho$

11 $\llbracket \mathsf{ad} :_\circ \mathsf{ArchitectureDescription} \rrbracket$

12 $\stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureDescription} : \circ \rrbracket \;\cup$

13 $\llbracket \mathsf{ad.usedAFs} \rrbracket \;\cup\; \llbracket \mathsf{ad.usedADLs} \rrbracket \;\cup$

14 $\llbracket \mathsf{ad.concerns} \rrbracket \;\cup\; \llbracket \mathsf{ad.stakeholders} \rrbracket \;\cup$

15 $\llbracket \mathsf{ad.architectureViewpoints} \rrbracket \;\cup\; \llbracket \mathsf{ad.architectureViews} \rrbracket \;\cup$

16 $\llbracket \mathsf{ad.correspondences} \rrbracket \;\cup\; \llbracket \mathsf{ad.correspondenceRules} \rrbracket \;\cup$

17 $\{\mathsf{adDocM} :_\bullet \mathsf{TerminalModel},$

18 $\mathsf{adscrMgM}^{\mathsf{i}} :_\bullet \mathsf{Megamodel},$

19 $\mathsf{adMgM} :_\bullet \mathsf{Megamodel}\}$ for $1 \le \mathrm{i} \le n, 0 \le n$ such that

20 $\mathsf{adDocM} \triangleleft \mathsf{ArchitectureDescriptionDocMM}_\rho \;\wedge\; \mathsf{ad}_\mu \lesssim \mathsf{adDocM} \;\wedge$

21 $\mathsf{adMgM} \triangleleft \mathsf{MgMM}_\rho \;\wedge$

22 $\forall \mathsf{af} \in \mathsf{ad.usedAFs}.\; \nu(\mathsf{afMgM}) \lesssim \mathsf{adMgM} \;\wedge$

23 $\forall \mathsf{adl} \in \mathsf{ad.usedADLs}.\; \nu(\mathsf{adlMgM}) \lesssim \mathsf{adMgM} \;\wedge$

24 $\forall \mathsf{avp} \in \mathsf{ad.architectureViewpoints}.\; \nu(\mathsf{avpMgM}) \lesssim \mathsf{adMgM} \;\wedge$

25 $\forall \mathsf{av} \in \mathsf{ad.architectureViews}.\; \nu(\mathsf{avMgM}) \lesssim \mathsf{adMgM} \;\wedge$

26 $\forall \mathsf{c} \in \mathsf{ad.correspondences}.\; \nu(\mathsf{cMgM}) \lesssim \mathsf{adMgM} \;\wedge$

27 $\forall \mathsf{cr} \in \mathsf{ad.correspondenceRules}.\; \nu(\mathsf{crMgM}) \lesssim \mathsf{adMgM} \;\wedge$

28 $\mathsf{adscrMgM}^{\mathsf{i}} \triangleleft \mathsf{MgMM}_\rho \;\wedge\; \nu(\mathsf{adscrMgM}^{\mathsf{i}}) \lesssim \mathsf{adMgM} \;\wedge$

29 $\nu(\mathsf{crMgM}^{\mathsf{i}}) \lesssim \mathsf{adscrMgM}^{\mathsf{i}}$

30 with $\mathsf{crMgM}^{\mathsf{i}}$ the megamodel of the applied correspondence rule $\mathsf{cr}^{\mathsf{i}} \;\wedge$

31 $\nu(m^{\mathrm{i}}_j) \lesssim \mathsf{adscrMgM}^{\mathsf{i}}$ for $1 \le j \le n^{\mathrm{i}}$

32 with $\mathsf{crT}^{\mathsf{i}}$ the transformation of the applied correspondence rule cr^{i},

33 n^{i} is the number of In parameters of $\mathsf{crT}^{\mathsf{i}}$,

34 $\mathsf{MM}^{\mathsf{i}}_j$ is the j-th parameter of $\mathsf{crT}^{\mathsf{i}}$,

35 $\mathsf{m}^{\mathsf{i}}_j :_\bullet \mathsf{TerminalModel}, \quad \mathsf{m}^{\mathsf{i}}_j \triangleleft \mathsf{MM}^{\mathsf{i}}_j; \;\wedge$

36 $\nu(\{\mathsf{MMM}_\rho, \mathsf{MgMM}_\rho, \mathsf{TMM}_\rho, \mathsf{ArchitectureDescriptionDocMM}_\rho, \mathsf{adDocM},$

37 $\mathsf{CMM}_\rho, \mathsf{CM}, \mathsf{CMergeTM}_\rho,$

38 $\mathsf{SMM}_\rho, \mathsf{SM}, \mathsf{SCWMM}_\rho, \mathsf{SCWM}, \mathsf{SMergeTM}_\rho\}) \lesssim \mathsf{adMgM}$

Table 3.16: *Semantic equations:* ArchitectureDescription.

construction of ad the working context, the denotation of ad includes the terminal model CM. Moreover, the denotation of ad is compositional on the denotation of each identified concern c in ad.concerns (14). As defined in Table 3.8–(10), the denotation of a concern c enforces the assertion that c is represented in the *concerns model* CM available in the working context. Then, the denotation of ad :$_{\circ}$ ArchitectureDescription includes a *concerns model* CM that captures and preserves all the architecturally significant concerns ad.concerns identified in the construction of ad.

Analogously, and according to the definition of the semantic function for *stakeholders* in Table 3.9, the denotation of ad includes a *stakeholders model* SM that captures and preserves all the stakeholders holding concerns on the architecture. Also, it includes the weaving model SCWM, defined in Table 3.9–(7), that captures the relationship between the concerns captures in CM and the stakeholders captured in SM.

The concepts Concern and Stakeholder are considered as simple concepts in the sense that they are not understood as a work product by themselves. Each instance of these concepts is actually represented by means of a model element in a corresponding terminal model for this purpose, as defined in Tables 3.8–(10) and 3.9–(17). The corresponding terminal models are part of the foundational modeling artifacts provided by the denotation of the concepts themselves. It is important to notice that this is only the case for these two simple concepts and differ from the cases of every other concept we have analyzed so far. The reason of this lack of homogeneity is that concerns and stakeholders are actually real-world concepts that need to be represented in the architecture description, as we discussed in the introduction of Section §3.2. In the case of complex concepts which are considered as complex work products pertaining the architecture description practice, we have included a *representative* megamodel for each concept instance that preserves the representation of the complete set of participating modeling artifacts. There is no *representative* megamodel for a concern or stakeholder instance. Then, the concept instance conforming the working context (for instance the *architecture description* ad) has the responsibility to preserve the complete set of artifacts required to capture its concerns and its stakeholders. It is important to notice that this is intentional in our specification. By including a representation of the set of modeling artifacts for concerns and stakeholders in the megamodel of the working context, we intended to explicitly represent the fact that these artifacts capture the concerns and stakeholders of the working context. Then, the megamodel adMgM includes the representation of the modeling artifacts in the denotation of Concern : ○ (37) and Stakeholder : ○ (38).

Architecture viewpoints & views. The denotation of ArchitectureDescription : ○ is compositional on both the denotation of ArchitectureViewpoint : ○ and the denotation of ArchitectureView : ○ (4), according to the compositional traversal of the conceptual model illustrated in Figure 3.6. In turn, the denotation of ad :$_{\circ}$ ArchitectureDescription is compositional on the denotation of its aggregated architecture viewpoints and architecture views (15), expressed as ad.architectureViewpoints and ad.architectureViews respectively. As we discussed before, the compositionality makes the denotation of ad to include all the modeling artifacts that are required to denote the related concept instances. However, it does not capture the actual relationship (association) between ad and its aggregated architecture viewpoints and architecture views.

To this end, we follow the same approach as before to represent the relationship between complex concept instances. We include in the *representative* megamodel adMgM of the architecture description ad, the representation of the *representative* megamodel avpMgM of each architecture viewpoint avp $\in$ ad.architectureViewpoints (24), and analogously, the *representative* megamodel avMgM of each architecture view av $\in$ ad.architectureViews (25). The representative megamodel for any architecture viewpoint avp and for any architecture view av were defined in Table 3.12–(16) and Table 3.13–(11) respectively.

Correspondences. The denotation of ArchitectureDescription : ○ is compositional on the denotation of Correspondence : ○ (5) according to the compositional traversal of the conceptual model. In turn, the denotation of ad $:_\circ$ ArchitectureDescription is compositional on the denotation of its aggregated correspondences (16). These correspondences capture n-ary relations between the architecture description elements defined in the context of ad. As we defined in Table 3.14, we conceive a *correspondence* as a complex work product denoted by several modeling artifacts. Then, the denotation of each c $\in$ ad.correspondences includes a megamodel cMgM that represents the whole set of modeling artifacts participating in the denotation of c. In order to capture the association, we include in adMgM a representation of each cMgM. As we discussed before, the notion of *correspondence kind* has no first-class representation in the conceptual model and then, we provided no semantic equation for it. The denotation of each correspondence kind is embedded in the denotation of its governed correspondence. Particularly, for any c $:_\circ$ Correspondence, its denotation includes the customized metamodel cWMM$_\rho$ that specifies the modeling language for capturing the n-ary relation, as defined in Table 3.14–(10). This metamodel is actually the denotation of the correspondence kind.

When we discussed *correspondence rules* before, we distinguished two separate meanings for the concept. First, a *correspondence rule* is conceived as a system-independent reusable asset that provides an intentional definition of an n-ary relation on architecture description elements. This meaning of correspondence rules is captured in the semantic equations of Table 3.15. Correspondence rules are captured and defined in *architecture frameworks* and *architecture description languages*. When the architect decides to use any particular framework or description language, its sets of correspondence rules are made available for the architect to use, i.e. to be applied to concrete cases of architecture description elements. In addition, an architect may decide to define correspondence rules that are specific to the architecture being described. In this case, the correspondence rules are not defined in frameworks or description languages, but are directly captured in the architecture description. In our model-based interpretation, we intentionally understand the association between ArchitectureDescription and CorrespondenceRule in the conceptual model in Figure 3.2 as the set of correspondence rules directly defined in the architecture description. In other words, given ad $:_\circ$ ArchitectureDescription, ad.correspondenceRules does not include any correspondence rule defined in the frameworks or description languages being used by ad. We explain later how these excluded correspondence rules are made available to the context of the architecture description. Then, to capture the correspondence rules directly defined by an architecture description, the denotation of ArchitectureDescription : ○ is compositional on the denotation of CorrespondenceRule (5) and the denotation of ad $:_\circ$ ArchitectureDescription is compositional on the denotation of ad.correspondenceRules (16). A representation of the representative megamodel of each correspondence rule is included in adMgM (27).

The second meaning for the *correspondence rule* concept was introduced before as *selected correspondence rules*. A *selected correspondence rule* represents the fact that an available *correspondence rule* (in the first meaning of the concept) was selected by the architect to rule on a particular set of architecture description elements. While an available correspondence rule is system-independent and does not refer to any particular architecture description elements, a selected correspondence rule determines which are those particular architecture description elements on which the available correspondence rule must be applied. Noticeably, the same available correspondence rule can be selected and applied more than once in order to rule on different architecture description elements. As we analyzed before when we discussed the denotation of *correspondences*, in our model-based interpretation *model element* is the unifying construct for the concept of *architecture description element* and all the model elements that might be ruled by a correspondence rule are defined in *terminal models*. Then, as we defined in Table 3.15–(14), the denotation of any $\mathsf{cr} :_\circ \mathsf{CorrespondenceRule}$ includes a transformation model crTM_ρ that specifies how the rule is to be validated, and that is parameterized by a set of *metamodels* available in the working context. These metamodels are named MM_i in Table 3.15–(18). As a consequence, the denotation of the *selected correspondence rule* of an *architecture description* must determine which are the actual *terminal models*, that conforms to those *metamodels*, on which the correspondence rule must be validated. In other words, which are the terminal models to input the transformation model crTM_ρ for validation.

As the conceptual model does not distinguish between *correspondence rule* and *selected correspondence rule*, there is not a separate association for each case. In the conceptual model, the association between ArchitectureDescription and CorrespondenceRule is also representing this relationship. In our model-based interpretation, we assume that for any $\mathsf{ad} :_\circ \mathsf{ArchitectureDescription}$, it is known and can be determined the possibly empty set of selected correspondence rules $\{\mathsf{adscr}^1, \ldots, \mathsf{adscr}^n\}$ for $0 < \mathrm{i} \le n, n \in \mathbb{N}$. Also, we assume that every selected correspondence rule $\mathsf{adscr}^\mathrm{i}$ determines, identifies or refers to the available correspondence rule cr^i to be applied. For each $\mathsf{adscr}^\mathrm{i}$, we include a megamodel $\mathsf{adscrMgM}^\mathrm{i}$ (18) to denote the selected correspondence rule and that captures the representation of all the modeling artifacts pertaining the application of cr^i to the particular case of $\mathsf{adscr}^\mathrm{i}$. Each megamodel $\mathsf{adscrMgM}^\mathrm{i}$ is represented in the representative megamodel adMgM of the architecture description ad (28). One might argue that a *transformation record* would be a more suitable denotation for each selected correspondence rule $\mathsf{adscr}^\mathrm{i}$, as the denotation of the correspondence rule cr^i to be applied includes the transformation crT^i, as defined in Table 3.15–(18). As we explained in Section §2.3.1, a *transformation record* captures the fact that a transformation was executed and that it produced particular output entities. It is an after-the-fact construct and requires the output entities to exists in the model repository. Moreover, by using a transformation record we would miss the relationship to the *representative* megamodel of the correspondence rule, as the transformation record refers directly to the transformation. By means of a megamodel we are actually capturing the set of modeling artifacts that must be used to execute the transformation. It is a before-the-fact construct defining the capability of execution, and that does not require the output entities to be actually captured and preserved in the model repository. The megamodel $\mathsf{adscrMgM}^\mathrm{i}$ includes a representation of the representative megamodel of the applied correspondence rule $\mathsf{crMgM}^\mathrm{i}$ (29–30) and every *terminal model* m_j on which the correspondence rule is to be applied (31–35). As we discussed before, m_j may be the weaving model of a *correspondence* conforming to the metamodel of its *correspondence kind*, or any terminal model woven by this weaving model. As it must be

possible to use the terminal models m_j as parameter values to the transformation crT defined in the denotation of the correspondence rule, each m_j must conform to the metamodel MM_j (35) defined as a parameter of crT (32).

Consistency. Achieving consistency among architecture views and architecture models is one of the most challenging and difficult problems in the software architecture field [CBB+10]. Quoting the standard in [ISO11, Section 5.7.1], "while consistent architecture descriptions are to be preferred, it is sometimes infeasible or impractical to resolve all inconsistencies for reasons of time, effort, or insufficient information." An architecture description must record any known inconsistencies across architecture views and architecture models. Recording inconsistencies improves the understanding of the architecture description. This record warns stakeholders of known existing issues in the architecture description. When a stakeholder detects an issue that is not recorded, it tends to be confusing for the stakeholders as they have no way to determine whether they are facing an inconsistency in the description or they are misunderstanding it.

We classify inconsistencies in architecture descriptions in two kinds: structural and non-structural. An structural inconsistency is the fact that the architecture description, or any of its aggregated work products, fail to adhere to their corresponding governing definitions. Sources of structural inconsistencies are: (i) an *architecture description* fails to meet every adherence requirement imposed by the *architecture frameworks* and/or *architecture description languages* it uses, (ii) an *architecture view* fails to meet the conventions and restrictions imposed by its governing *architecture viewpoint*, (iii) an *architecture model* fails to meet the conventions and restrictions imposed by its governing *model kind*, and (iv) a *correspondence* fails to meet the conventions and restrictions of its governing *correspondence kind*. The adherence to a governing definition is manifested in two different levels of abstraction. First, a governing definition imposes the number and kind of work products (artifacts) that can be used. For example, an *architecture framework* imposes the *architecture viewpoints* that must be defined in an *architecture description*. Also, an *architecture viewpoint* imposes that a single *architecture view* governed by this viewpoint must be part of the *architecture description*, and restricts the kind of *architecture models* that can be used in this *architecture view*. Second, a governing definition imposes the well-formedness rules on the internal organization of a work product (artifact). For example, a *model kind* imposes the language constructs that can be used by *architecture models* of this model kind.

A non-structural inconsistency is the fact that some architecture description elements populating the *architecture models* break some of the rules that the architect had decided to enforce. As we discussed before, *architecture views* and *architecture models* provide a decomposition mechanism for architecture descriptions. Provided that they refer to different aspects of the same system, the architecture description elements defined within these views and models are interrelated. *Correspondences* provide a composition mechanism by explicitly capturing these relationships within an architecture description. Whether the architecture description elements and the captured correspondences among them are consistent or not, depends on the *correspondence rules* that the architect had decided to enforce on those elements. Every *selected correspondence rule* imposes an assertion that the architect expects to hold on the architecture description. Architecture description elements breaking these

selected assertions are non-structural inconsistencies. In the context of a single *architecture model*, the set of selected constraints defines the assertions that the architect expects to hold on the architecture model. Elements breaking these assertions are also non-structural inconsistencies.

We define our model-based interpretation to cope with *inconsistencies* in a way that they do not need to be captured explicitly. Depending on the kind of inconsistency, they are either avoided by controlling the design activity or automatically detected and reported by the supporting modeling environment.

Structural consistency on the number and kind of work products participating in an architecture description is achieved by controlling the architecture design activity. Provided an incremental approach to architecture design, in which an architecture description is developed from scratch by successive increments (updates), the consistency is preserved by forcing each update not to break the adherence relationships of the participating work products. Chapter §4 defines such an incremental approach and formally specifies the available updates and their effect, in order to ensure the preservation of consistency.

Structural consistency on the well-formedness of work products is automatically validated by the supporting modeling platform. In our semantics specification, we systematically used *metamodels* to define the modeling languages that are used to build the terminal models. This was the case for *model kinds* and *correspondence kinds*, whose metamodel provides the constructs and rules to define well-formed *architecture models* and *correspondences*. The underlying modeling environment is capable of validating whether the *conforms-to* relation between a *model* and its *reference model* is being satisfied or not. The supporting tools generally prevent ill-formed models to be constructed (for instance, model editors and model transformation engines ensures that). However, even if an ill-formed model is actually stored in a model repository, the supporting modeling environment is able to detect and report the conflicting model elements. Then, by means of metamodels, our model-based interpretation either avoid or at least provide the automatic detection of ill-formed models.

Non-structural consistency is also automatically validated in the context of our model-based interpretation. In our semantics specification, we systematically used *transformations* to capture the knowledge on how to validate if a given rule or assertion holds on a model or set of models. For instance, the denotation of *architecture models* includes a set of transformations that captures the *selected constraints* that rules the architecture model, as defined in Table 3.11–(16). In the case of *correspondences*, the *architecture description* preserves a set of *selected correspondence rules* that are denoted by means of a megamodel that refer to the correspondence rule to be applied (which in turns is denoted by a transformation that validates the assertion as defined in Table 3.15–(14)), and to the set of terminal models on which the assertion must be validated, as defined in (18). As stated in the standard in [ISO11, Section 5.7.3] "for each identified correspondence rule, an architecture description shall record whether the rule holds or otherwise record all known violations." The purpose of these transformations is to automate this task. Then, non-structural inconsistencies can be automatically detected by applying these transformations on the corresponding terminal models. Each transformation produces a *problems model* conforming to $\mathsf{ProblemMM}_{\rho}$, that contains the detailed error information on the inconsistencies detected. When all *problems*

models are empty (i.e. no problem is recorded), then the architecture description has no non-structural inconsistencies.

Used frameworks & description languages. The standard defines in [ISO11, Section 6.2] the requirements that an *architecture description* must meet to *adhere* to an *architecture framework*. This *adherence* relationship mainly entails that *concerns*, *stakeholders*, *architecture viewpoints* and *correspondence rules* defined in the architecture framework are included in the architecture description. However, this *adherence* relationship is not represented in the conceptual model. Moreover, for the case of *architecture description languages*, no *adherence* requirements are specified in the standard and no relationship is captured in the conceptual model. We claim that an analogous relationship for the case of architecture description languages should be defined.

In our model-based interpretation, we consider the *adherence* relationship to capture the architecture frameworks and architecture description languages to which the architecture description adheres to, i.e. those used to develop the architecture description. To this end, we make the denotation of the concept ArchitectureDescription to be compositional on the denotation of the concepts ArchitectureFramework and ArchitectureDescriptionLanguage (2). In turn, we make the denotation of ad :$_\circ$ ArchitectureDescription to be compositional on the denotation of set of architecture frameworks (13) and the set of architecture description languages (13) that are used by ad (13). We use ad.usedAFs and ad.usedADLs to express these sets, respectively. Also, in order to capture the relationship, we include a representation of their representative megamodels in adMgM. For each used architecture framework af, its representative megamodel afMgM is represented in adMgM (22). Analogously, for each used architecture description language adl, its representative megamodel adlMgM is represented in adMgM. These megamodels are included in the denotation of af and adl, that we define later in Section §3.2.3.

An architecture description may adhere to more than one architecture framework and/or architecture description language at the same time. However, as stated in a side note in [ISO11, Section 6.2], this would entail a reconciliation between their definitions. As we discuss in Chapter §4, a development company or organization should consider the development of their custom architecture framework(s) by reconciling existing ones, prior to use them in architecture descriptions. By this means, the knowledge on how to reconcile is captured in the new architecture framework and can be reused across development projects. This is also important in those cases when the *architecture description* adheres to no framework. We claim that the development team should consider to define their custom architecture framework separately from the architecture description in order to make the architecture knowledge reusable at least company-wide.

When an *architecture description* adheres to an *architecture framework*, the *architecture viewpoints* defined in the framework must also be defined in the set of architecture viewpoints of the architecture description. Intentionally, the semantic equation for ad :$_\circ$ ArchitectureDescription (11) neither captures nor enforces this restriction. This restriction is an invariant that should be actually defined in the conceptual model. However, the standard cannot define such an invariant as the conceptual model does not capture the adherence

relationship by means of an association. Without the association, there is no way to refer to the architecture frameworks an architecture description adheres to, and consequently, the invariant cannot be expressed either. Having an invariant, our semantic equation would be mapping only consistent architecture descriptions. As a consequence, we allow inconsistent architecture descriptions to be defined. As we discussed before, we achieve and preserve consistency by controlling the architecture design activity. In Chapter §4 we define the atomic updates that can be performed in an architecture description when applying an incremental architecture design method, and we embed in the semantics of these atomic updates the enforcement of the invariant.

Provided then that the denotation of $\mathsf{ad} :_\circ \mathsf{ArchitectureDescription}$ contains in its representative megamodel adMgM the representation of the representative megamodels for the architecture frameworks and architecture description languages ad uses, we can now state which are the available *correspondence rules* in the context of ad. In addition to those directly defined in ad (that we explained before), those captured in the representative megamodels of the used frameworks and description languages are also available. Then, this whole set of *correspondence rules* can be used to define *selected correspondence rules* in ad.

Architectures & Systems-of-interest

The conceptual model of the architecture description practice includes two different kind of concepts: those referring to real-world concepts and those referring to modeling-concepts. We discussed this distinction in the introduction of Section §3.2 and both categories were illustrated in Figure 3.2 by means of separate columns in the diagram. The concept ArchitectureDescription refers to a *representation-of* Architecture, understanding *representation-of* in the sense of Model-Driven Engineering as illustrated in Figure 2.2. While ArchitectureDescription inhabits the modeling-world as it can be conceived as a representation of a *system*, Architecture and SystemOfInterest inhabit the real-world as they refer to the *system* being represented (modeled) by ArchitectureDescription. Then, the concepts we analyzed so far covered every modeling-world concept in the practice of architecture description, according to the conceptual model of the standard. It is expected, however, that defining the denotation of real-world concepts is not similar to define the denotation of modeling-world ones. We faced a similar scenario before, when we considered the denotation of Concern and Stakeholder. However, in these cases, we understood these concepts as the concerns and stakeholders that were identified and captured by an architecture description. In other words, we conceived them as the *representation-of* the real-world concerns and stakeholders, that need to be included in the architecture description.

The compositional traversal of the conceptual model we decided, as illustrated in Figure 3.6, isolates modeling-world concepts from the real-world concepts Architecture and SystemOfInterest. The top-most modeling-world concept ArchitectureDescription represents the complex work product capturing, describing and communicating the architecture of a system-of-interest, and, according to the traversal, it is compositionally defined in terms of the other modeling-world concepts. Then, the denotation of a concept instance of ArchitectureDescription is sufficient to capture the description of an architecture of a system-of-interest. In other words, we have defined the denotational semantics in a way that we expect no

1	$[\![\mathsf{Architecture} : \circ]\!]$	$\stackrel{\text{def}}{=}$	$[\![\mathsf{ArchitectureDescription} : \circ]\!]$
2	$[\![\mathsf{a} :_\circ \mathsf{Architecture}]\!]$	$\stackrel{\text{def}}{=}$	$[\![\mathsf{Architecture} : \circ]\!] \ \cup \ [\![\mathsf{a.architectureDescription}]\!]$
3	$[\![\mathsf{SystemOfInterest} : \circ]\!]$	$\stackrel{\text{def}}{=}$	$[\![\mathsf{Architecture} : \circ]\!]$
4	$[\![\mathsf{s} :_\circ \mathsf{SystemOfInterest}]\!]$	$\stackrel{\text{def}}{=}$	$[\![\mathsf{SystemOfInterest} : \circ]\!] \ \cup \ [\![\mathsf{s.architecture}]\!]$

Table 3.17: *Semantic equations:* Architecture *and* SystemOfInterest.

contribution from the real-world concepts to the denotation of an architecture description, or any of its composed modeling-world concepts. As a consequence, we define the semantic equations for the real-world concepts to capture this fact. It can be argued that there is no need to provide a denotation of these concepts. Even though we agree with this position, we define the semantic equations to achieve a complete coverage of the conceptual model.

Table 3.17 defines the semantic equation for the concept Architecture (1) and its instances (2), and for the concept SystemOfInterest (3) and its instances (4). Following the compositional traversal of the conceptual model, the denotation of SystemOfInterest : ○ is compositional on the denotation of Architecture : ○ (3), which in turn is compositional on the denotation of ArchitectureDescription : ○ (1). The denotation of a *system-of-interest* s is compositional on its concept and on the denotation of its *architecture* (4). In turn, the denotation of an architecture a is compositional on its concept and on the denotation of its *architecture description* (2). As we explained before, none of these concepts or their instances directly contributes with modeling artifacts. Bluntly, the denotation of a system-of-interest s is the denotation of its architecture a, which is actually the denotation of its architecture description.

3.2.3 Semantics of Frameworks and Languages

The denotational semantics of *architecture frameworks* and *architecture description languages* is formalized by means of the same semantic functions we used for *architecture descriptions*. We defined these semantic functions in Section §3.2.1 and we defined the semantic equations for the concepts directly involved in with the concept ArchitectureDescription in Section §3.2.2. In this section, we define the semantic equations or clauses for those concepts of the conceptual model that are directly involved with the ArchitectureFramework and ArchitectureDescriptionLanguage concepts. We reviewed these concepts in Section §3.1.2 and illustrated them in Figures 3.4 and 3.5.

Most of the concepts pertaining to the conceptualization of *architecture frameworks* and *architecture description languages* have already been covered when we defined the semantic equations for ArchitectureDescription in Section §3.2.2. Provided that we are defining more semantic equations for the same semantic functions, the semantic equations for the concepts involved in the conceptualization of *architecture descriptions* also apply to the case of *architecture frameworks* and *architecture description languages*. This is one of the major advantages of defining the semantic functions compositionally. As the denotation of complex (composed) constructs is defined as a composition of the denotation of simpler (composing) constructs, the latter denotations can be used to compose the denotation of more than one

complex construct. Then, the semantic equations that are yet to be defined are only those for the concepts ArchitectureFramework and ArchitectureDescriptionLanguage, and their instances.

The compositional traversal of the conceptual model illustrated in Figure 3.6 need to be extended as it does not consider the conceptualization of architecture frameworks and architecture description languages illustrated in Figures 3.4 and 3.5. We do not change the compositional traversal that we have already defined. We are defining more semantic equations for the same semantic functions and we need the semantic equations we already defined to keep being valid. We simply extend the traversal to include the concepts that are still missing. To this end, we decide that the concepts ArchitectureFramework and ArchitectureDescriptionLanguage are compositional on all the concepts associated to them, which were illustrated in Figures 3.4 and 3.5 respectively.

We define the semantic equations for *architecture frameworks* and *architecture description languages* in what follows.

Architecture Frameworks

An *architecture framework* is a complex work product that captures the conventions and common practices for creating *architecture descriptions*. It is a system-independent reusable asset that define a prefabricated structure and conventions within a specific domain or stakeholder community, that can be used to organize and describe the *architecture* of *systems-of-interest*. An *architecture framework* identifies a set of *stakeholders*, a set of their *concerns*, a set of *architecture viewpoints* framing these concerns, and a set of *correspondence rules* that can be enforced on *architecture views* governed by those *architecture viewpoints*. Figure 3.4 illustrated this conceptualization.

Table 3.18 defines the semantic equations for the concept ArchitectureFramework (1) and its instances (9). The denotation of ArchitectureFramework : ○ is compositional on the denotation of its associated concepts (2–3), according to the compositional traversal we decided before. Also, the denotation includes the customized metametamodel MMM_ρ (4) that determines the technical space, the customized metamodel $\mathsf{ArchitectureFrameworkDocMM}_\rho$ (5), and the metamodel for megamodels MgMM_ρ (6). The denotation of every concept instance af $:_\circ$ ArchitectureFramework is compositional on the denotation of its concept (10) and on the denotation of all related concept instances according to the compositional traversal (11–13).

The denotation of af includes the terminal model afDocM (14) that captures the documentation of the architecture framework which is expressed in terms of the customized metamodel $\mathsf{ArchitectureFrameworkDocMM}_\rho$ (17). This documentation includes every administrative and technical information pertaining the architecture framework that is not being captured by the aggregated concept instances. The denotation of af also includes a megamodel afMgM (16) to capture the complete representation of the modeling artifacts that participate in the architecture framework (20–24).

The denotation of af is compositional on the denotation of its concept (10), which in turn is compositional on the denotation of Concern : ○ (2). Thus, the denotation of af includes

1	$[\![\mathsf{ArchitectureFramework} : \circ]\!]$
2	$\stackrel{\mathrm{def}}{=}$ $[\![\mathsf{Concern} : \circ]\!] \;\cup\; [\![\mathsf{Stakeholder} : \circ]\!] \;\cup$
3	$[\![\mathsf{ArchitectureViewpoint} : \circ]\!] \;\cup\; [\![\mathsf{CorrespondenceRule} : \circ]\!] \;\cup$
4	$\{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$
5	$\mathsf{ArchitectureFrameworkDocMM}_\rho :_\bullet \mathsf{Metamodel},$
6	$\mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
7	$\mathsf{ArchitectureFrameworkDocMM}_\rho \lhd \mathsf{MMM}_\rho \;\wedge$
8	$\mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho$
9	$[\![\mathsf{af} :_\circ \mathsf{ArchitectureFramework}]\!]$
10	$\stackrel{\mathrm{def}}{=}$ $[\![\mathsf{ArchitectureFramework} : \circ]\!] \;\cup$
11	$[\![\mathsf{af.concerns}]\!] \;\cup\; [\![\mathsf{af.stakeholders}]\!] \;\cup$
12	$[\![\mathsf{af.architectureViewpoints}]\!] \;\cup$
13	$[\![\mathsf{af.correspondenceRules}]\!] \;\cup$
14	$\{\mathsf{afDocM} :_\bullet \mathsf{TerminalModel},$
15	$\mathsf{afMM}_\rho :_\bullet \mathsf{Metamodel},$
16	$\mathsf{afMgM} :_\bullet \mathsf{Megamodel}\}$ such that
17	$\mathsf{afDocM} \lhd \mathsf{ArchitectureFrameworkDocMM}_\rho \;\wedge\; \mathsf{af}_\mu \lesssim \mathsf{afDocM} \;\wedge$
18	$\mathsf{afMgM} \lhd \mathsf{MgMM}_\rho \;\wedge$
19	$\forall \mathsf{avp} \in \mathsf{af.architectureViewpoints}.\; \mathsf{avpMM}_\rho \multimap \mathsf{afMM}_\rho \;\wedge$
20	$\forall \mathsf{avp} \in \mathsf{af.architectureViewpoints}.\; \nu(\mathsf{avpMgM}) \lesssim \mathsf{afMgM} \;\wedge$
21	$\forall \mathsf{cr} \in \mathsf{af.correspondenceRules}.\; \nu(\mathsf{crMgM}) \lesssim \mathsf{afMgM} \;\wedge$
22	$\nu(\{\mathsf{MMM}_\rho, \mathsf{MgMM}_\rho, \mathsf{TMM}_\rho, \mathsf{ArchitectureFrameworkDocMM}_\rho, \mathsf{afDocM}, \mathsf{afMM}_\rho,$
23	$\mathsf{CMM}_\rho, \mathsf{CM}, \mathsf{CMergeTM}_\rho,$
24	$\mathsf{SMM}_\rho, \mathsf{SM}, \mathsf{SCWMM}_\rho, \mathsf{SCWM}, \mathsf{SMergeTM}_\rho\}) \lesssim \mathsf{afMgM}$

Table 3.18: *Semantic equations:* ArchitectureFramework.

a *concerns model* CM, as defined in Table 3.8–(3), that captures and preserves the set of significant concerns in the *working context*, i.e. the significant concerns in the context of af. Moreover, according to (11) and Table 3.8–(10), every concern $\mathsf{c} \in \mathsf{af.concerns}$ is represented in CM. As we explained when we discussed the semantic equations for *architecture descriptions* in Section §3.2.2 on page 137, the responsibility of preserving the foundational modeling artifacts denoting the *concerns* of a specific *working context*, is assigned to the denotation of that *working context*. Then, as defined in Table 3.16–(37), the representative megamodel adMgM of $\mathsf{ad} :_\circ \mathsf{ArchitectureDescription}$ contains a representation of the foundational modeling artifacts for capturing the *concerns* identified in the architecture description. Analogously, the representative megamodel afMgM for the architecture framework af contains a representation of the foundational artifacts capturing the significant *concerns* of the architecture framework. By this means, we intend to reflect the fact that each working context preserves its own (separate) version of the foundational modeling artifacts, mainly CM which is the one actually capturing and preserving the concerns. Although we are not explicitly stating it in the semantic equations, each CM is actually understood as it is qualified by the working context: i.e. we would have adCM for an architecture description ad and afCM for an architecture framework af. In order to specify this formally, we need to move the denotation of $\mathsf{Concern} : \circ$ and every $\mathsf{c} :_\circ \mathsf{Concern}$ into the denotation of the working context itself. We preferred not to

pursue this level of formality in order to enhance the readability and understanding of our specification.

We proceed analogously for the case of the set of *stakeholders* to whom the architecture framework is targeted. The foundational modeling artifacts for stakeholders defined in Table 3.9–(1) are part of the denotation of af :∘ ArchitectureFramework, provided (2) and (10). All the stakeholders targeted by af are defined in the *stakeholders model* SM, provided (11) and Table 3.9–(17). Finally, all the foundational modeling artifacts are represented in the representative megamodel afMgM (24).

The denotation of af :∘ ArchitectureFramework is compositional on the denotation of all the *architecture viewpoints* (12) and *correspondence rules* (13) defined by af. As we discussed in Section §3.2.2, in order to capture the relationship between two complex concepts, we include a representation of the *representative* megamodel of the composing concept instance in the *representative* megamodel of the composed concept instance. Thus, for each avp in the set of architecture viewpoints defined by the architecture framework af, the megamodel avpMgM is represented in afMgM (20). Analogously, for each cr in the set of correspondence rules defined by af, the megamodel crMgM is represented in afMgM (21).

When we defined and analyzed the semantic equations for *architecture viewpoints* in Section §3.2.2 on page 121, we discussed the standard's recommendation for *architecture viewpoints* to define a single *metamodel* that unifies the metamodels of the aggregated *model kinds*. We argued that a single metamodel was too restrictive in practice and that the actual recommendation for architecture viewpoints should be to define a base metamodel that must be extended by the metamodels defined in the aggregated model kinds. As a side note in [ISO11, Section B.2.6.2], the standard continues that recommendation adding that "it is often helpful to use a single metamodel to express multiple, related viewpoints (such as when defining an architecture framework)." Using the same line of reasoning we followed for the case of an architecture viewpoint, we claim that it would be helpful that an architecture framework defines a base metamodel for the aggregated architecture viewpoints to extend. This is the recommendation we capture in our semantics specification. To this end, the denotation of af :∘ ArchitectureFramework includes the customized metamodel afMM_ρ (15) that defines the extensible modeling language that provide the foundation for the base languages of the aggregated architecture viewpoints. Then, for every avp ∈ af.architectureViewpoints, it holds that its metamodel avpMM_ρ is an *extension* of the metamodel afMM_ρ (19).

Architecture Description Languages

An *architecture description language* is a complex work product that captures any form of expression for use in *architecture descriptions*. It is a system-independent reusable asset that rigorously provides the conventions for describing some aspects of the *architecture* of a *system-of-interest*. An *architecture description language* defines one or more *model kinds* as a means to frame some *concerns* for its audience of *stakeholders*. These *model kinds* may or may not be organized into *architecture viewpoints* [ISO11, Section 6.3]. Also, an architecture description language defines a set of *correspondence rules* that can be enforced on *architecture views* and *architecture models* governed by the defined *architecture viewpoints*

1	$\llbracket \mathsf{ArchitectureDescriptionLanguage} : \circ \rrbracket$
2	$\stackrel{\text{def}}{=} \llbracket \mathsf{Concern} : \circ \rrbracket \ \cup \ \llbracket \mathsf{Stakeholder} : \circ \rrbracket \ \cup$
3	$\llbracket \mathsf{ModelKind} : \circ \rrbracket \ \cup \ \llbracket \mathsf{ArchitectureViewpoint} : \circ \rrbracket \ \cup \ \llbracket \mathsf{CorrespondenceRule} : \circ \rrbracket \ \cup$
4	$\{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel},$
5	$\mathsf{ArchitectureDescriptionLanguageDocMM}_\rho :_\bullet \mathsf{Metamodel},$
6	$\mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
7	$\mathsf{ArchitectureDescriptionLanguageDocMM}_\rho \triangleleft \mathsf{MMM}_\rho \ \wedge$
8	$\mathsf{MgMM}_\rho \triangleleft \mathsf{MMM}_\rho$
9	$\llbracket \mathsf{adl} :_\circ \mathsf{ArchitectureDescriptionLanguage} \rrbracket$
10	$\stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureDescriptionLanguage} : \circ \rrbracket \ \cup$
11	$\llbracket \mathsf{adl.concerns} \rrbracket \ \cup \ \llbracket \mathsf{adl.stakeholders} \rrbracket \ \cup$
12	$\llbracket \mathsf{adl.modelKinds} \rrbracket \ \cup$
13	$\llbracket \mathsf{af.architectureViewpoints} \rrbracket \ \cup$
14	$\llbracket \mathsf{af.correspondenceRules} \rrbracket \ \cup$
15	$\{\mathsf{adlDocM} :_\bullet \mathsf{TerminalModel},$
16	$\mathsf{adlMgM} :_\bullet \mathsf{Megamodel}\}$ such that
17	$\mathsf{adlDocM} \triangleleft \mathsf{ArchitectureDescriptionLanguageDocMM}_\rho \ \wedge \ \mathsf{adl}_\mu \lesssim \mathsf{adlDocM} \ \wedge$
18	$\mathsf{adlMgM} \triangleleft \mathsf{MgMM}_\rho \ \wedge$
19	$\forall \mathsf{mk} \in \mathsf{adl.modelKinds}.\ \nu(\mathsf{mkMgM}) \lesssim \mathsf{adlMgM} \ \wedge$
20	$\forall \mathsf{avp} \in \mathsf{adl.architectureViewpoints}.\ \nu(\mathsf{avpMgM}) \lesssim \mathsf{adlMgM} \ \wedge$
21	$\forall \mathsf{cr} \in \mathsf{adl.correspondenceRules}.\ \nu(\mathsf{crMgM}) \lesssim \mathsf{adlMgM} \ \wedge$
22	$\nu(\{\mathsf{MMM}_\rho, \mathsf{MgMM}_\rho, \mathsf{TMM}_\rho, \mathsf{ArchitectureDescriptionLanguageDocMM}_\rho, \mathsf{adlDocM},$
23	$\mathsf{CMM}_\rho, \mathsf{CM}, \mathsf{CMergeTM}_\rho,$
24	$\mathsf{SMM}_\rho, \mathsf{SM}, \mathsf{SCWMM}_\rho, \mathsf{SCWM}, \mathsf{SMergeTM}_\rho\}) \lesssim \mathsf{adlMgM}$

Table 3.19: *Semantic equations:* ArchitectureDescriptionLanguage.

and *model kinds*. Figure 3.5 illustrated this conceptualization.

Table 3.19 defines the semantic equations for the concept ArchitectureDescriptionLanguage (1) and its instances (9). The denotation of ArchitectureDescriptionLanguage : $\circ$ is compositional on the denotation of its associated concepts (2–3), according to the compositional traversal we decided before. The denotation also includes the customized metametamodel MMM_ρ (4), the customized metamodel $\mathsf{ArchitectureDescriptionLanguageDocMM}_\rho$ (5), and the metamodel for megamodels MgMM_ρ (6). The denotation of every concept instance adl $:_\circ$ ArchitectureDescriptionLanguage is compositional on the denotation of its concept (10) and on the denotation of all related concept instances according to the compositional traversal (11–14).

Analogously to the case for *architecture frameworks*, the denotation of adl includes a terminal model adlDocM, expressed in terms of $\mathsf{ArchitectureDescriptionLanguageDocMM}_\rho$, that captures administrative and technical information that is not otherwise captured (17). The denotation also includes a megamodel adlMgM (16) to capture a representation of the modeling artifacts participating in the denotation of adl (19–24). Also, the denotation of adl includes a *concerns model* CM and a *stakeholders model* SM that captures the concerns and stakeholders

framed by adl. As we explained before, these terminal models are owned by the denotation of adl and are separate from the corresponding models of other top-level concepts in the conceptual model, such as ArchitectureDescription and ArchitectureFramework.

The denotation of adl $:_\circ$ ArchitectureDescriptionLanguage is compositional on the denotation of all the *architecture viewpoints* (13), if any, that are defined by adl. The *representative* megamodel avpMgM of each avp $\in$ adl.architectureViewpoints is represented in the megamodel adlMgM (20). Each architecture viewpoint avp defines its own set of *model kinds*. Opposed to the cases of *architecture descriptions* and *architecture frameworks*, an *architecture description language* may define *model kinds* that are not aggregated in any architecture viewpoint. The standard does not specify whether the association between ArchitectureDescriptionLanguage and ModelKind in the conceptual model illustrated in Figure 3.5 refers to (i) the model kinds that are directly defined by an architecture description language excluding those defined by the architecture viewpoints, or (ii) to all the model kinds defined by the architecture description language including those defined by the architecture viewpoints. Formally, given an architecture description adl, the standard does not specify whether (i) $\text{adl.architectureViewpoints.modelKinds} \cap \text{adl.modelKinds} = \emptyset$ or (ii) $\text{adl.architectureViewpoints.modelKinds} \subseteq \text{adl.modelKinds}$. An invariant discriminating this ambiguity is required in the conceptual model, although it is not present. None of these cases is more expressive than the other, but (i) provides better modularization and is not error-prone (i.e. by missing a model kind in the association). In the context of our model-based interpretation, we consider (i) to be the intention of the association. By this means, we simplify how to refer to those model kinds directly defined by the architecture description language. As a consequence, the denotation of adl captures in adlMgM a representation of the *representative* megamodel mkMgM of every model kind directly defined by adl (19). Finally, we capture the defined *correspondence rules* in the same way (21).

As we discussed in Section §3.1.2, *architecture description languages* are developed with a set of companion tools to provide visualization, edition, checking and analysis of work products expressed in the languages defined by the architecture description language. We cannot expect that this tool support is actually built in terms of Model-Driven Engineering techniques. This tools works on a tool-specific representation of the work products, possibly the concrete syntax of the languages defined. As the architecture knowledge embedded in these tools is relevant and is available to practitioners in the architecture description practice, our model-based interpretation need to be integrated with them. Noticeably, we have already achieved such an integration. In terms of the conceptual model of the standard, an *architecture description language* does not define any language by itself. It relies on its aggregated *model kinds* to fulfil this purpose. Then, the companion tool support actually work on *architecture models* governed by the model kinds defined by the architecture description language. When we discussed the denotation of *model kinds* in Section §3.2.2 on page 111, we studied that purpose-specific transformations can be define either to codify tools using Model-Driven Engineering techniques or to codify the integration to external tools providing analysis or update of architecture models. Then, these transformations allow practitioners to provide interoperability of our model-based interpretation with already existing tools.

3.2.4 Communicating Architecture Descriptions

The architecture description is the architect's instrument with two purposes: to captures the decisions on the structure, behavior and quality of the system, in order to guide the construction, maintenance and evolution of the system, and to communicate such decisions to all the stakeholders that hold any kind of concern on the system. These two forces — designing the architecture and communicating the architecture — are not always easy to align. Ideally, the architect should be focused on capturing the complete set of decisions that satisfies all concerns, producing an architecture description that preserves all those decisions, and the resulting structure and behavior specification. Besides, each stakeholder should be presented with the exact information he or she requires, at the appropriate level of detail, and by means of languages and notations that are according to the skills of the stakeholder. Structuring the architecture description in terms of *architecture views* and *architecture models* is the main instrument at hand for the architect to target a specific subset of stakeholders with the corresponding set of decisions that are of interest for them. However, producing an architecture view or architecture model that is suitable to the various needs and skills of the targeted stakeholders is not always possible, as two different stakeholders may require the same information at two different levels of detail, or expressed in terms of two different languages — for instance, while a stakeholder such as an implementer may require the detailed refinement of the system in terms of components, connectors, their interfaces and their interactions, a stakeholder such as the operator of the production environment may require the high level structure in terms of components and connectors only.

We reviewed and discussed in Section §3.1.1 the conceptual model for architecture descriptions. In Section §3.2.2 we defined our formal semantics of these concepts in terms of Model-Driven Engineering (MDE) constructs. One of the main benefits of capturing architecture descriptions in terms of interrelated modeling artifacts is that the architecture design practice can then be understood and formalized in terms of the manipulation of these modeling artifacts. Thus, in Section §4.2 we define our formal specification of the architecture design practice in terms of fine-grained updates to architecture description modeling artifacts. However, architecture descriptions that solely rely on models present an obstacle to stakeholders. Even though they can benefit from the available tool support for visualizing and exploring the architecture description, the set of modeling artifacts conforming the description not only includes information that goes beyond the interest of each stakeholder, but also such information is captured at a level of detail and possibly by means of modeling languages that are not understood by all stakeholders, mainly external ones. Ideally, each stakeholder should be presented with the exact information he or she requires, at the appropriate level of detail, and by means of languages that are according to the skills of the stakeholder. *Architecture views* are the main instrument of the architect for aiming the subset of architecture decisions and structures to specific subsets of stakeholders. As a consequence, either the architect repeats information in more than one architecture view or model to deal with the different necessary levels of detail and languages, or the stakeholders must conform with architecture views and models with a larger coverage than that required and must learn the languages used. These two forces, designing the architecture and communicating the architecture, are not always easy to align and hence, information tend to be repeated (which makes inconsistencies more probable) or stakeholders tend to be informed with more than

they really need. According to P. Clements et al. in [CBB+10], to choose the appropriate set of views, the architect must identify the stakeholders that depend on the documentation and their information needs. For the authors, these needs must be classified in levels of detail, and the selected views must be combined to reduce their number although superseding one stakeholder's needs to that of others. In practice, by augmenting the architecture description with a companion guide, the architect instructs stakeholders which parts of the description may be of their interest, and warns them to pay attention only to some aspects when the level of detail exceeds the stakeholders' needs or skills.

As we reviewed in Section §2.1.2, there are different ways to call the activity of building an architecture description: documentation, description, representation and specification. These are all valid perspectives to conceive the artifact *architecture description*, each emphasizing a different purpose. While the focus representation and specification is on capturing architecture decisions and structures to cover the complete system, the focus of documentation and description is on communicating those decisions to stakeholders. It is the responsibility of the architect to decide which is the suitable perspective for the particular development scenario, and then, to design the architecture in a way that favors such a perspective. The ISO/IEC/IEEE 42010:2011 standard for architecture description is independent on the actual focus or intention of the architect — the selected architecture viewpoints and the way they are used by the architect ends up determining the predominant perspective of the architecture description as an artifact.

We claim that this approach lacks the separation of the two main concerns of architecture descriptions: to capture the architecture design to its necessary extend, and to communicate such a design satisfying the stakeholder information needs. Separating these concerns not only allows architects to focus their attention in one problem at a time, but also to capture, encapsulate, share and reuse the knowledge on how to extract information from the architecture description to create a document that communicates the segment of that description that satisfies the specific information needs of a given type of stakeholder. We define a three-forces approach to architecture description that addresses the separation of concerns providing these benefits.

The Tree-Forces Approach to Architecture Description

Creating an architecture description is guides by two main forces: design and communication. The *design* force is responsible for generating the complete representation of the architecture that satisfy the information needs of <u>all</u> the stakeholders that are involved or have an interest on the architecture. The design force aims for achieving the highest level of accuracy and detail as possible, avoiding all inconsistencies in the architecture description. The *communication* force is responsible for generating the projections of the architecture description that best satisfy the information need of <u>each</u> stakeholder. The communication force creates one single document for each stakeholder, covering the correct fragment of the architecture description that is of interest to that stakeholder, expressing it at the appropriate level of detail, and using only notations that can be understood by that stakeholder.

These two forces present *opposite* interests. The design force is concerned with all stake-

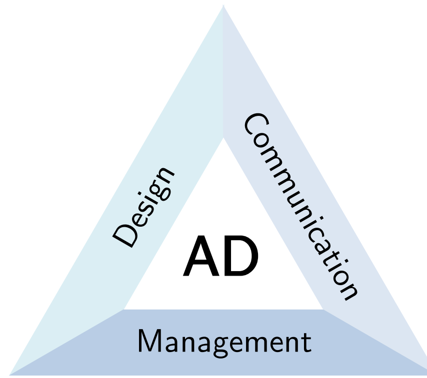


Figure 3.7: *The Three-Forces Approach to Architecture Description.*
The figure illustrates the three forces the impact and guide the construction of the architecture description.

holders at once, making available all the required information, favoring the design activity by using the best suited architecture description languages and by avoiding redundancy in the architecture description. It is not concerned on the ability of stakeholders to understand what is captured, as far as the architect does understand it. The communication force is concerned with one stakeholder at a time, presenting the information in the precise way that best suit that stakeholder. In practice, favoring one of these two forces is in detriment of the other. However, these two forces can also be *complementary*. Instead of favoring one or the other, they can be enacted in a way that one complements the other. The communication force can use the architecture description produced by the design force to generate the corresponding documentation for each stakeholder. In turn, the needs detected by the communication force help the design force to determine to what extent the architecture description must be detailed. However, as we discussed before, to enact both forces at the same time require additional effort as multiple versions of the same architecture description are required, one complete and one for each stakeholder. This is impractical in the absence of the corresponding tool support. We define a third force to bring balance to the enactment of the other two forces. The *management* force is responsible to facilitate the interaction between the other two forces, using the information needs detected by the *communication* force to limit the effort of the *design* force, and guiding the *design* force on the actual architecture views and languages to use to favor the communication of the architecture description. The management force takes into account budget and time restrictions, the skills of the team producing and consuming the architecture description, and the availability of tool support. Figure 3.7 illustrates these three forces schematically.

The Three-Forces Model-Based Approach

The ISO/IEC/IEEE 42010:2011 standard is independent of the actual medium to express architecture descriptions, mentioning document-based, repository-based and model-based as the most notorious ones. Conformance to the standard requires the architecture description to be structured in a particular way, regardless of the medium used to achieve it. However, in practice, document-based and repository-based architecture descriptions currently predominate. Document-based architecture descriptions are constructed using plain documents,

including text and figures, where architecture views and architecture models are mapped to document sections and subsections. Correspondences are captured along with the views and models, or by means of a separate section in the document. Cross references within the document are used to assist stakeholders in relating information from one section (view or model) to another. Repository-based architecture descriptions, like wikis, also rely in text and figures, but architecture views and architecture models are mapped to different document (e.g. wiki pages).

We argue that by means of document- or repository-based architecture descriptions, the separation of concerns between the *design* and *communication* forces is hard to achieve. To obtain different levels of detail or to use different notations require the information to be repeated in more than one place, with the subsequent effort to maintain and evolve them. In practice, the *management* force opt for avoiding duplication and hence providing a single architecture description with the corresponding companion guide to instruct stakeholders on how to read the architecture description. By using a model-based architecture description, however, architects can achieve a higher level of automation than by using any text-oriented approach. To this end, we define the three-forces model-based approach to architecture description based on our interpretation of the architecture description practice in terms of Model-Driven Engineering techniques. The main goal of our approach is to allow the architect to separate concerns between design and communication, and to provide a mechanism to capture the architecture knowledge on how to generate the stakeholder-oriented version of the architecture description from a base architecture description, in a way that it can be shared and reused across architecting scenarios or projects.

In our approach, the *design* force is responsible for creating the architecture description in terms of modeling artifacts. The architect uses our model-based interpretation — defined in Sections §3.2.1 and §3.2.2 — to determine the required modeling artifacts needed, and populate them in order to meet all architecturally significant concerns while satisfying all stakeholder's information needs. The architect decides the structure of the architecture description in terms of architecture views and architecture models considering the complete set of stakeholder information needs instead of particular ones. The *communication* force is responsible for identifying the information needs for each stakeholder, to decide how to obtain such information from the architecture description constructed by the *design* force, and to capture such knowledge in a way that it is shareable and reusable. To this end, the architect defines and develops a *model transformation* that takes as input one or more artifacts from the model repository conforming the architecture description, and that generates as output the external artifact containing the information required by the stakeholder. These external artifacts are composed by structured (e.g. sections) text and diagrams, that must be fully created from the information in the input artifacts. By this means, when the design force updates the architecture description, the model transformation can be re-applied to generate a new version of the external artifact. The upfront extra effort for producing the model transformation is paid-off by obtaining a version of the external artifact at any time, and without additional effort to maintain or evolve the artifact when the architecture description changes. Moreover, this kind of transformations could be included in the architecture frameworks and architecture description languages to provide as a reusable asset particular ways of communicating their defined model kinds to different stakeholders. It is important to remark, however, that we are not suggesting to create a model transformation that explain in

natural language an architecture model. As we defined in Section §3.2.2, complex concepts (like architecture view, architecture model, correspondence) include in their denotation a *documentation model*. These documentation models are the source for most of the textual information provided in the external artifact. Besides, part of the narrative in the produced artifact can also come from any textual property attached to the model elements in the terminal models conforming the architecture description. For instance, an architecture model from a Module View can include for each module a description of its responsibilities. Then, this description can be used in the narrative of the external artifact. Also, terminal models are used to generate diagrams to be included in the external artifact.

We define two main approaches to encode the *communication* force in terms of a model transformation: one-step and two-step approach. The *one-step* approach consists of implementing an model-to-external model transformation that creates the external artifact directly. This transformation not only encodes how to extract the information from the architecture description, but also it encodes how to produce the external artifact with the extracted information. Clearly, two concerns are tangled in these transformations. The *two-step* approach addresses the separation of these concerns. First, a model transformation encodes how to extract the information from the architecture description, generating an intermediate model of the external artifact to be produced. This transformation is a model-to-model transformation. The modeling language for this intermediate model provides the constructs for representing textual documents and graphical figures. The constructs for text documents are an aggregation of section, subsection and paragraph. The constructs for diagrams can be as general as boxes, lines and labels. Technologies such as the Eclipse GMF and Graphiti projects can be used in this latter case. Then, a second transformation takes the intermediate model and produce the corresponding external output. The additional benefit of this approach is that more than one model-to-external transformation can be conceived, targeting external artifacts in different technologies, such as PDF and HTML, among others. Figure 3.8 illustrates both approaches.

The responsibility of the *management* force is to balance the effort between the design and communication forces. This force is continuously analyzing whether the model transformations can be implemented with the selected structure of the architecture description, and identifying opportunities of reuse existing transformations (e.g. from the same or other projects) for generating the external artifacts.

3.3 Contributions & Discussion

In this chapter we addressed the problem of the lack of a homogeneous means to capture and capitalize architecture knowledge on architecture description. Thus, the goal and main contribution of this chapter is the definition of such a homogeneous means, making such knowledge shareable, reusable, tool-friendly and directly applicable during architecture design.

In the case of architecture description, it is already available to the community a contextual and conceptual model of the practice defined by the ISO/IEC/IEEE 42010:2011 standard [ISO11]. In this chapter we provided a thorough revision and discussion of both

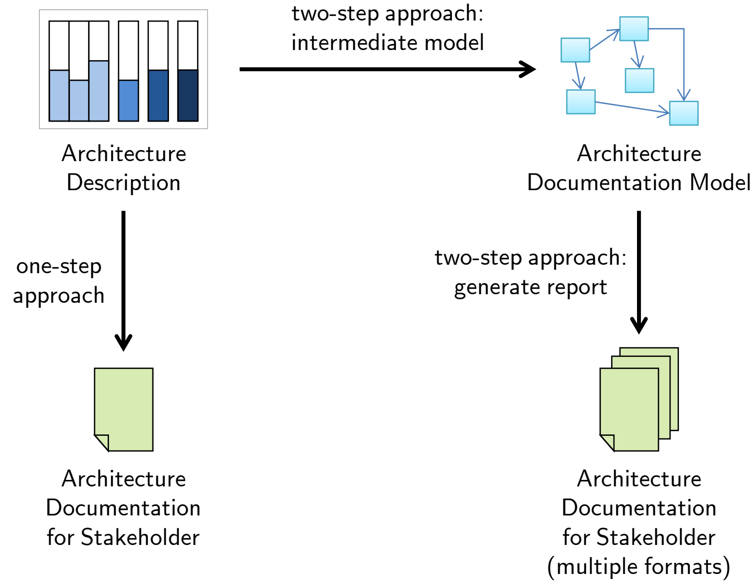


Figure 3.8: *Model-Based Communication Force.*
The figure illustrates the two different approaches to reify the communication force in terms of a model transformation. A *one-step* approach produces an external artifact directly from the architecture description. A *two-step* approach produces an intermediate model that is later transformed to one or more external artifacts in different formats.

models. We analyzed the gap in the definition between the two, identifying that while the contextual model refers to multiple architecture descriptions of multiple architectures of a single system, the conceptual model is unnecessary restricted to a single architecture of a single system. Also, we reported the ill-formed aspects of the conceptual model that we discovered when considering it from the modeling perspective. In particular, the standard's conceptual model mixes real world concepts with representation (model world) concepts. As a consequence, we defined our own adaptation to the conceptual model that captures only those concepts pertaining the description of architectures of systems-of-interest and their interrelation, excluding the real world concepts. Although we kept aligned to the standard's model to ensure compliance of our model-based formalization, we defined it in such a way that it is mainly the formalization of our adapted conceptual model.

The original contribution of this chapter is the model-based interpretation of the conceptual model in terms of modeling artifacts categorized, classified and structurally interrelated in terms of the megamodeling approach to modeling-in-the-large. To this end, we applied the denotational semantics approach to the formalization of the conceptual model in terms of Global Model Management techniques. Formal semantics have been used in the context of software architecture mainly to specify the meaning and behavior of Architecture Description Languages. Formal semantics have been used in the context of Model-Driven Engineering mainly to provide theoretical support for the practice, from meaning of modeling to behavior of model transformations. In our work, we apply formal semantics for a different purpose: to specify how any concept and concept instance in the software architecture practice can be understood, represented or denoted, in terms of model-driven techniques. While an informal approach to capture this mapping might be enough to grasp the principles and some details, by following a formal approach we provide an unambiguous and rigorous specification that

can serve as the basis for tool support. Besides, our formalization can serve as an example of the application of the denotational semantics approach to the formalization of a specific domain in terms of model-driven techniques and concepts.

In Section §2.3 we contributed with a normalized and tool-independent metamodel for megamodels, strongly inspired in the Global Model Management realization of the megamodeling approach, and preserving tool-compatibility. In this chapter, we contributed with a formal syntax to express elements such as model repositories and modeling artifacts, and to express assertions on these elements. This formal syntax can be used in similar efforts where the megamodeling approach is conceived as the semantic domain for denoting any particular application domain. In particular, our syntax allows to express complex predicates such as conjunction and quantification, which is usually not possible using informal approaches such as UML object diagrams.

We do not claim that the defined interpretation is neither the unique possible one nor the best one. In order to make such a claim, it would have been necessary to determine the expected properties and quality of the interpretation, and a later evaluation of the extent that our interpretation achieves such expectations. Our goal was twofold: to make it complete and to make it feasible in the current state of the art of modeling techniques. This is the goal we actually achieved. First, we covered all concepts in the architecture description practice, and we provide the denotation for the concepts themselves and for any instance of these concepts. Second, our semantic domain is explicitly inspired, aligned and supported by the Global Model Management approach, which encompasses the available modeling techniques in the current state of the art. Moreover, our semantics uses customization to allow practitioners to build the modeling artifacts in a way that they are specific to their development scenario, such as the organizational culture, the skills of the team, time and budget restrictions, and the expertise on tool support.

Our model-based interpretation is based in the megamodel concept, as we understood software architecture as modeling-in-the-large. It is not plausible that a single modeling artifact can cope with all the aspects of a single system and hence, we cannot expect a single metamodel to provide a complete and comprehensive modeling language. This is suggested in the FAQ [ISO13] of the standard and is also commonly referenced in architecture frameworks. For instance, DoDAF [DoD09] defines a metamodel for the framework. As we discussed in this chapter, these metamodels are practically domain models as they capture the main concepts and relations in a specific domain. From the modeling perspective, these metamodels are not defining a specific modeling language to describe an architecture, an architecture framework, or an architecture description language, but rather they are characterizing which kind of elements exists in a particular domain and how they are interrelated. Thus, they are actually defining a domain-specific metamodel for a megamodel, which elements end up being modeling artifacts in a model repository. Moreover, multiple artifacts are required to avoid redundancy and to provide knowledge encapsulation and reuse. For example, A. Jossic et al. in [JFL+07] uses software architecture as a case study for modeling complex domains, and in particular, they study model integration using model weaving to create a model-based representation of DoDAF architecture framework. The authors use multiple models for capturing DoDAF views and use weaving models to capture the interrelation of the populating elements. Examples of those modeling artifacts are available in [Jos14]. This

is the exact same approach that we follow for the general case, capturing architecture models as terminal models and correspondences between those models as weaving models.

Our adapted conceptual model can be regarded as a domain-specific metamodel for the megamodels in the architecture description practice. This domain-specific metamodel enables the development of specialized tool support, providing visualization, exploration, navigation and operations in terms of the specific domain. This is the main idea behind domain-specific development environment, a integrated environment providing tool support for a given domain, i.e. a specialized CASE tool. For example, J. Estublier et al. study in [EVLL08] techniques for developing such environments. ArchStudio [DAH+07] is an example of a development environment targeting software architecture. Our model-based interpretation of the conceptual model provides a formal basis for building such tool support. Domain-specific tools can be implemented in terms of modeling tools provided by a modeling environment, following our formal mapping. However, as the Model-Driven Engineering discipline is still going through an active evolution in terms of concepts but mainly tool support, our position is that it is yet too early to build such a tool in a way that it can achieve broad adoption. Hence, we defined our formal interpretation targeting a conceptualization of modeling techniques to protect our work from the evolution of the available technology. Adapting current and future tools to the conceptualization is enough to achieve tool support. Nevertheless, a domain-specific development environment, possibly integrated within a larger development environment, can provide the modeling community with new directions of evolution of current tools in order to improve support for current practical applications of the technology.

In summary, our model-based interpretation uses modeling artifacts as a homogeneous means to capture architecture knowledge in architecture description. Modeling constructs are used to capture both system-independent knowledge such as model kinds, architecture viewpoints, architecture frameworks and architecture description languages, and system-dependent knowledge such as architecture description themselves, architecture views, architecture models and correspondences. Thus, by publishing and consuming the corresponding modeling artifacts, architecture knowledge is being shared and reused within an organization or among the community. Current tool-support for modeling techniques make such knowledge tool-friendly. In Chapter §4 we study how this knowledge can be directly applicable during architecture design.

Extensible Architecture Description Languages. In Section §3.1.2 we reviewed extensible architecture description languages and introduce them as ADLs that provide practitioners the ability to extend them in order to fit their specific architecture description needs. Extensible ADLs provide a basic set of constructs that describe certain common architecture concerns and additionally include support through notation and associated tools for extending the base syntax to support new practitioner-developed constructs [TMD09]. The basic approach to employ an extensible ADL during architecture design present two cases. First, the existing ADL capability is used for those concerns that can be modeled with the ADL baseline. Second, for those concerns that cannot, developers need to choose how to extend the baseline to support the required modeling capability, build such extension and improve current tool support if necessary, and apply it to model the concerns.

The remarkable characteristic of extensible ADLs is that they provide the foundation to capture, document, share and reuse architecture knowledge on architecture description. This characteristic allows the research and practitioner community to collaborate on the development of such reusable assets and make them available to the whole community. As extensible ADLs are formally defined, it also enables the development of supporting tools for visualizing and editing descriptions, checking description syntax and semantics for well-formedness, for performing analysis and processing, among others. Current extensible ADLs are XML-based, taking advantage of already existing technologies such as DTD, XML Schemas, XQuery and XSLT.

Our model-based approach to capture, share and reuse architecture knowledge on architecture descriptions follows the same principles than extensible ADLs: to define a technical space that conforms a homogeneous means for capturing architecture knowledge on architecture description making it shareable, reusable and tool-friendly. However, both approaches present differences. While current extensible ADLs are XML-based and hence uses the XML technical space, our approach is model-based and hence uses the Model-Driven Engineering (MDE) technical space [Béz05a]. A *technical space* (TS) is defined as a working context with a set of associated concepts, body of knowledge, tools, required skills and possibilities [KBA02]. The XML TS is technology specific, as it relies in XML-related technologies. The MDE TS, however, is technology independent which can be reified by different technologies, for instance, the Meta Object Facility (MOF) [OMG11b] and the Eclipse Modeling Framework (EMF) [SBPM09]. Moreover, our model-based interpretation relies on the Global Model Management characterization of the MDE TS, which abstractly defines the kinds of constructs that are available. The equivalent in the XML TS are technology specific tools like XQuery and XSLT.

Besides, in contrast to extensible ADLs, our work cope with all the concepts of the architecture description practice, as identified by the ISO/IEC/IEEE 42010:2011 standard [ISO11]. We provide a denotation for every construct, including extensible ADLs among others. We argue that the MDE TS provides more specificity to the modeling activity with respect to the XML TS, rendering it more appropriate for modeling architecture descriptions. Moreover, we claim that the continuous evolution of the tool support for the MDE TS, being EMF the most advanced, will level the balance on tool support between the both.

Chapter 4

Model-Based Software Architecture Design

Software architecture is the centerpiece of modern software development processes. It plays an essential role in achieving intellectual control over the sophistication and complexity of medium- and large-scale software systems. It provides practitioners a means for dealing with size and intricacy, managing and controlling development goals and risks, and consistently assessing the quality. Architecture design is the creative and intellectual process performed by an architect or architect team to design the architecture of a software system. Its goal is to decide how the stakeholders' expectations are to be addressed by the system of interest. Such decisions determine the ultimate functionality and quality of the system and its fundamental characterization pertaining its constituent elements and their interrelation, and promotes the success of the development effort. Designing the architecture of a system is not a phase of software development. It follows its own sub-process that spans system conception, construction and evolution. It is intertwined to the activities in other development disciplines, looking for achieving the overall potential and capabilities of the software system.

Architecture design is a complex sub-process consisting of three major activities that are repeated iteratively. First, the architect team identifies and captures the significant concerns by analyzing the requirements, constraints and risks that are yet to be addressed, in the light of the software architecture being built so far for the system of interest. Second, the architect team devises one or more potential solutions to address the identified significant concerns, having each of these solutions its own pros and cons. The alternative solutions are compared and weighted, in order to decide the solution that offers the best system-wide long-term benefits despite its drawbacks. Then, the architect team updates the architecture description being built in order make it reflect the decisions made. Third, the architect team, together with the appropriate stakeholders, evaluate the architecture to detect inconsistencies and flaws, and to determine additional or pending requirements to be analyzed. The iteration ends when all significant concerns of all involved stakeholders are satisfied by the architecture that is captured and communicated by means of a complete- and detailed-enough architecture description. The architecture description is not simply the output of the architecture design sub-process. It is the artifact for capturing architecture decisions and their effect in the fundamental characterization of the system, as soon as those decisions are made. The

Chapter Contents

4.1 Concepts of Software Architecture Design **164**
4.1.1 Architecture Patterns, Tactics & Styles 171
Architecture Styles 173
Architecture Tactics 174
Conceptualization 179
4.1.2 Architecture Update Statements 184
Start Statements 187
Identify Statement 190
Use Statements 191
Create Statements 193
Improve Documentation Statements 195
Constrain Statements 197
Apply Statements 198
4.1.3 Architecture Solutions, Decisions & Rationale 202
Conceptualization 203
4.2 Model-Based Architecture Design **207**
4.2.1 Shifting Focus from Description to Design 209
4.2.2 Definition of the semantic functions 212
(i) Syntactic domain 213
(ii) Semantic domain 214
(iii) Semantic function 218
(iv) Semantic equations 220
4.2.3 Semantics of Architecture Design 221
Architecture Design Library 222
Architecture Solutions, Decisions & Rationale 225
Architecture Update Scripts 228
Architecture Design 243
4.2.4 Traceability 245
4.3 Contributions & Discussion **247**

iterative and incremental nature of architecture design allows the architect team to cope with complexity, favoring the resolution of the critical concerns first, possibly at the expense of others. Delving into alternative solutions by trial-and-error, going forward and backwards in the paths of decisions made, is a common scenario during architecture design. The architect team needs to explore the solution space to devise the architecture that provides the best benefits and quality while addressing the most significant concerns.

In the current state of the art, architecture knowledge on architecture design is large and heterogeneous, dealing with different aspects of the architecture design practice. The research and practitioners community have actively developed a plethora of methods and predefined solutions to assist architect teams. From the methodological point of view, there are several software architecture design methods for practitioners to count upon. Techniques coping only with one of the three major activities are also available. Moreover, tool-support has improved over the years, going from isolated tools targeting specific architecture description languages, to both commercial and non-commercial tools participating in integrated development environments. From the predefined solutions point of view, the community has captured, documented, shared and reused pre-cooked designs in terms of architecture patterns. Also, the community has captured and made available pre-cooked implementations in the form of technological frameworks, virtual machines, application programming interfaces, generic and reusable components and products, standardized protocols and languages, among others. These pre-cooked implementations permeate architecture design as they are considered by architect teams when devising alternative solutions. The technological capability of the customer's production environment, the availability of specific platforms and product versions and configurations, and the skills of the development team, installation team and operating team, are determining for shaping the architecture of the system.

Our work focuses on the decision-making activity of the architecture design sub-process. From the methodological point of view, we are interested on how to capture architecture decisions and solutions and how to reflect their effect on architecture descriptions, while facilitating practitioners the exploration of the solution space and promoting the integration with already existing tools. The systematization and formalization of architecture design methods is out of the scope of this work. We suggest it as a further line of research. From the predefined solutions point of view, we are interested on how to capture pre-cooked designs, i.e. architecture patterns, in a way that allows practitioners to apply them on architecture descriptions in a tool friendly manner. In the case of pre-cooked implementations, no special treatment is required. Pre-cooked implementations impact architecture descriptions by providing special architecture viewpoints and model kinds that govern the architecture views and architecture models that are specific to solutions based on a given technology. Pre-cooked implementations impact architecture design by providing technology-specific patterns embodying solutions on that technology.

The research and practitioners communities have reached a consensus on the conceptualization of the architecture description practice. The ISO/IEC/IEEE 42010:2011 standard [ISO11] specifies the requirements for well-constructed architecture descriptions, architecture frameworks and architecture description languages. However, no similar consensus has been reached with respect to the conceptualization of the architecture design practice. The standard is not dependent on nor requires any particular architecture design method or system life cycle process. Standards on system engineering processes and life cycles have a general purpose and are not specific to software architecture design. In any case, as we studied in Chapter §3 for the architecture description practice, reaching a consensus on conceptualization or requirements, although crucial, is insufficient. Architecture knowledge on the architecture design practice is available in the form of books and articles, both printed and published in the Internet, and encapsulated in proprietary or community-developed tools. Moreover, proposals on how to capture different aspects of this body of knowledge are still

emerging in the community. To the best of our knowledge, there is no consensus in the community on how to capture and communicate the architecture knowledge on the architecture design practice. We argue that this lack of an uniform means in hindering the growth of the software architecture discipline and the pace of its adoption in industry. The mechanisms that researchers and practitioners rely on today do not encourage, promote or facilitate collaboration. Our position is that today, community-developed tools are the mechanism with the best chances to become a consensual means in the future. However, they lack a solid foundation and their application in practice is not widespread yet.

We address this problem by applying Model-Driven Engineering techniques as the homogeneous means for explicitly capturing architecture knowledge on the decision-making activity of architecture design. To this end, we first define a conceptual model of the concepts pertaining the activity. Second, we define how these concepts can be captured or interpreted in terms of modeling artifacts, particularly by those that are characterized by the Global Model Management approach explained in Section §2.3. We extend the denotational semantics we defined in Chapter §3 to formally define this interpretation. Then, Model-Driven Engineering constructs conforms the homogeneous medium for capturing, documenting, sharing and reusing architecture knowledge on architecture description and design.

This chapter is structured as follows. Section §4.1 defines a conceptual model of the practice of the decision-making activity of the architecture design sub-process. This conceptual model is intended as an extension of the conceptual model proposed in the ISO/IEC/IEEE 42010:2011 standard [ISO11]. First, we study architecture patterns, tactics and styles, and we define how these concepts are related to each other and to the concepts in the standard. Second, we define architecture solutions and the set of possible fine-grained updates to architecture descriptions that can be made by such solutions. Third, we review how architecture decisions and rationale are defined in the standard and comment on how other authors are dealing with them. We then capture decisions and rationale in our conceptual model for architecture design. Section §4.2 introduces our model-based approach to architecture design as a paradigm shift from current approaches: to capture how to create an architecture description instead of capturing the architecture description itself. Then, we define the model-based interpretation of the constructs in our extended conceptual model, in terms of the constructs defined by the Global Model Management approach. To this end, we extend the denotational semantics defined in Section §3.2.1 to cope with the extended conceptual model. Section §4.3 concludes this chapter with contributions and discussion.

4.1 Concepts of Software Architecture Design

Creative and intellectual disciplines are founded on four cornerstones that can be synthesized by establishing who produces what, how and when. A discipline determines the main roles involved and the significant stakeholders participating (who), the concepts and ideas to be devised and the artifacts to be developed (what), the activities and methods to be performed and followed (how), and the organization of these activities in time (when). These four practical aspects are characterized and shaped by the needs or causes (why) for the discipline, and by its goals and purpose (for what).

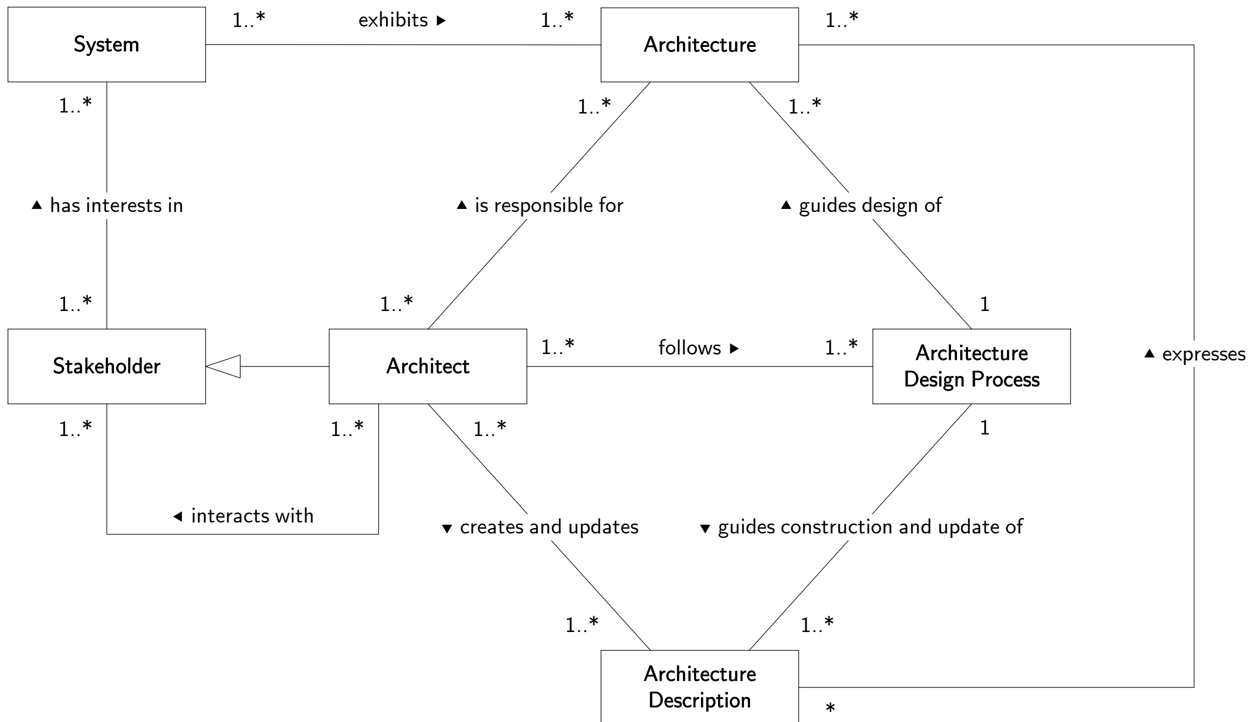


Figure 4.1: *Context of the software architecture design practice.*

The figure illustrates the main concepts pertaining to the context of the practice of software architecture design.

As we reviewed in Section §2.1, the *Software Architecture discipline* empowers organizations by allowing them to achieve intellectual control, conceptual integrity, effective communication and appropriate quality, on the systems and platforms that their businesses rely on. Also, it empowers development teams by allowing them to cope and overcome the sophistication and complexity of the systems that they develop. The discipline positions the role of software *architect* as the leader, mentor and negotiator, with knowledge, skills and expertise on business domains, development processes, design techniques and available technology. The architect interacts with the *stakeholders* to discover, elicit and prioritize the significant concerns that they have on the *system*. The architect devises and weights alternative solutions and decides the solution that offers the best system-wide long-term benefits while meeting those significant concerns. It is the addressed concerns of the stakeholders, the solution, the potential alternatives, their comparison and the rationale behind the decisions, that conforms the *architecture* of the system. The architecture is an intangible concept or idea devised by the architect, which must be captured and described in order to be preserved, communicated, analyzed and evaluated. An *architecture description* is the tangible work product that is created by the architect to express the architecture. The architect follows an *architecture design process* that defines the activities that must be performed to develop the architecture of a system. An architecture design process spans system conception, construction and evolution, and it is intertwined with every discipline in software development. The architecture design process uses an iterative and incremental life cycle to guide the design of the architecture and, simultaneously, the construction of the architecture description.

Then, in terms of the four cornerstones we identified above, we have that the *Software*

Architecture discipline establishes that the *architect* (who) in interaction with *stakeholders* (who), devises the *architecture* (what) and expresses it in an *architecture description* (what), following an *architecture design process* (how), that is iterative, incremental, and is performed during the system lifespan (when). Figure 4.1 illustrates these key concepts pertaining the context of the Software Architecture discipline, particularly from the perspective of the *architecture design practice*. The context of the *architecture description practice* was discussed in Section §3.1 and illustrated in Figure 3.1. Both contexts intersect as they refer to two intertwined practices of the Software Architecture discipline.

An *architecture design process*, also known as *architecture definition process* and mainly *Software Architecture Design Method* (SADM), is the complex sub-process of software development that defines the broad, creative, dynamic and intellectual activities that must be performed by the architect to design the architecture of a software system. As we reviewed in Section §2.1.2, there are several processes available and there is no consensus on a unified process that fits all architecting scenarios. However, there are three goals that are shared by these processes: to understand the problem, to solve it, and to evaluate the solution. Thus, in general, an *architecture design process* consists of the iterative repetition of three major activities: Requirement Analysis, Decision Making, and Architectural Evaluation. We capture this abstract process in Figure 4.2. The *Requirement Analysis* activity is responsible for the discovery, elicitation and prioritization of the critical functionality, the expectations on quality attributes, the constraints on the system, on its environment and on the development project, the agreements made with stakeholders, and the risks of failure. A thorough and correct requirement analysis guarantees that the architect is making system-wide long-term decisions that actually address the critical aspects of the system of interest. Provided the iterative nature of the process, the requirement analysis is performed in the light of the architecture being built so far. Hence, those decisions previously made also affect the significant concerns identified, and might require their reformulation or renegotiation. The *Decision Making* activity is responsible for devising alternative solutions to address these critical aspects, comparing and weighting these alternatives, and deciding the solution that promises the best benefits in spite of its drawbacks. This activity is about analyzing the acceptable trade-offs among the potential virtues of the different architectures for the system. The architect explores the solution space to decide or agree the most suitable architecture in the light of the information the architect has available at the moment. The *Architectural Evaluation* activity is responsible for assessing that the architecture being built so far guarantees or meets the significant stakeholders' goals and expectations, and that the decisions made are the best compromise given the potential alternatives found. The *Decision Making* activity is the core of the process as it guides the architect on designing the architecture and capturing it by means of an architecture description. However, for this activity to be enacted successfully, the other two activities should be carefully integrated [HKN+05, RSM+04].

As we reviewed in Section §2.1.2, in the context of the *Decision Making* activity, there are different approaches to address the significant concerns and to achieve the stakeholders' expectations on quality attributes [BBKS05]. Earlier approaches use predefined decision trees or predictive models to guide the architect on the resolution of quality requirements. Although these approaches render successful results in certain architecture scenarios, they lack complete coverage of quality attributes and concerns, and they have poor scalability to accommodate large designs for a large number of (potentially conflicting) requirements.

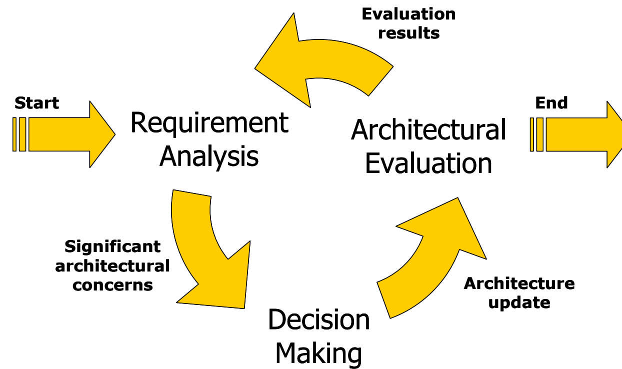


Figure 4.2: *Abstract model for a Software Architecture Design Method.*
The figure is an adaptation of the proposals in [FCK07, HKN+05]. The iterative method starts by performing Requirement Analysis in which significant architectural concerns are identified. Then, in the Decision Making activity a solution for these concerns is found and the architecture description is updated. Afterwards, the new architecture is evaluated. The process ends when all architectural significant concerns are satisfied by the architecture. When they are not, a new iteration is performed in order to cope with uncovered concerns. This figure was already presented as Figure 2.1 in Chapter §2.

Recent approaches use an iterative and incremental life cycle for architecture design. These approaches, known as *intuitive approaches* [BCK03, Bos00], discover, organize and process requirements effectively. The main disadvantage of these approaches is that they strongly rely on the architect knowledge, expertise and skills, to find a satisfying architecture. However, the techniques and tools used by previous approaches might be considered by the architect in order to compliment the lack of expertise.

Creating the architecture description for the architecture being designed is neither an optional nor an a posteriori task. The architecture description is not simply the outcome of architecture design, it is an intrinsic aspect of architecture design. Quoting P. Clements et al. in [CBB+10, Section P.2.5], “don’t consider architecture documentation [description] as a task separate from design; rather, make it an essential part of the AD [architecture design] process, serving as a ready vessel for holding the output of architecture decisions as soon as those decisions are made.” This is specially important in iterative and incremental architecture design processes, such as the intuitive approaches. The architecture description is incrementally constructed, iteration by iteration. After each iteration, the architecture description is updated to reflect the decisions made in that iteration. The set of architecture views and architecture models is possibly adjusted, and the architecture models and correspondences are populated by architecture description elements that embody and reify the intention of the architect. The architecture description being constructed so far, is considered during *Requirement Analysis* to analyze the pending significant concerns to be addressed and to reformulate or renegotiate the conflicting ones. It is used during *Decision Making* to determine how to improve it in order to address the next pressing concerns. It is used during *Architectural Evaluation* to analyze how the concerns are being addressed, why a given solution was chosen over its alternatives, and making explicit the pros and cons of the current architecture. We illustrate this succession of architecture descriptions in Figure 4.3.

Activities in an architecture design process tend to be more fluid that the activities of

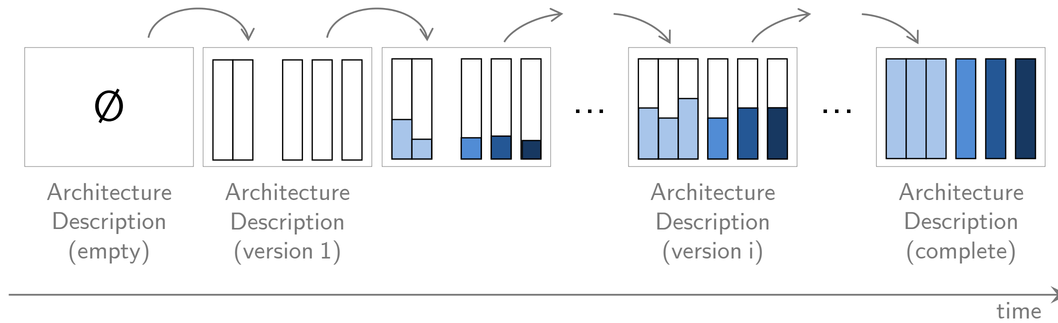


Figure 4.3: *Iterative & Incremental Architecture Design.*

The figure illustrates the successive versions of the *architecture description* of the architecture of the system being developed. The figure uses a large rectangle to represent an architecture description, a small vertical rectangle to represent an architecture model, and attached vertical rectangles to represent an architecture view. The colored section of the architecture models represents the presence of architecture description elements populating the architecture models. In the example in the figure, the architect starts with an empty architecture description, decides the main structural organization of the architecture description by adhering to an architecture framework, and iteratively and incrementally populates the architecture views and architecture models to capture and reflect the decisions made. The figure also illustrates that an additional architecture model was appended to one of the architecture views at some iteration between 2 and i.

the other software development disciplines [RW05]. Architecture design starts early in the project life cycle, often when the scope, requirements and constraints are still unclear and unspecified, and when the current view that stakeholders have of the system may differ substantially from the system that is eventually built. The activities of software development processes are anchored in the architecture [TMD09] and are performed in a more stable context. Requirement specification, system design, implementation and testing, are usually performed when the problem to be addressed is better understood and when the solution to be built is well-envisioned and formulated. This is the responsibility of the software architecture. Architecture design progresses by trial-and-error, going forward and backward while exploring the solution space. Based on the requirements and constraints discovered and prioritized so far, the architect identifies the significant concerns and uses them to drive the design of the architecture. The architect explores alternative solutions and decides the one providing best system-wide long-term benefits. In turn, the architect also explores the problem space in interaction with the corresponding stakeholders. The stakeholders vision of and expectations on the system is refined during architecture design. On the one hand, the architect's knowledge and expertise enriches the potential of the system being developed. On the other hand, the benefits and drawbacks of the architecture being designed so far, both guide and restrict the requirements that are yet to be addressed by the architecture. Figure 4.4 illustrates the exploration of the solution space that usually takes place during architecture design. It is important to remark that, as the figure shows, not every alternative path is explored in depth until a complete architecture is achieved. However, these paths are explored sufficiently enough for stakeholders to be convinced that the decided path provides the best overall benefits. Time is not considered in the figure. The architect may decide to jump from one architecture to another when exploring the solution space.

Given the iterative and incremental nature of architecture design, conflicts may arise at one iteration with respect to what was discovered or decided at a previous iteration. While

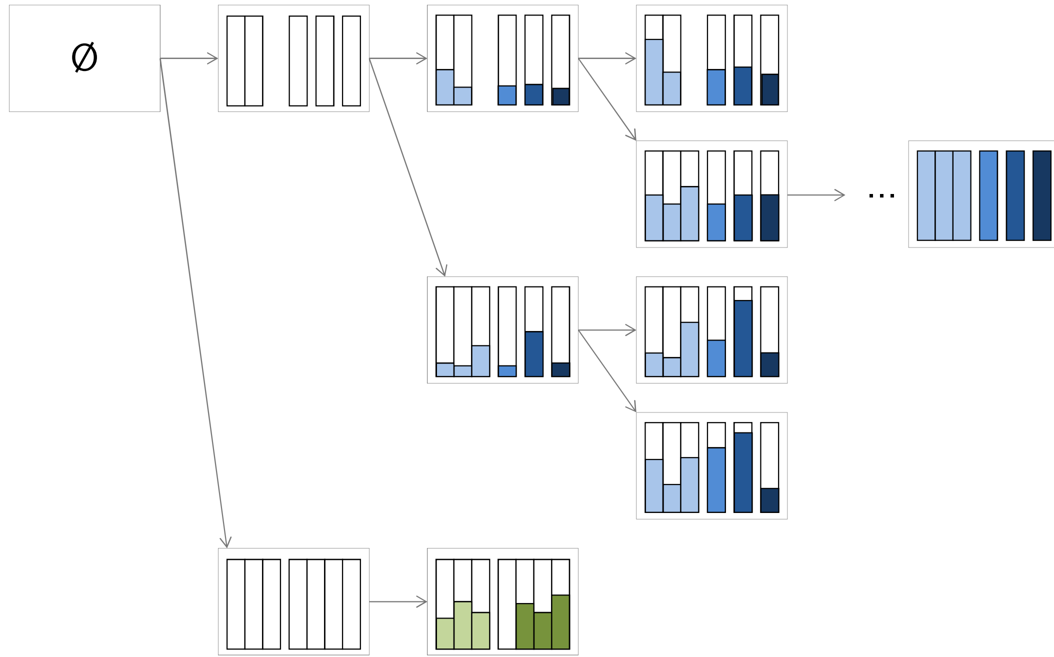


Figure 4.4: *Exploration of the Solution Space at Architecture Design.*
The figure illustrates the exploration of the solution space when designing the architecture of the architecture of the system being developed. The figure uses a large rectangle to represent an architecture description, a small vertical rectangle to represent an architecture model, and attached vertical rectangles to represent an architecture view. The colored section of the architecture models represents the presence of architecture description elements populating the architecture models. In the example in the figure, the architect starts with an empty architecture description and explore two different structural organizations by adhering to different architecture frameworks. Different alternatives are explored for one of these organizations. Each path is incrementally developed until stakeholders are convinced that the path is not the best solution. One of the paths is completely developed being it the complete architecture description of the system. The exploration of the solution space is depicted as a tree but it might be a directed graph as some particular decisions might be commutative. Time is not represented in the figure. The architect traverses the graph as needed.

the architect's expertise may reduce the emergence of these conflicts, it cannot actually avoid them. Quoting P. Kruchten, "the life of a software architect is a long and rapid succession of suboptimal design decisions taken partly in the dark" [Kru01]. Conflicts may arise during Requirement Analysis when newly discovered concerns are considered more important than others that were previously discovered and already addressed by the architecture. Conflicts may arise during Decision Making when the concerns to be addressed can only be poorly provided by improving the current architecture or when a previously found alternative would provide a better balance to achieve previous and current concerns. As decisions are constrained by the decisions previously made, the system-wide long-term benefits provided by the architecture might dissipate when new significant information or requirements emerge. Conflicts may arise during Architectural Evaluation when the reviewing team determine that an alternative solution would be better than the solution decided, for instance in the light of additional information that was not available to or detected by the architect. Quoting N. Rozanski et al. in [RW05], "It may be necessary to change direction, possibly even significantly, partway through the exercise as a result of information that you [the architect]

have uncovered through your [the architect's] work." When the architecture is still unstable, i.e. it is subject to significant changes with low impact on the current development progress, conflicts can be resolved by revisiting and changing previous decisions, taking a different path over alternative solutions. When the architecture is already considered stable, i.e. the most significant decisions were already made and the fundamental structure and behavior of the system is already determined, conflicts are not easy to resolve. The work and work products of the others disciplines that relied on what was previously considered a stable architecture, are certainly advanced enough and would be severely impacted by significant changes in the architecture. In this scenario, the renegotiation and reformulation of the stakeholders' vision and expectations is required to avoid the conflicting situation. In the case of disagreement, the extreme measures of proceeding with the significant changes and re-work what was already done in other disciplines, or even of canceling the project, might be necessary.

The exploratory nature of architecture design, together with the latent possibility of facing conflicts due to undiscovered information and inaccurate prioritization of requirements, have compelled the software architecture community to refocus the architecture design and architecture description practices. In the last decade, the focus has been noticeably shifting from considering the software architecture as the set of architecture elements that characterizes the system, to considering it as the set of critical decisions that leads to these elements and the rationale supporting these decisions. As we discussed in Section §2.1.1, the definition of *software architecture* itself has evolved in order to reflect the importance of architecture decisions and rationale. Moreover, the IEEE 1471:2000 standard [IEE00] on *architecture description*, which was later adopted as the ISO/IEC 42010:2007 standard [ISO07], lacked a first-class representation for architecture decisions and rationale. It was the successor standard ISO/IEC/IEEE 42010:2011 [ISO11] that finally included these concepts in the conceptual model of architecture description. Considering and capturing the architecture decisions and rationale as part of the architecture description provides the stakeholders the underlying reasoning that justifies the presence or absence of architecture elements in the characterization of the system. In addition, traceability from concerns to decisions, and from decisions to the involved architecture elements, enable stakeholders to understand which are the elements involved in the resolution of concerns, why those elements were chosen, and which were the alternatives considered by the architect. Traceability allows the architect and stakeholders to determine the architecture decisions and elements impacted by a change in the significant concerns, and vice versa.

Then, in the current state of the art, the *architect* captures the characterization of the system in terms of architecture elements populating the architecture views and architecture models, and the reasoning that leads to selecting those elements. Thus, an *architecture description* answers <u>what</u> elements the architecture embodies and <u>why</u> they were used. The architect uses the activities defined by a particular *architecture design process* to guide the creation and modification of the architecture description in order to address the significant stakeholders' concerns. Architecture design processes provide the coarse-grained steps to be followed when performing each of the design activities that we identify for our abstract process, as illustrated in Figure 4.2.

The systematic conceptualization and formalization of the activities and their coarse-grained steps is out of the scope of our work. We are focused on the Decision Making

activity of architecture design, and particularly, on the fine-grained actions that are performed by the architect to create and update the architecture description both to capture the solutions and decisions, and to reflect their effect on the organization of architecture elements. These fine-grained actions are usually considered as create-update-delete operations on architecture description elements and hence embedded in the tool support. To the best of our knowledge, the research and practitioner community is still lacking a conceptualization of decisions and solutions conceived in terms of these actions, that is seamlessly integrated to the conceptualization of the architecture description practice defined in [ISO11]. In this section, we define such a conceptualization. The interpretation of the defined concepts in terms of Model-Driven Engineering constructs is developed later in Section §4.2.

4.1.1 Architecture Patterns, Tactics & Styles

Historically, the practice of the Software Engineering discipline has had a poor record of learning from experience [RW05]. Software architects and designers often ignore existing and proven design solutions and instead, they tend to create and develop their own solutions to problems that present familiar challenges. This scenario used to be caused by a lack of easy accessible and well documented standard solutions for common architecture and design problems. During the 1990s, design and architecture patterns emerged as a mechanism for capturing, documenting, sharing and reusing proven solutions to recurring problems. Since then, and inspired on C. Alexander's work on patterns for building architecture [AIS+77], a still-growing number of patterns has been identified, documented, catalogued and published [BCK98, BMR+96, CGB+02, Fow02, SC97, SG96, SSRB00]. Currently, the significance of patterns is well-established in the community, and they are understood as essential to the architecture design practice. However, describing, finding and applying patterns still remains largely ad-hoc and unsystematic [AZ05]. On the one hand, the academic training of architects and designers is rarely extensive in patterns, and skills are often gained by experience. Often, the semantics of patterns, their combination and interaction, are not fully comprehended by practitioners. On the other hand, patterns vary in the level of abstraction and granularity, and there is still no consensual mechanism for documenting, classifying and cataloguing them. Moreover, poor or unavailable tool support in development environments is also hindering the extensive application of patterns.

The term *pattern* is widely used in the Software Engineering discipline, most noticeably in Analysis, Design and Implementation. In the Software Architecture discipline, the term *architecture pattern* was first used in [BMR+96] as the result of making the connection between the concepts of *architecture style* and *design pattern* [CBB+10], both emerging in the early 1990s. In the years that followed until today, a plethora of architecture patterns have been published, making a distinction between *architecture pattern* from those patterns used in the other Software Engineering disciplines. We argue that actually any pattern coming from any discipline can be considered as an architecture pattern. Software architecture anchors the activities of the other disciplines and architecture design is about making the critical long-term system-wide decisions that guarantee the expectations of external stakeholders and that guide and rule the work of the development team. Then, such critical decisions can have an impact on any activity of any other discipline and hence, patterns from those

disciplines might be applied in order to use a common vocabulary and well-known solutions. This claim is supported by other authors. Originally, F. Buschmann et al. in [BMR+96] distinguished between *architecture styles* for organizing the system as a whole, *design patterns* for refining elements and their relationships, and *language idioms* guiding the implementation of particular aspects of the system in terms of a given language. As argued by N. Rozanski et al. in [RW05], all these three types of patterns are useful to the architect, either to determine the structure and behavior of the system, or to communicate design and implementation advice and constraints to the development team. Also, according to P. Avgeriou et al. in [AZ05], design patterns can potentially be used as architecture patterns, if one applies them at the level and scope of a system's architecture.

Consequently, there is no generally satisfying, short, crisp definition for the term *architecture pattern*. Bluntly, a *pattern* is a proven solution to a recurring problem faced in a given context. Then, intuitively, an *architecture pattern* is a pattern in which the problem and solution are architecturally significant, i.e. a proven solution whose quality can be predicted, for a recurring problem faced during architecture design.

Architecture patterns are common architectural problem-solution pairs that are well understood and documented [BMR+96, HAZ07]. From the design point of view, each architecture pattern describes the structure and behavior of a software system or an element composing a software system, that aims to satisfy several functional and quality requirements. Architecture patterns are chosen in <u>response</u> to design decisions. Thus, architecture patterns provide the major structure in which multiple design decisions are <u>realized</u> [HA10]. R. Taylor et al. in [TMD09] define an architecture pattern as "a set of architecture design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears". We refine this definition by considering an architecture pattern not as the set of design decisions, but the <u>realization</u> of a set of design decisions, as stated by [HA10]. For us, an architecture pattern captures a frequently-used parameterized mechanism to reflect in an architecture description the effect of making a set of design decisions.

Predefined design solutions in the form of architecture patterns are captured in *architecture pattern libraries*. Each library consists of an aggregation of architecture patterns, either with a general design purpose, or specific to a particular problem domain or system type. There is no homogeneous mechanism for libraries to capture and document the patterns. As noted by M. Fowler in [Fow02], "every author has to choose his pattern form,"and, in addition, the languages and notations used. Although a consistent mechanism would have been preferable to improve comprehension and reuse of design knowledge, and to facilitate tool support, there is no consensus on any particular technique. In general, a pattern is documented by five pieces of information [RW05]:

- *Name.* A unique and meaningful name to allow practitioners to identify and discuss the pattern, and to refer to it by its name when exploring alternative design solutions.
- *Context.* The context sets the stage of the pattern and describes in which situations it may be applied.
- *Problem.* A clear statement of the problem that the pattern solves and any conditions that need to be met in order for the pattern to be effectively applied.

- *Solution.* A description of the solution to the problem, consisting of the type of elements, their relationships and their behavior, and the explanation of how elements of these types work together to solve the problem.
- *Consequences.* The pros, cons and tradeoffs that result from the application of the pattern, in order to allow the architect to decide whether the pattern is a suitable solution to the problem.

Additional information can be documented for architecture patterns, such as: administrative information like title, author and version; examples of the result of applying the pattern; known uses of the pattern in existing systems or types of systems; advice, warning and details on how to implement the pattern; dependencies or influences from and to other architecture patterns or design mechanisms; and reference or supporting material, and bibliographic references.

Architecture Styles

The concept of *architecture style* traces back to the early research on software architecture. In 1990, M. Shaw published her findings on recurring architecture concepts that occur in many systems, calling these concepts "elements of a design language for software architecture" [Sha90] or "design idioms" [Sha91]. In 1992, D. Perry and A. Wolf published in [PW92] similar findings on recurring forms of design in software architectures. Inspired on the building architecture discipline, the authors saw that the term *architecture style* would be a useful and appropriate term to describe those recurring forms of design [CBB+10]. Thus, *architecture styles* emerged as the result of having observed that certain design forms or approaches, manifested as types of elements and relationships, were being repeatedly used in practice. Then, architecture styles were focused on capturing and communicating those recurring solutions. In the years that followed, the community actively worked on the discovery, documentation and categorization of those solutions. As we mentioned before, in 1996, F. Buschmann et al. coined in [BMR+96] the concept of *architecture pattern* by interrelating and fusing together the concepts of *architecture style* and *design pattern*.

The distinction between the concepts of *architecture style* and *architecture pattern* tend to obscure for practitioners as the difference of meaning is usually regarded as subtle, if any. Some works, like [MAS+03], consider these terms as synonyms and hence use them interchangeably. Other works, like [QZX08], state that there is still a debate in the community on whether these concepts refer to the same thing or not. In other cases, as we mentioned before for [RW05], *architecture styles* are considered to be one kind of *architecture pattern*. There are two schools of thought in the literature with respect to recurring design in the software architecture discipline [AZ05], one uses the term *architecture pattern* (e.g. [BMR+96, SSRB00]) and another uses the term *architecture style* (e.g. [BCK03, CBB+10, SC97, SG96]). Both concepts refer to recurring solutions that solve common problems or scenarios in software architecture practice. They provide a common vocabulary to facilitate communication, a means to reason on the resolution of quality attributes, and both assist on documenting the design decisions made by the architect. The key difference rely on their underlying philosophy [AZ05].

An *architecture pattern* is considered as a problem-solution pair that occur in a given context and is affected by it. The context and problem are a central part of a pattern, and determine the conditions to be met for the pattern to be applied. The solution is captured in terms of elements of different types, interacting with each other. Moreover, the pattern not only captures how the solution solves the problem, but also why it is solved. In other words, it also captures the rationale behind the particular solution. In the case of an *architecture style*, the problem does not receive much attention nor does the rationale behind choosing a specific solution. An architecture style is focused on the types of architecture elements, the type of their relationships, their behavior or interactions, and the set of constraints that rule their structure and organization. In other words, the style is focused on the form of the solution and its characteristics, with more lightweight guidance on when a particular style may or may not be useful.

Both *architecture styles* and *architecture patterns* are sets of prepackaged design decisions involving the choice of element types, relation types, properties, and constraints on the topology and interaction among the elements via the relations [CBB+10]. However, patterns are often more specific than styles as patterns show instances of the element types interacting with each other. The *architecture description practice* is concentrated on capturing the variety of solution approaches, i.e. forms of design or *architecture styles*, as the goal of architecture description is to enable the representation of the system being built. An architecture description does not capture architecture patterns, it captures the application of patterns. The *architecture design practice* is concentrated on architecture patterns as they provide proven solution to recurring problems.

Architecture Tactics

Architecture patterns provide a way to manage the unconstrained nature of the architecture design process and to reduce the enormous size and complexity of the solution space. Thus, the core of the Decision Making activity is about understanding, choosing, tailoring, combining and applying patterns. Each architecture pattern defines a solution that resolves and forces multiple quality attributes at the same time, provides the rationale of the solution, and identifies benefits and liabilities (being many of them quality attributes) of applying the pattern. However, those characteristics defined by a pattern are not applicable in the general case. They are only valid in the particular context or situation of the problem addressed by the pattern, and they depend on how the pattern is applied, used or instantiated. Then, *architecture patterns* are not a succinct mechanism to capture how individual quality attributes can be addressed. As a consequence, in addition to architecture patterns, architects use *architecture tactics*, a complementary technique that focuses on the known alternatives that are available to address single quality attributes.

Architecture tactics were first introduced in 2002 by F. Bachmann et al. in [BBK02] and published later in the context of SEI's vision on the practice of Software Architecture [BCK03]. Quoting the authors, "just as design patterns and architecture patterns are carriers of design knowledge, we [the authors] envision a collection of architectural tactics as the carriers of quality-attribute-based design knowledge." An *architecture tactic* is a design option that helps to achieve or improve an individual quality attribute, and that influences

the architect's control over that specific quality attribute. Quality attributes are not simply met or not met by an architecture or system, but rather, to satisfy a quality attribute is a matter of degrees. Typically, a software system has multiple significant quality attributes that need to be satisfied to the degree demanded by the quality-attribute requirements. As it is unlikely that a single architecture pattern addresses the exact problem imposed by the system-of-interest, and also benefits exactly those quality attributes that are significant, the simultaneous application of several *architecture patterns* and *architecture tactics* is required.

The interaction or interference between architecture patterns themselves is not clear, well-documented, well-known, or if any, easy to grasp. Architecture patterns are complex artifacts that are usually informally or underspecified, making understanding, tailoring and combining them a hard task [SK09]. Moreover, there is a close relationship between architecture patterns and architecture tactics. Both define structures and behaviors that shape the architecture of the system-of-interest, although the impact of tactics has a smaller scale. They have an influence on each other as the result of their application needs to co-exist in the architecture description. However, this interaction has been scarcely studied by the community. Current literature on architecture patterns does not even mention architecture tactics. This makes it difficult for architects to make informed decisions as they inevitably work with both architecture patterns and architecture tactics during architecture design.

From the architecture decision making perspective, the relationship and interaction between *architecture tactics* and *architecture patterns* is commented by L. Bass et al. in [BCK03] when the authors introduced the concept of *architecture tactic*. An in-depth analysis of this interaction is studied by N. Harrison et al. in [HA10]. Inspired by these works, we study the interaction between architecture patterns and architecture tactics in what follows. We identify three scenarios of interaction:

(A) Architecture tactics are realized within architecture patterns, as they become part of the patterns alongside other structures. Architecture tactics are considered as foundational *building blocks* from which architecture patterns are created [BCK03]. Researches and practitioners identifying, capturing and documenting architecture patterns may use architecture tactics to devise the generic solution provided by the pattern, and to determine the consequences (benefits and liabilities) of the pattern in terms of quality attributes. Thus, within the structure and behavior defined by an architecture pattern, there are prepackaged applications of architecture tactics. For instance, F. Bachmann et al. in [BBN07] studied how modifiability tactics are prepackaged (i.e. were applied) in the architecture patterns from [BMR+96] that affect modifiability.

(B) Architecture tactics help architects to understand the tradeoffs of selecting one architecture pattern over another. To select (decide) the application of architecture tactics helps determine which architecture patterns can be applied. Provided (A), architecture patterns packages the application of architecture tactics. The architect can use this knowledge to select the potential architecture patterns to be applied, given the set of architecture tactics the architect has decided to apply. This is a bottom-up approach to design where fine-grained decisions (the selection of tactics to be applied) leads to coarse-grained decisions (the selection of patterns).

(C) To select (decide) the application of architecture patterns leads to further selection of architecture tactics. By applying an architecture pattern, the architects resolve a

concrete version of the pattern's problem faced in the architecture being build, by means of a concrete reification of the generic elements participating in the pattern solution. As a consequence, the benefits and liabilities described by the pattern characterize the resulting architecture. The architect can then apply particular aspects to fine-tune the resulting architecture in order to improve the level of satisfaction of quality attributes. This is a top-down approach to design where coarse-grained decisions lead to subsequent fine-grained decisions. As noted by N. Harrison et al. in [HA10], "[architecture patterns] provide the starting point for incorporation of tactics." We distinguish three different cases for this scenario, varying in the precedence in time of their application:

(C1) The application of an architecture tactic is an after-the-fact action. Once the architecture description reflects the application of a particular architecture pattern, the architect decides to apply an architecture tactic to improve how a particular quality attribute is satisfied. This case is common in brownfield architecture design where an architecture has been previously built and the architect need to refine the architecture in order to improve the level of satisfaction of a particular quality attribute. It can also take place in greenfield architecture design when the architect proceeds stepwise [TMD09].

(C2) The application of the architecture pattern and the one or more application tactics required to refine the solution provided by the pattern, are performed at the same time. The architecture description is updated so as to reflect the application of all the patterns and the tactics. This case is common in practice. Once the architect decides to apply a pattern and decides how to refine the solution by means of certain tactics, the architecture description is updated once to reflect all required changes. In document-centric and repository-based architecture descriptions, the effort required to update textual descriptions and diagrams is not minor, so architects tend not to be too gradual in changes.

(C3) The interaction between the architecture pattern and the architecture tactics is a before-the-fact consideration. Architecture pattern designers or architecture tactic designers can capture for each participant type of element in the solution of an architecture pattern, which are the potential architecture tactics that might be applied once the pattern is applied. It differs from scenario (A) as in (C3) architecture tactics are not actually applied to define the solution of the pattern. It differs from scenario (B) as in the latter the tactics, helping the decision of the pattern, are those already applied in the solution of the pattern. In (C3), the solution of the pattern is further characterized to assist architects to devise alternatives when applying an architecture pattern. Anyway, this information on the potentially applicable tactics might also be useful in (B).

These scenarios do not take place all at the same time. On the one hand, scenarios (A) and (C3) are considered when identifying, capturing and documenting architecture patterns. When researchers and practitioners develop an *architecture pattern library*, they use architecture tactics to devise, develop and document the solution and consequences of each architecture pattern they are defining, as in (A). Also, architecture tactics are used to further characterize the participants of the pattern's solution with respect to the potential tactics that can be applied on those participants, as in (C3). These scenarios are not actually part of the architecture Decision Making activity, but rather, they are activities performed by architecture pattern library builders to improve and refine the pattern's solution and docu-

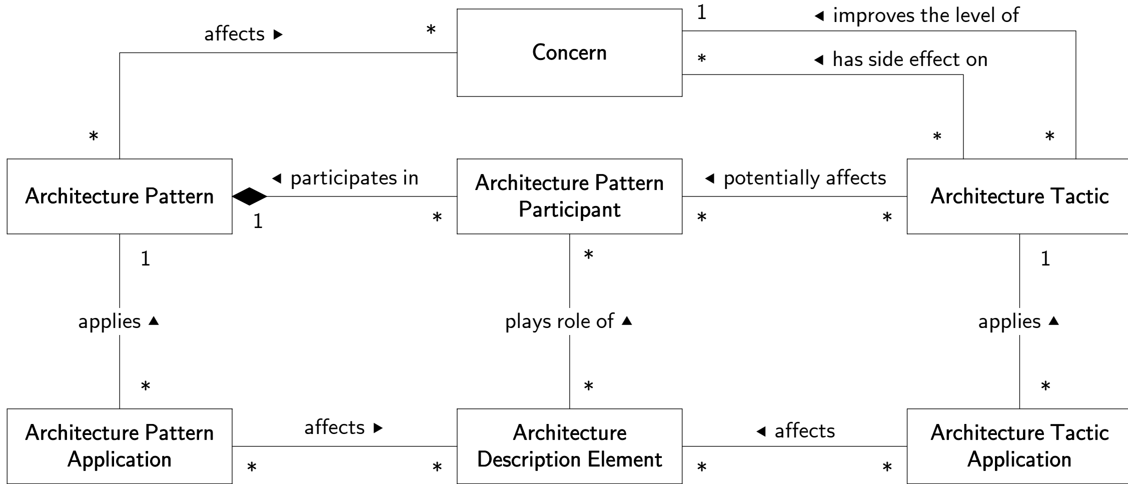


Figure 4.5: *Interaction of architecture patterns and architecture tactics.*
The figure illustrates the interaction between the concepts *architecture pattern* and *architecture tactic*. This figure is inspired on the model for patterns, quality attributes and tactics defined in [HA10]. In our figure, we explicitly separate between *architecture pattern participants* and *architecture description elements* that result from applying the corresponding *architecture pattern*. By this means, we can distinguish the fact that an *architecture tactic* affects an *architecture pattern participant* in the sense that the tactic can be potentially applied on the elements playing the role of that participant. Then, we explicitly illustrate that an *architecture tactic* needs to be applied in order to further characterize and/or modify the *architecture description elements* participating of the *architecture description* being designed.

mentation. On the other hand, scenarios (B), (C1) and (C2) take place during the Decision Making activity of architecture design. These scenarios refer to the interaction of patterns and tactics with respect to devising and capturing the alternative architecture solutions to the requirements being addressed by the architect. While (B) assists the architect in devising (i.e. mentally conceiving) the alternatives, their benefits and liabilities, (C1) and (C2) provide two alternatives to the <u>combined</u> application of patterns and tactics.

The impact of applying patterns and tactics combinedly, and consequently the effort required to do so, varies depending on which patterns and tactics are being combined. In some cases, the application of an architecture tactic can be easily combined into the application of an architecture pattern when the tactic uses the same structures and compatible behavior as the pattern. In other cases, the application of an architecture tactic may require significant changes to the structure and behavior of the pattern, or may even require an entirely new structure and behavior [HA10]. We claim that in these latter cases, using a stepwise approach by separating the decision on the pattern first, and the tactic(s) later, as in (C1), facilitates reasoning and improves traceability from decisions to architecture description elements. Following (C2) the resulting structure may strongly differ from that proposed by the pattern and hence the fact that it was applied may remain obscure. N. Harrison et al. in [HA10] provide a characterization of the different kinds of changes and the magnitude of the impact. These changes are mainly creating, modifying and possibly deleting architecture description elements.

We illustrate in Figure 4.5 the main concepts and relations concerning the interaction of

architecture patterns and architecture tactics. The figure is inspired in the model for patterns, quality attributes and tactics defined by N. Harrison et al. in [HA10], but it makes emphasis on the separation between the definition level of abstraction from the application level of abstraction. We compare both models later when we discuss related work in Section §4.3. Figure 4.5 presents that *architecture patterns* and *architecture tactics* are related by means of the *concerns* they affect. A pattern is not necessarily oriented to the resolution of a given concern, but rather, it is oriented to the resolution of a particular problem in a given context. However, a pattern affects a set of concerns as captured by the consequences (benefits and liabilities) of the pattern. A tactic is oriented to improve the level of a given quality attribute (which is a special kind of *concern*), but it may have side effects on other quality attributes. This interaction through concerns comprises the scenarios (A) and (B). The concerns affected by a pattern were addressed by means of tactics, in detriment of other concerns. An *architecture pattern* is composed by a set of *architecture pattern participants*. A pattern participant is any element type or relation type that characterizes the pattern's solution, and altogether, they conform the structure and behavior of the pattern. For example, in the case of the Pipes & Filter architecture pattern, *pipes* and *filters* are pattern participants, being pipes a special kind of *connector* and filters a special kind of *component*. An *architecture tactic* potentially affects some of these participants. In other words, the architect has the possibility of applying the given tactic on the pattern participants, once the latter have been reified (i.e. instantiated) in the architecture description. Hence, the relationship between *tactics* and *pattern participants* captures potential (possible) applications of tactics, not the tactic applications themselves. This interaction through pattern participants comprises the scenario (C3) and partially (B). While the center row of Figure 4.5 refers to *definitions*, the bottom row refers to *applications*. An *architecture pattern application* represents the fact that the architect applied a given pattern affecting a particular set of *architecture description elements*, i.e. creating, modifying or possibly deleting any element populating the *architecture models* of the *architecture description*. For example, architecture description elements can be the particular *filter* components that are connected by particular *pipe* connectors that conform the architecture of a given system-of-interest. The architecture description elements involved in the pattern application play any of the roles (element and relationship types) defined by the pattern participants. An *architecture tactic application* represents the fact that the architect applied a given tactic affecting a particular set of architecture description elements. The applied tactics are usually those that are defined as the potentially applicable tactics on the corresponding pattern participant. However, we do not enforce this assertion by means of an invariant to stress that an architecture tactic can be applied independently of the application of any architecture pattern. This interaction through architecture description elements comprises the scenario (C1). The scenario (C2) in which a pattern and a set of tactics are applied simultaneously is not illustrated in the figure. It might be added by introducing a *combined application* concept to be a special case of both *architecture pattern application* and *architecture tactic application*.

Architecture tactics are a fine-grained approach to architecture design [SK09]. They provide the building blocks for addressing quality attribute concerns both at a system-wide level and for the refinement of any particular element participating in the architecture of the system-of-interest. Architects usually apply architecture tactics in combination with architecture patterns, either simultaneously or after the patterns have been applied. However, tactics can be used independently of any particular pattern. However, the scope of tactics

goes beyond architecture design. Architecture tactics play an important role during architecture analysis or evaluation. As studied in [BBK02] when F. Bachmann et al. introduced the concept, each tactic provides an analytic model that can be used to determine to what extent the architecture built satisfies its significant quality attributes. Nevertheless, as our work is focused on the Decision Making activity of architecture design, our concern on architecture tactics refers to how they are actually applied by the architect to improve the architecture description of the architecture of the system-of-interest.

Conceptualization

Architecture patterns, *tactics* and *styles* are complementary architecture mechanism available to the architect in the practice of architecture design. As we discussed before, however, practitioners seldom grasp the sometimes subtle differences between these concepts. Such differences are in their purpose, their scope, their applicability, and the means for their definition and documentation. As a summary of this discussion, in Table 4.1 we present the characterizing questions that are generally answered by the definition of a concrete pattern, tactic, and style, and that are used by the architect to decide whether these concrete mechanisms can be applied to the particular scenario the architect is working on. The table shows that *architecture patterns* and *architecture tactics* are similar mechanisms. Their main difference lies in the kind of problem they address. While *architecture patterns* focus on recurring design problems, usually involving functional requirements, quality attribute requirements, and constraints, *architecture tactics* focus on improving the quality of a single quality attribute. *Architecture styles* differ from the other two concepts. Architecture styles are not focused on addressing particular design problems, but rather, on capturing recurring design forms by means of the recurring constructs that can be used to define architecture solutions in terms of interrelated elements and their behavior. As a consequence, architecture styles are not focused on any instantiation mechanism of particular solutions, as they are not targeted to any particular problem. However, capturing recurring design forms is also part of architecture patterns and tactics, as Table 4.1 shows.

A noticeably property of *architecture styles* is that they can be regarded both as *architecture design mechanisms* and as *architecture description mechanisms*. From the *design* perspective, an architecture style characterizes a family of systems that are related by shared structural and semantic properties [AAG93]. By deciding to use an architecture style, the architect is deciding that some aspects of the system-of-interest are governed by those structural and semantic properties. From the *description* perspective, an architecture style provides a specialized design language for a specific class of systems, typically by means of a vocabulary of design element types, design rules or constraints, a semantic interpretation, and analysis techniques that can be performed on architectures designed in that style [MKMG97]. Then, an *architecture style* captures the notations, semantics and tool support for designing and describing some concerns of the architecture of the system-of-interest. This characterization of architecture styles resembles that of *architecture description languages* (ADLs). As we reviewed in Section §3.1.2, an ADL captures notations, semantics and tool support of design languages that are used to build *architecture models*. However, the scale of an ADL is larger than the scale of an *architecture style*. On the one hand, an ADL packages a set of design languages, extension mechanisms to refine these languages, and tool support for visualization,

		Pattern	Tactic	Style
Context	In which context can it be applied?	✓		
Problem	What problem does it solve?	✓	✓	~
	In which cases can it be solved?	✓	✓	
Solution	In which terms is the solution expressed?	✓	~	✓
	Which constraints does it impose?	✓	~	✓
	How is the solution instantiated?	✓	✓	
	Why is it a solution?	✓	✓	
Consequences	What are the benefits?	✓	✓	✓
	What are the liabilities?	✓	✓	✓

Table 4.1: *Patterns, Tactics & Styles.*
The table presents a categorization of the concepts *architecture pattern*, *tactic* and *style*, in terms of the kinds of answers that characterize the definition of any of their concrete instances. These characterizing questions are organized in the main sections conforming the description of *architecture patterns*. While ✓ means that the definition of a concrete instance provides an answer for the particular question, ~ means that the answer is only partially or tangentially addressed.

editing, checking and analyzing architecture models expressed in these languages. On the other hand, an architecture style packages a single design language and the corresponding tool support for this language.

In the context of the conceptual model for the architecture description practice defined by the ISO/IEC/IEEE 42010:2011 standard [ISO11], we conceptualize an *architecture style* as the mechanism that affects one or more concerns, that provides a single *model kind* that captures the governing principles for *architecture models* following the architecture style and that provides the appropriate tool support for analyzing such architecture models. To this end, an architecture style may extend existing model kinds defined by more general styles. Figure 4.6 illustrates the concept of *architecture style* in our conceptual model. It is important to remark that our conceptual model makes explicit the distinction between *architecture style* and *model kind*. *Architecture style* is an concept of the Software Architecture discipline, and researchers and practitioners identify, capture, document and communicate them regardless if the style is conceived as a model kind or not. *Model kind* is a more general concept, that is applicable and used in the broader context of Software Engineering, as stated by the standard in [ISO11, Section 4.2.5]. Then, in addition to defining a model kind, an architecture style may capture, document and communicate additional information pertaining the architecture practice.

Architecture patterns have a broad problem-scope, covering any recurring design problem involving functional requirements, quality attribute expectations and design constraints. Each architecture pattern is situated in a particular context in which the problem usually arises, and provides a solution targeting this particular problem. A central part of the solution consists of identifying and defining the element types, their properties and relationships, and the constraints on their topology and behavior. This is actually the solution provided by an architecture style. Quoting R. Monroe et al. in [MKMG97], "an architecture style is probably better thought of as a design language that provides architects with a vocabulary and framework with which they can build useful design patterns to solve specific problems." While architecture styles provide the building-block design elements, rules and constraints

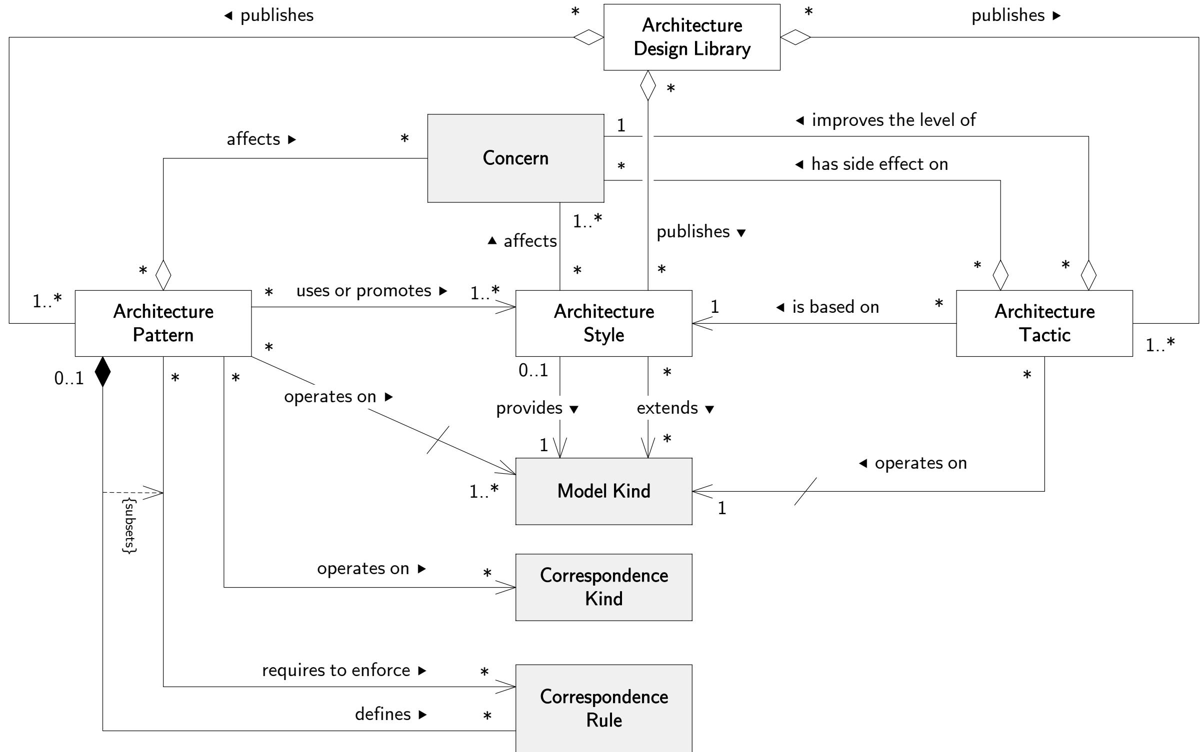


Figure 4.6: *Conceptual model of architecture patterns, tactics and styles.*
The figure illustrates the conceptual model we define to relate the concepts of *architecture pattern*, *architecture tactic* and *architecture style*, in the context of the conceptual model for architecture descriptions illustrated in Figure 3.2.

for composing the building blocks, an architecture pattern uses these building blocks to solve a specific problem within a set of architecture styles. Then, defining the architecture style or styles required or assumed is only part of the solution provided by an architecture pattern. In addition, an architecture pattern provides a generic yet parameterized instantiation of these element types in order to address the specific problem. Such an instantiation can only be illustrated by examples in the documentation of architecture patterns. To reify or materialize a concrete instantiation depends on the actual elements in the architecture description that are already defined or that need to be defined. As a consequence, an architecture pattern provides an design mechanism that embodies how to alter or update a set of *architecture models* in order to reflect or include the solution that the pattern mandates. This is a remarkable difference between patterns and styles. While styles offer potential design forms, patterns also guide how these design forms must be reified to address the problem in the concrete case decided by the architect.

Although architecture patterns are usually defined in terms of styles specializing or refining the *Component & Connector* style, this is not actually a requirement. Architecture patterns can be used on any set of architecture styles, as long as these styles are the most suitable to capture the effect of the pattern in the architecture being designed. For instance, the *Layers* pattern is actually a pattern that is applied to a style structured by the *Module* style. Moreover, architecture patterns have varied granularity. While a fine-grained pattern provides a solution that affects (is to be reflected on) a single *architecture model*, a coarse-

grained pattern provides a solution that affects multiple *architecture models*, and potentially the *correspondences* on these architecture models and the set of *selected correspondence rules* governing their relation. Such correspondence rules may be defined by the pattern itself, or they may be defined elsewhere (e.g. an architecture framework). While the effects of applying an architecture pattern lie on these system-dependent architecture assets, the architecture pattern must be defined in terms of system-independent architecture assets, particularly, those governing the potential system-dependent assets. We conceptualize an *architecture pattern* as the mechanism that affects a set of architecture *concerns*, that uses and promotes a specialized *architecture style*, that operates on a particular set of *model kinds* and *correspondence kinds*, and that defines and promotes the enforcement of *correspondence rules*. Figure 4.6 illustrates the conceptual model for *architecture patterns*. A pattern *operates on* model kinds and correspondence kinds in the sense that it defines a procedure to update the corresponding *architecture models* and *correspondences* to reflect the solution of the pattern, according to the particular scenario determined by the architect. This particular scenario determines the actual architecture models and correspondences, and provides any parameter value required to reify the parameterized generic solution defined by the pattern. For example, let us consider the Client/Server *architecture pattern*. Its solution promotes the Client/Server *architecture style* that defines two component types Client and Server. Also, the solution operates on the model kind for the Client/Server *architecture style*, and defines how a given component is decomposed in two or more components, one of them being the *server* and the others the *clients*. Then, to apply the Client/Server *architecture pattern*, the architect must indicate on which *architecture model* the pattern must be applied, which is the particular component to be decomposed, and which are the properties (e.g. name) of the new components to be created. We explain this in detail later in Section §4.1.2 when we discuss the application of patterns.

Architecture tactics have a narrower problem-scope than architecture patterns. Each architecture tactic addresses the problem of how to improve the level of satisfaction of a particular (single) quality attribute, making no additional assumptions on the problem or the context. The solution of architecture tactics presents the same characteristics than the solutions for patterns. However, accordingly to their narrower scope, they have an impact of a smaller scale. In addition to the defined or assumed architecture style to use, an architecture tactic defines a fine-grained modification of the *architecture models* that they are applied on. In other words, as is the case for architecture patterns, architecture tactics also define a mechanism to instantiate its solution in the architecture description. According to the revised definition of the *architecture tactic* concept defined by F. Bachmann et al. in [BBN07], an architecture tactic is an architecture transformation that affects the level of satisfaction of a particular quality attribute. Moreover, N. Harrison et al. in [HA10] characterize the interaction between patterns and tactics by the set of changes that need to be applied to the architecture description when applying a tactic to refine the application of a pattern. Then, we conceptualize an *architecture tactic* as the mechanism that improves the level of a particular *concern* possibly having a side effect on other concerns, that is based on a particular *architecture style*, and that *operates on* the particular *model kind* defined by the architecture style. We focus architecture tactics on a single model kind to remark the localized effect of architecture tactics. More complex tactics affecting multiple model kinds might be captured as patterns in our conceptualization. In Figure 4.6 we illustrate the conceptual model for *architecture tactics*.

Figure 4.6 provides a refinement of the interaction of architecture patterns and architecture tactics illustrated in Figure 4.5. In the latter figure, the interaction is defined by means of the *architecture pattern participants* that participate in architecture patterns and that are potentially affected by architecture tactics. As we discussed before, a pattern participant is any element type or relation type that characterize the pattern's solution, and altogether, they conform the structure and behavior of the pattern. In terms of Figure 4.6, the *architecture pattern participants* are represented by the *constructs* of the design language captured by the *model kind*, i.e. the constructs that represent the types of elements and relationships that can be used by *architecture models* governed by the model kind.

In our conceptualization, architecture styles are descriptive and architecture patterns and tactics are transformational. We claim that the community would benefit by separating these two concerns in architecture design. On the one hand, the focus of architecture styles lies in how to describe the recurring solutions or design forms found in practice, providing the possible tool support for analyzing solutions described in those terms. On the other hand, the focus of architecture patterns and tactics lies on how to populate or update the architecture descriptions expressed in terms of promoted architecture styles, the supporting rationale and the consequences of doing so. In our conceptual model, we use structural constraints to enforce the model kinds the architecture patterns and architecture tactics operate on, to be those defined by the architecture styles they promote. Also, we distinguish between definition and application. Architecture patterns and architecture tactics are the definition of mechanisms that can be later applied on particular architecture design scenarios. We discuss their application later in Section §4.1.2. Thus, patterns and tactics are conceived in direct relationship to architecture description concepts, such as model kinds and correspondence kinds, to facilitate their application in the Decision Making activity of architecture design.

As we discussed before, an *architecture pattern library* captures and communicates a set of architecture patterns, either with a general design purpose, or specific to a particular problem domain or system type. In our conceptual model, we consider the more general concept *architecture design library* to be the aggregation of sets of architecture design mechanisms, including architecture patterns, architecture tactics and architecture styles. Architecture design libraries might be related to other libraries defined elsewhere, providing then a means for cross-referencing. In addition, an architecture design library can provide classification and categorization of the aggregated elements, and it can define and characterize the interrelation among the elements defined in the library or in the referred libraries. The particular techniques to provide this additional information depends on the selection of authors of the library.

Figure 4.6 illustrates our conceptual model for the concepts *architecture pattern*, *architecture tactic* and *architecture style*, conceived in integration to the conceptual model of the ISO/IEC/IEEE 42010:2011 standard [ISO11] illustrated in Figure 3.2. Our conceptual model is independent of any particular approach to capture and document concrete mechanisms. Our focus is not the inner structures of their documentation, but mainly their interrelation and their conception in terms of the key concepts of the architecture description practice studied in Chapter §3.

4.1.2 Architecture Update Statements

The *architecture* of a system is not designed in one single step. Rather, it requires an iterative and incremental approach that produces the architecture by means of successive approximations. In each iteration, the *architect* identifies which are the most critical concerns among those that are still to be addressed, explores the solution space to devise potential alternatives, and decides which of these alternative solutions satisfies the identified significant concerns possibly at the expense of the other concerns. The iteration ends when all significant stakeholders are convinced that the designed architecture is the best in meeting their expectations given the characteristics of the development project. Capturing and communicating the *architecture* by means of an *architecture description* is not an after-the-fact activity. Rather, capturing the *architecture description* is the essential part of the decision making activity that reflects the effect of the decisions as soon as they are made. While the *architecture* is the intangible conceptualization of the fundamental characterization of the system being developed, the *architecture description* is the tangible representation of the *architecture* communicated to stakeholders for them to analyze and evaluate. The *architecture description* is the actual work product in the development project. Thus, at each iteration the architect refines the *architecture* of the system-of-interest, but materializes this refinement by updating the *architecture description* under construction in order to reflect the decisions made. This poses the question: which are the required updates and how are they materialized?

Architecture design methods provide guidance on this regard. The main purpose of these methods is to define the coarse-grained *tasks* that must be performed by the architect and other involved stakeholders, to achieve a satisfying *architecture* for the system-of-interest, captured and communicated by means of a complete- and detailed-enough *architecture description*. Some of these tasks involve visualizing and exploring the architecture description for analysis, while others require to update the architecture description in order to capture new information that is available to the architect, such as new concerns, new prioritizations of these concerns, new analysis models, new decisions made, among others. Considering the three major activities of an architecture design method that we illustrated in Figure 4.2, the Decision Making activity is the most intensive in updates to the architecture description, as opposed to the other activities that are intensive in visualizing and exploring and the description. However, architecture design methods provide little to no guidance on which fine-grained updates to the architecture description are required by the coarse-grained tasks they define. The architect figures out these required updates based on experience and expertise, constrained by the available time and tool support. Particularly, it is the tool support available in the development environment that preserves the knowledge on the potential fine-grained updates that can be performed. For text- and diagram-based architecture descriptions, generic editors are used. For rigorous and formal architecture descriptions, specialized domain-specific tools can be used, being the application of a pattern the most notorious example. Thus, the fine-grained updates are method-independent but tool-dependent. As a consequence, researchers defining architecture design methods lack a means to provide a finer characterization of their methods. In addition, software architects lack a tool-independent means to express or capture how they proceeded to design an architecture. Then, this knowledge cannot be directly reused as the architect only counts on the resulting architecture description, instead of counting on the procedure that lead to that architecture description.

To effortlessly reuse previous updates to the architecture description is a crucial productivity boost in a scenario in which the exploration of alternative solutions, going forward and backward, is recurrent. In what follows, we define a method- and tool-independent fine-grained set of atomic updates to architecture descriptions, that as such, can be regarded as an *architecture design language* to express how architecture descriptions are built.

The ISO/IEC/IEEE 42010:2011 standard [ISO11] provides a conceptualization of the architecture description practice. As we reviewed in Section §3.1.1, this conceptualization consists of a domain model of the practice that mixes real-world concepts and representation or descriptive concepts. We analyzed in Section §3.2 that adjusting the domain model to contain only those concepts that provide a representation of real-world concepts, then the domain model would be an accurate definition of the structural organization of architecture descriptions. Then, one might argue that any create-read-update-delete (CRUD) operation on the elements in that domain model would be an accurate set of fine-grained actions for architecture design. However, this is not actually the case. First, some of these CRUD operations are never or rarely used in architecture design. For instance, an architect does not create an *architecture viewpoint* during a development project. Instead, the architect adheres to an *architecture framework* or *architecture description language* which provides the *architecture viewpoints* to use. As we discuss later, organizations with proprietary architecture viewpoints and model kinds should consider developing their own architecture frameworks or architecture description languages independently of any particular development project, favoring encapsulation and reuse. Second, some intensively used operations are missing in these CRUD operations. For instance, the architect systematically applies patterns and tactics to address the significant architecture concerns of the system. Applying a pattern is not simply about updating (modifying) an *architecture model* in the architecture description. As we discussed before, an architecture pattern operates on architecture models, correspondences, and the set of selected correspondence rules, possibly more than one at the same time. Thus, several CRUD operations are involved in the application of a pattern. Third, and even more important, CRUD operations are not at the appropriate level of abstraction. CRUD operations are low level update actions that are tightly tied to the structure of architecture descriptions. They are not conceived in terms of architecture design concepts and techniques. The case of applying an architecture pattern is an example of the mismatch in the abstraction level. Then, CRUD operations are inspiring, but do not provide an accurate definition of atomic updates to architecture descriptions from the architecture design perspective.

We define an *architecture update statement* as the prescriptive and descriptive *statement* of an *update action* that can be performed on an *architecture description*. An architecture update statement is prescriptive in the sense that it can state an update that is to be performed on an architecture description, and it is descriptive in the sense that it can state an update that was performed. Figure 4.7 illustrates the role of an architecture update statement as an architecture design step. In the figure, the version i of an architecture description is updated by means of a non-empty sequence of *architecture update statements*, producing the i + 1 version of the architecture description. In a broader example, arrows in Figures 4.3 and 4.4 represent architecture update statements that produce new versions of an architecture description from a predecessor version. Figure 4.8 illustrates the different kinds of architecture update statements we define. The conceptualization of each of these kinds is illustrated in Figures 4.9 to 4.19. We use the suffix Statement for the abstract concepts in the

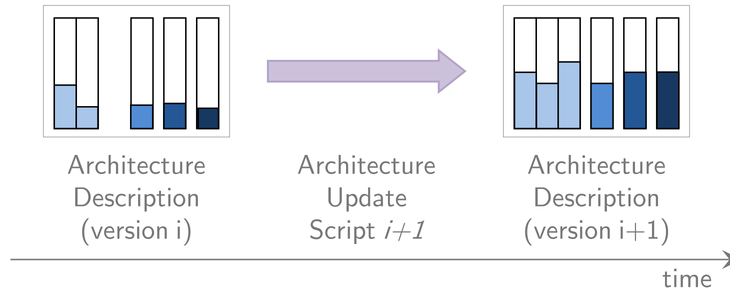
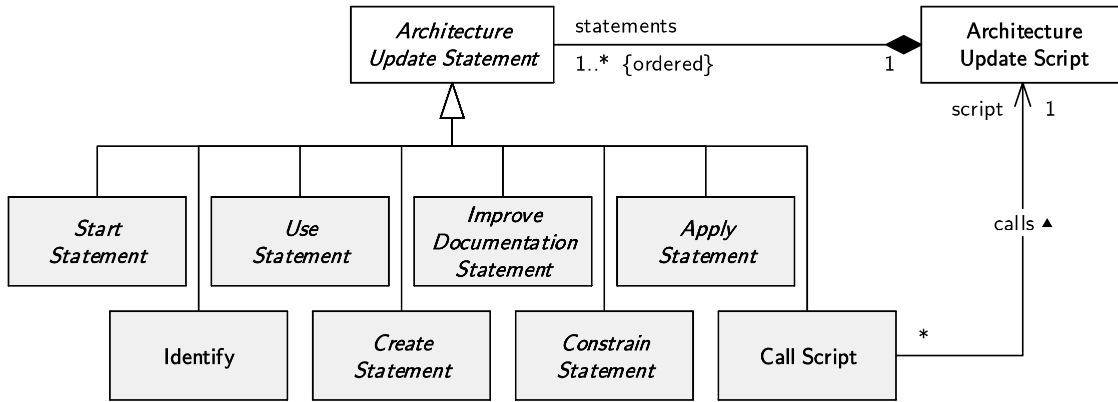


Figure 4.7: *A design step as architecture update statements.*

The figure illustrates the role of an architecture update statement as an architecture design step. Given the i-th version of an architecture description, an *architecture update script* consisting of a non-empty sequence of *architecture update statements*, determines the modifications that are to be or were performed on the architecture description to produce its $i + 1$ version.

conceptual model, but we omit the suffix for concrete concepts for brevity and clarity. There are three major categories of architecture update statements. *Start* statements provide the starting point for the architecture design activity. An architect uses them to set up the initial architecture description, possibly using some externally defined artifacts, that is to be successively refined until the satisfying "complete" architecture description is achieved. *Atomic* statements are the minimal updates that can be performed on an architecture description. These minimal updates can be either structural or non-structural. Atomic *structural* updates are those that determine the structural organization or the architecture description, such as adhering to an architecture framework or architecture description language, or creating a new architecture model. Atomic *non-structural* updates are those that determine the architecture description elements populating the structure, such as identifying concerns and stakeholders, and adding and modifying the elements defined in the architecture models and correspondences. Finally, *control* statements determine the order in which architecture update statements are performed.

We do not define a full-featured action language in terms of control statements. We restrict to *sequences* and non-recursive non-parameterized *calls*, excluding conditionals, loops and recursion. We define an *architecture update script* as the named encapsulation of a non-empty sequence of architecture update statements, as illustrated in upper right corner of Figure 4.8. We use scripts later in Section §4.1.3 when we define *solutions*. An architect can benefit from a script as it might be reused in more that one path of the exploration of the solution space. When weighting alternative solutions, the same script might be applied in more than one of these alternatives, and thus, reducing the effort for capturing them. We define a *call script* statement as the momentary redirection of the control flow to the statements defined by the script being called, and the redirection backwards when all those statements were executed. The call mechanism we provide is basic: there is no parameterization of scripts, no argument passing, and no recursion. Figure 4.8 illustrates the *call script* statement in the lower right corner, and uses invariants to restrict circular references in nested calls. By this means, we provide minimal reuse capability without introducing complexity to the language. From the semantical point of view, this complexity would have impacted in our semantic specification of Section §4.2.3 in which variable declaration and argument passing would require handling a separate state and environment, and loops and recursion would require using fixed-point constructs to deal with infinite recursion and no termination. A full-featured action language




```
context ArchitectureUpdateScript
  def: UsedScripts : Set(ArchitectureUpdateScript) =
      self.statements→select(stm | stm.oclIsKindOf(CallScript))
        →collect(cstm | cstm.script)

  def: AllUsedScripts : Set(ArchitectureUpdateScript) =
      self.UsedScripts→closure(s | s.UsedScripts)

  inv: self.AllUsedScripts→excludes(self)
```

Figure 4.8: *Conceptual model of architecture update scripts and statements.*
The figure illustrates the different kinds of ArchitectureUpdateStatement we define in the conceptual model. The conceptualization of these kinds of statements is illustrated in Figures 4.9 to 4.19. Additionally, the figure illustrates the modularization technique we define for architecture updates. ArchitectureUpdateScript encapsulates a named and reusable non-empty sequence of statements that can be called by name using the CallScript statement. The invariant excludes circular references in the calls to avoid recursion in their execution.

is out of the scope of our work. We are mainly concerned with the atomic updates that can be done on an architecture description and their semantics in terms of the formal denotation for architecture descriptions that we defined in Section §3.2.2.

In what follows, we discuss each kind of *start* and *atomic* architecture update statement. Architecture patterns, tactics and styles that we discussed before in Section §4.1.1, are used at the end of this section when we define the *apply* statements. Architecture update statements are used later in Section §4.1.3 to conceptualize solutions, decisions and rationale. The model-based interpretation of these constructs is defined later in Section §4.2.3.

Start Statements

The kick-off of following an architecture design process is to set up the environment with an initial possibly empty version of the architecture description. In Figure 4.3, we illustrate this starting point as the first arrow sourced in an empty environment, denoted by $\varnothing$, and targeting an initial architecture description. *Start* statements allow the architect to take this

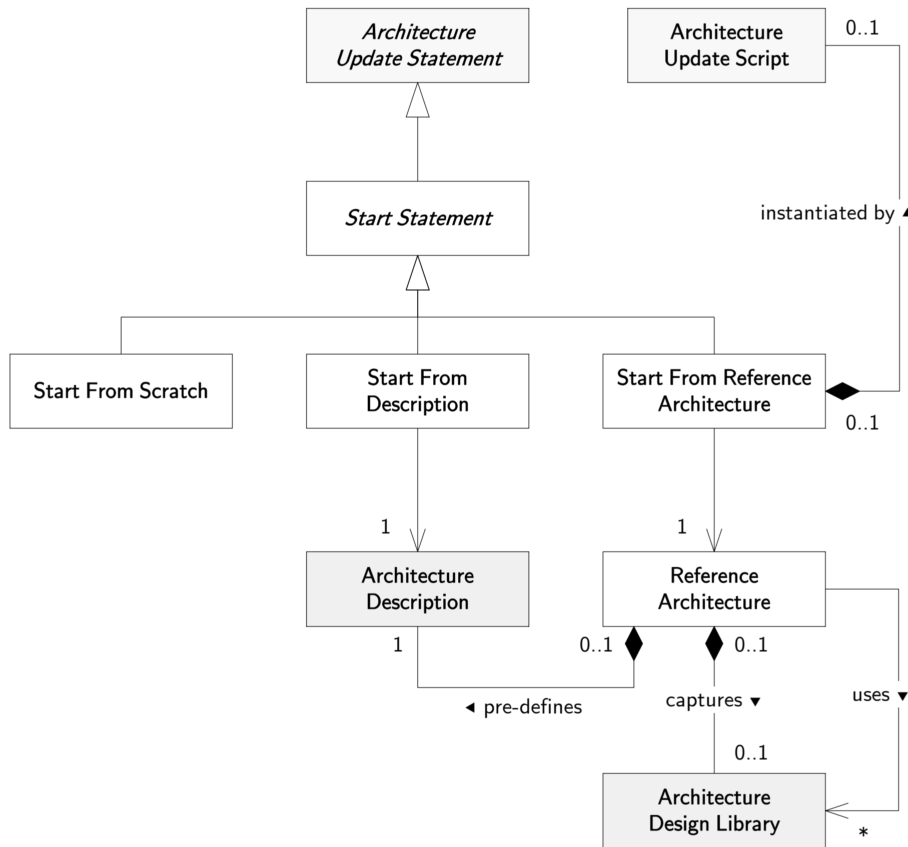


Figure 4.9: *Conceptual model of the* Start *statements.*

The figure illustrates the conceptual model for the different kinds of StartStatement. A *start* statement is the starting point of an architecture design process by which the initial *architecture description* is obtained, and which is later incrementally built by means of the other kinds of architecture update statements.

initial step by creating the initial work products aggregated in the architecture description to be built. This kind of statements can only be used once during architecture design, an it must be used before any other statement. We define StartStatement as the general kind of these statements, and we define three different concrete statements representing different starting scenarios. Figure 4.9 illustrates the conceptual model for these statements.

We define the StartFromScratch statement as the concrete means to create an initial empty architecture description. It only sets the minimal environment for the architecture design process to take place. Particularly, it produces an empty architecture description with no aggregated elements conforming it. One might argue that this scenario is rare in practice as the architect usually creates the initial version architecture description with a pre-sketched architecture before proceeding with the architecture design process. If this pre-prepared sketches are informal, such as drafts in paper or whiteboards, the architect actually starts with an empty architecture description and then subsequently refines it by reproducing what was sketched. If the sketches are expressed in a formal language that is not part of the architecture description, i.e. no model kind provides this language, the architect faces the same scenario as for informal sketches. In this case, however, some kind of automation might be achieved to reproduce the sketches using an assisting tool. In the case that the sketches

are expressed in terms of model kinds that are actually part of the architecture description, the architect then has a pre-existing architecture description from which to start from. The StartFromArchitectureDescription statement should be used in this case.

We define the StartFromArchitectureDescription statement as the concrete means to replicate an existing architecture description to create the initial version of the one that the architect will build. This pre-existing architecture description can be originated from different sources. First, it might have been created manually, possibly assisted by specific tools, either as an initial sketch for the architecture to be built or as the result of a previous architecture design effort. Second, it might have been the result of a previous project that needs to be maintained or evolved, or even from a parallel project developing another system for a common platform or on a common structure. Third, it might have been a discarded alternative solution (architecture) from a past architecture design effort that re-emerged as suitable for the system-of-interest at hand. In any of these cases, the StartFromArchitectureDescription sets up the initial architecture description as a replica of the one being provided.

We define the StartFromReferenceArchitecture statement as the concrete means to start an architecture design effort from a reference architecture that is available in the development organization. A *reference architecture* is a complex work product containing a consistent set of architectural best practices intended to be used in every development project in an organization that targets similar systems. The concept of reference architecture is novel in the business and development world. Although architects tend to use the term in practice, the notion is not concisely defined or understood. As studied by R. Cloutier et al. in [CMV+10], the purpose of a reference architecture is to provide guidance for future developments. It incorporates the vision and strategy for the future. A *reference architecture* is a reference for the multiple teams related to ongoing developments. It defines a customer or organization context, the business architecture and the technical architecture. It captures both a predefined (instantiated) architecture description that can be used as the foundation for the architecture of future developments, and a set of design principles, guidelines and mechanisms that can or should be used to build architectures from the reference architecture. In order to capture this notion, we conceptualize a ReferenceArchitecture as a predefined ArchitectureDescription and an ArchitectureDesignLibrary offering the related design mechanisms to use. *Architecture frameworks* and *architecture description languages* are usually part of a reference architecture. However, we do not include them explicitly in our conceptual model as they are captured by the *architecture description* by means of the *adheres-to* relation. Then, the StartFromReferenceArchitecture statement uses an existing ReferenceArchitecture and an ArchitectureUpdateScript that instantiates or reifies any generic or template element populating the *architecture description* of the reference architecture, to their corresponding ones in the current project. This update might have been captured separately of this *start* statement, for instance, as the subsequent updates after starting. However, we prefer to add an optional instantiating script to allow the architect to separate reference architecture instantiation from the required updates concerning the actual subsequent design of the architecture. The effect of this statement is to set the architecture description of the reference architecture as the initial architecture description, and then applying the update script to this architecture description. The architecture design library of the reference architecture is used in the script to produce the initial instantiation. Additionally, the architecture design mechanisms it provides remain available to the architect for later use, if appropriate.

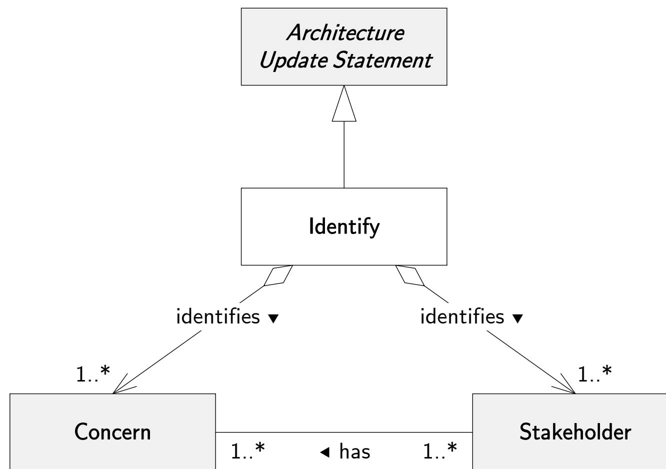


Figure 4.10: *Conceptual model of the* Identify *statement.*

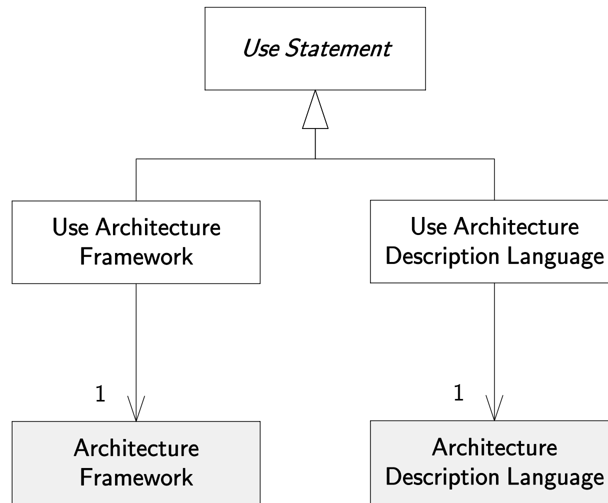
The figure illustrates the Identify statement used to capture the identified *concerns* and *stakeholders* that are significant to the architecture.

Identify Statement

The Requirement Analysis activity of an architecture design method is responsible for the discovery, elicitation and prioritization of the architectural significant concerns that are of interest for the critical stakeholders. These concerns can be critical functionality, expectations on quality attributes, constraints on the system, on its environment and on the development project, the agreements made with the stakeholders, and the potential risks of failure. The architect captures the significant stakeholders and the critical concerns in the architecture description. We define the Identify statement to be the concrete kind of update statements that capture this significant information into the architecture description.

There are several alternatives to conceptualize the Identify statement. Provided the purpose of capturing new concerns and stakeholders, or of modifying the properties of already identified concerns and stakeholders, or their interrelation, any create or update action on elements of these concepts would have served that purpose. However, these actions are too fine-grained to be practical in a development scenario. In practice, the architect lists the fresh information to be added or updated in the architecture description. Then, we define a single kind of statement Identify as an aggregation of the Concern and Stakeholder identified or redefined, their properties and their interrelations. The effect of the statement is to update the architecture description to capture the information provided. As some of the provided concerns and stakeholders may have been previously identified, and consequently they already exist in the architecture description, the effect of the statement is to update the architecture description according to the provided information. Figure 4.10 illustrates the conceptual model of the Identify statement.

The Identify statement is not the only mechanism that the architect has to capture concerns and stakeholders in the architecture description. In practice, *architecture frameworks* and *architecture description languages* define the set of general concerns they identify and tackle, and the set of stakeholders they identify and serve. Then, as we discuss later, by



context UseArchitectureDescriptionLanguage **inv:**
  let adl = **self**.architectureDescriptionLanguage **in**
    adl.architectureViewpoints.modelKinds→includesAll(adl.modelKinds)

Figure 4.11: *Conceptual model of the* Use *statements.*
The figure illustrates the different kinds of Use statements that allow to capture the use or adherence to *architecture frameworks* or *architecture description languages*. The invariant states that in the case of architecture description languages, only those which organize all their defined *model kinds* in terms of *architecture viewpoints* can be actually used in an architecture description. The invariant is expressed in terms of the conceptual model for architecture description languages illustrated in Figure 3.5.

adhering to any particular architecture framework or architecture description language, the sets of concerns and stakeholders they identify are also captured as concerns and stakeholders captured in the architecture description. We expect the architect to use the Identify statement mainly to capture the concrete concerns and stakeholders that are particular to the system being developed.

Use Statements

The architect is responsible for deciding how the architecture description is structurally organized in terms of *architecture views* and *architecture models*. As we reviewed in Section §3.1.1, architecture views are governed by *architecture viewpoints*, and moreover, an architecture description has exactly one single architecture view for each architecture viewpoint it uses. In turn, architecture models are governed by *model kinds*. In this case, a single model kind can be used to govern multiple architecture models, aggregated in one or several architecture views. However, the architect rarely conforms this structural organization from scratch. As we studied in Section §3.1.2, *architecture frameworks* and *architecture description languages* are architecture mechanisms for capturing reusable system-independent knowledge on how to organize and conform *architecture descriptions*. When the architect adheres to an architecture framework or architecture description language, the architecture description being built gets organized according to the definitions of the selected mechanism. We define

the UseStatement as the kind of statements that allows the architect to capture the use or adherence to a particular architecture description mechanism. Figure 4.11 illustrates the conceptual model of the different kinds of UseStatement.

We define the UseArchitectureFramework statement as the concrete means to make the architecture description to adhere or use a given architecture framework. We define the UseArchitectureDescriptionLanguage statement as the concrete means to adhere to a given architecture description language (ADL). However, not every architecture description language can be directly adhered to. It is mandatory for an architecture description to have all its model kinds aggregated by the architecture viewpoints being used. This is not the case for all ADLs. An ADL can define model kinds without organizing them in architecture viewpoints [ISO11, Section 6.3]. The purpose of these ADLs is to be used to define other ADLs and mainly architecture frameworks, not to be directly used in architecture descriptions. We enforce this restriction by means of an invariant in Figure 4.11. The effect of these statements is to include in the architecture description all the definitions captured by the reusable mechanism, i.e. the architecture framework or ADL being used. These definitions include concerns, stakeholders, architecture viewpoints, model kinds, and correspondence rules. For each architecture viewpoint, the corresponding single architecture view is also included in the description. However, no architecture model is included in the architecture view. It is up to the architect to decide which architecture models need to be created in each architecture view. The architect can achieve that by means of the CreateArchitectureModel statement that we discuss later.

Model kinds defined in architecture frameworks and ADLs usually provide a concrete syntax for the language they define, together with language-specific visualizers, editors, and other supporting tools. Once the architect adheres to any architecture mechanism defining a model kind, these sets of tools become available to the architect. Visualizers, checkers and analysis tools are usually based on *read* functionality, and consequently no *update* to the corresponding architecture models is required. However, language-specific editors allow the architect to *manually* update the architecture models conforming the architecture description in an uncontrolled fashion. Manual modifications undermine the purpose of our conceptualization: to define a tool-independent mechanism to capture the atomic updates that can be performed on an architecture description during architecture design. We do not provide any special treatment for manual modifications. We require the architect to capture by means of an architecture pattern the actual modifications that are needed, and to apply them by means of an ApplyPattern statement that we introduce later in this section. Tools capturing the difference between artifacts may help the architect to derive the update actions performed on an architecture model after manually modifying it, from which to inspire to conceive the corresponding architecture pattern. It is important to recognize the advantage of defining a pattern over manual modifications: a pattern can be reapplied with much less effort that manually reproducing the modifications. We proceed similarly for the case for architecture design experts, such as ArchE [BBKS04]. An architecture design expert or assistant elicits key quality attribute information from functional and quality requirements, derives candidate architectures, evaluates them, identifies tradeoffs and suggests alternative solutions. We do not provide special treatment for these kinds of tools either. The architect may use them for assistance in devising alternative solutions and weighting their pros and cons, but needs to update the architecture description accordingly in order to reflect this in-

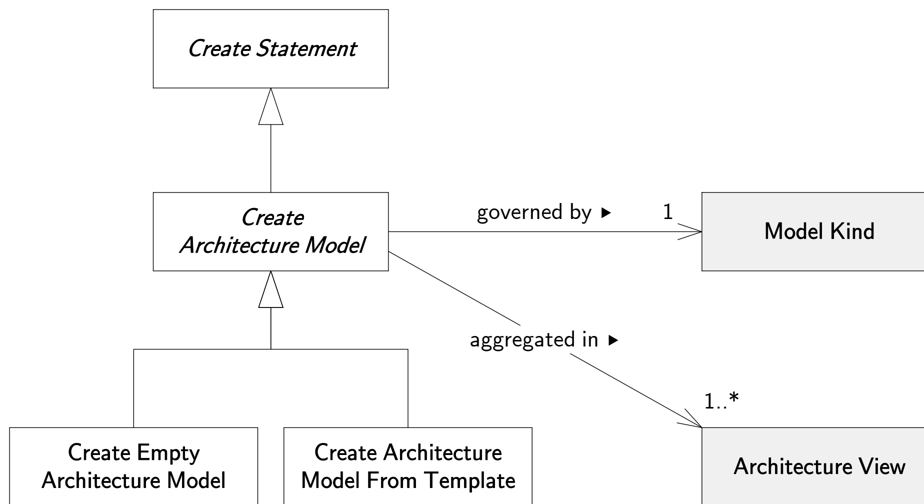
formation. These updates are performed by means of the *architecture update statements* we provide. It is important to remark, however, that there is an opportunity for a fine-grained integration and interoperation with this kind of tools. Such integration must be both at the architecture description level (i.e. the structural organization and the means to reify it in a given technology) and at the architecture design level (i.e. to express the expert's output in terms of the atomic statements of our conceptualization).

We allow the architect to use or adhere to one or more architecture frameworks and architecture description languages at the same time. The main purpose of multiple adherence is to allow the definition of small purpose-specific architecture frameworks and description languages that can be used in combination if needed. However, this is not actually a common practice in architecture design. The main reason is that when using more than one architecture framework or description language, reconciliation issues arise. For instance, concerns may be framed by more than one architecture viewpoint or model kind, different languages are made available for the same purpose, among others. The Use statements we define do not take into account any reconciliation issues. If the architecture viewpoints are defined by a different set of artifacts (with respect to a given equality relation), then the two architecture viewpoints will be added to the architecture description regardless that they share the same purpose or intention or that they frame the same concerns). It is up to the architect, and mainly to the development organization, to create their organization-specific architecture frameworks as the reconciliation of existing frameworks and description languages available in the community, prior to using them in development projects.

Create Statements

The Use statements discussed before allow the architect to capture the governing principles for the architecture description being built. By adhering to an architecture framework or an architecture description language, the architect determines which architecture viewpoints and model kinds are used to define architecture views and architecture models. As we discussed before, an architecture description has a single architecture view for each architecture viewpoint. However, several architecture models can be defined for a single model kind. It is up to the architect to decide which are the actual architecture models needed to describe the architecture. As we reviewed in Section §3.1.1, the architecture description also contains a set of *correspondences* to capture relations among architecture description elements. This was illustrated in Figures 3.2 and 3.3 in Chapter §3. Then, we define the CreateStatement as the kind of statements that allow the architect to create new architecture models and correspondences in the architecture description.

We define the CreateArchitectureModel statement as the means to create a new *architecture model* in the architecture description. Figure 4.12 illustrates the conceptual model for the different kinds of this statement. The architect must specify which *model kind* will govern the architecture model to create. As we discussed in Section §3.1.1, an architecture model can be aggregated in more than one architecture view. Then, the architect must also indicate which architecture views will aggregate the new architecture model. Each of these architecture views, however, must be governed by an architecture viewpoint that aggregates the specified model kind. We enforce this restriction by means of an invariant in



context CreateArchitectureModel **inv:**
self.modelKind.architectureViewpoints
→includesAll(**self**.architectureViews.architectureViewpoints)

Figure 4.12: *Conceptual model of the* CreateArchitectureModel *statements.*
The figure illustrates the different kinds of CreateArchitectureModel statements that allow to add to the *architecture description* a new *architecture model*, either empty or by means of a template defined by the *model kind*.

Figure 4.12. There are two different mechanisms for creating a new architecture model. We define the CreateEmptyArchitectureModel to create an *empty* architecture model that defines no architecture elements. We define the CreateArchitectureModelFromTemplate to use one of the *templates* defined by the model kind to create the architecture model. The effect of these statements is that a new architecture model was added to the architecture description, it is marked as governed by the specified model kind, and it is aggregated in the specified architecture views. In the case of using a template, the new architecture model is populated according to what is specified in that template.

We define the CreateCorrespondence statement as the concrete means to create a new *correspondence* in the architecture description. Figure 4.13 illustrates the conceptual model for this statement. The architect must specify the set of *architecture description elements* that are related by the correspondence being created. As we reviewed in Section §3.1.1, an architecture description element is any kind of construct in the architecture description, both those defined by the conceptual model in the ISO/IEC/IEEE 42010:2011 standard, and those introduced by the modeling languages defined by architecture viewpoints and model kinds. The new *correspondence* will capture relations between instances of these architecture description elements. The architect can optionally indicate the *correspondence kind* governing the new correspondence. We explicitly introduced this concept to allow the factorization of the definition of kinds of correspondences. As we explained in Section §3.2.1, even though this concept is not part of the ISO/IEC/IEEE 42010:2011 standard, it serves the purpose of encapsulating system-independent architecture knowledge on how to govern the definition of correspondences, that can be then reused in multiple scenarios. When the architect specifies an already defined correspondence kind, it is used to govern the correspondence to be created.

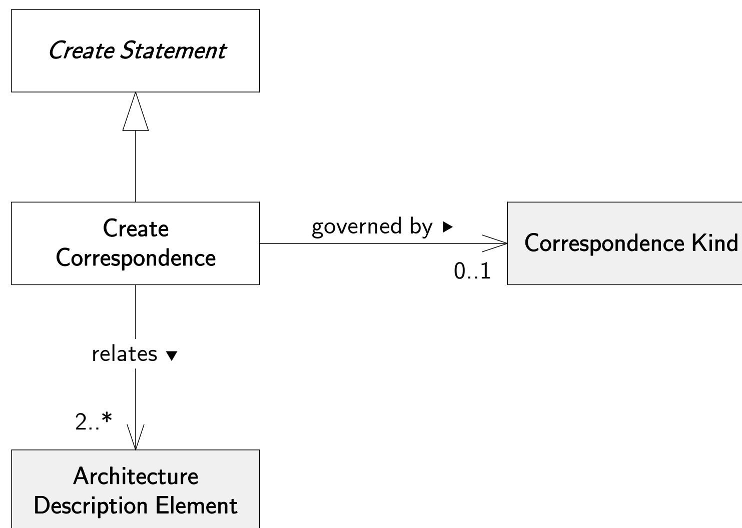


Figure 4.13: *Conceptual model of the* CreateCorrespondence *statement.*
The figure illustrates the CreateCorrespondence statement that allows to create a new correspondence relating two or more architecture description elements in the architecture description. An already existing CorrespondenceKind can be used, or one can be directly provided by the statement.

When no correspondence kind is specified, the architect still has to define how the new correspondence is to be "modeled", i.e. which will be the structure of the links captured by the new correspondence.

Improve Documentation Statements

An *architecture description* is not a single work product or artifact. It is a complex work product consisting of an aggregation of several interrelated work products, such as *architecture viewpoints*, *model kinds*, *architecture views*, *architecture models*, *correspondences*, and *correspondence rules*. In turn, these work products are also complex. Some of them consists of an aggregation of other work products, like architecture viewpoints and architecture views. Others consist of a structured catalog of interrelated elements, such as architecture models and correspondences. In addition to this complex well-structured content, all of these work products contain additional *documentation* that provides stakeholders with both contextual information regarding the work product, and internal information regarding the elements populating the work product's content. The documentation of these work products may include (i) administrative information such as name and title, overview, date and version, authors and affiliation, (ii) acronyms, definitions, reference materials and bibliographic references, and (iii) technical information such as usage examples, application scenarios and restrictions, general considerations, among others. The actual documentation depends of the work product and the practitioner community creating it.

During architecture design, the architect is responsible for creating and maintaining the documentation of the work products, mainly those created specifically for the system-of-interest. The architect does not create all the work products that populate an architecture description. For instance, the architect obtains system-independent *architecture viewpoints*

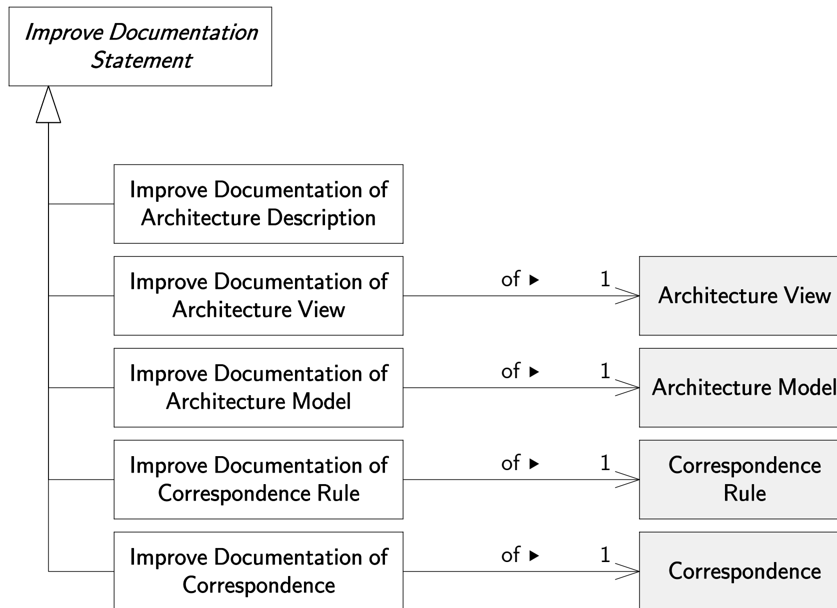


Figure 4.14: *Conceptual model of the* ImproveDocumentation *statements.*
The figure illustrates the different kinds of ImproveDocumentation statements that allow to add and modify any of the items or slots that are used to document the different constructs participating in the *architecture description*.

and *model kinds* from reusable mechanisms such as architecture frameworks and architecture description languages. Those work products are already documented by the practitioner community that created them. The architect actually creates *architecture views*, *architecture models*, *correspondences* and *correspondence rules*, together with the *architecture description* itself.

We define the ImproveDocumentationStatement as the kind of statements that allow to improve, refine or update the documentation of those work products created by the architect. We also define a concrete kind of statement for each kind of work product that can be created by the architect. Figure 4.14 illustrates the conceptual model of these statements. For each statement, the architect must indicate which is the actual work product whose documentation is being improved. For instance, for the ImproveDocumentationOfArchitectureView statement, the ArchitectureView must be indicated. In the particular case of the ImproveDocumentationOfArchitectureDescription statement, there is no need to specify on which work product to operate on as there is a single architecture description being built in the development environment.

In practice, to improve the documentation of a work product is a *manual* activity. The architect uses the corresponding editor tool to update the documentation as appropriate. Any of the ImproveDocumentation statements capture this fact. As we analyze later in Section §4.2.3, our goal is to provide a shareable and repeatable procedure to capture the improvement of the documentation of a work product, instead of relying of tool-specific manual edition. As a consequence, instead of producing the modified documentation, the architect captures how the documentation is to be modified.

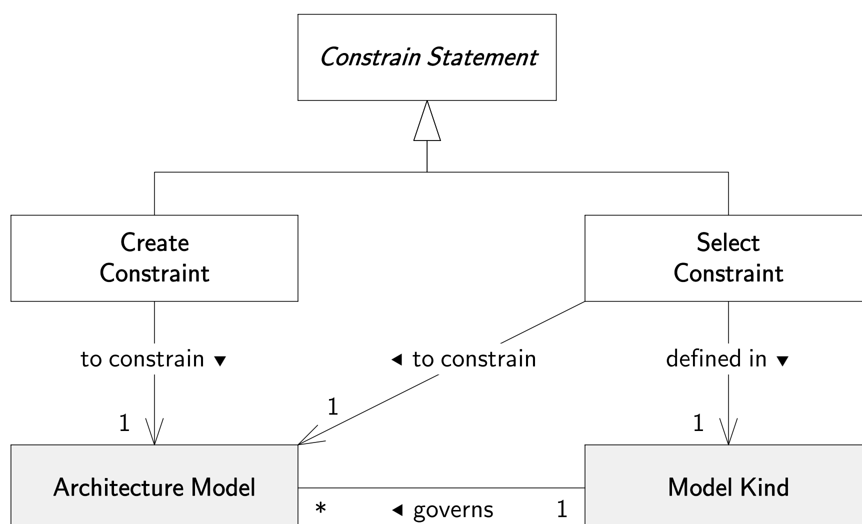


Figure 4.15: *Conceptual model of the* Constrain *statements on architecture models.*
The figure illustrates the different kinds of Constrain statements that allow to create and select a *constrain* on an *architecture model*. Constrain statements on correspondences are illustrated in Figure 4.16.

Constrain Statements

An architect can impose *constraints* on the architecture description to further characterize its consistency. A constraint is a property or assertion that is expected to hold for the architecture description. If any of these constraint assertions is broken, then the architecture description is in an inconsistent state. We define the ConstrainStatement as the kind of statements that allow the architect to capture which constraints are expected to hold. There are two different kinds of constraints, one ruling on a single *architecture model*, and another ruling on any interrelated set of *architecture description elements* in the architecture description. The latter is in fact the case of *correspondence rules*.

A *model kind* defines the conventions for building a specific kind of architecture model, and particularly, it defines the modeling language providing the constructs for building such architecture models. As we discussed in Section §3.2.2, the model kind uses invariants to rule out models that in spite of being well-structured, they are not actually valid in the sense of the model kind intention or domain. However, the architect may decide to further constrain an architecture model to establish the conditions for well-formedness of that particular model. Enforcing this condition helps the architect to detect inconsistencies in the architecture models that might be introduced by any later decision. Then, we define the CreateConstraint statement as the concrete means to create a new constraint on a given architecture model. The effect of this statement is to record in the architecture description that a given assertion is expected to hold. As a model kind may define a common set of potential constraints that might be applied in the architecture models it governs, we define the SelectConstraint as the concrete means to record that the already defined constraint is expected to hold in an architecture model. Figure 4.15 illustrates the conceptual model for these statements.

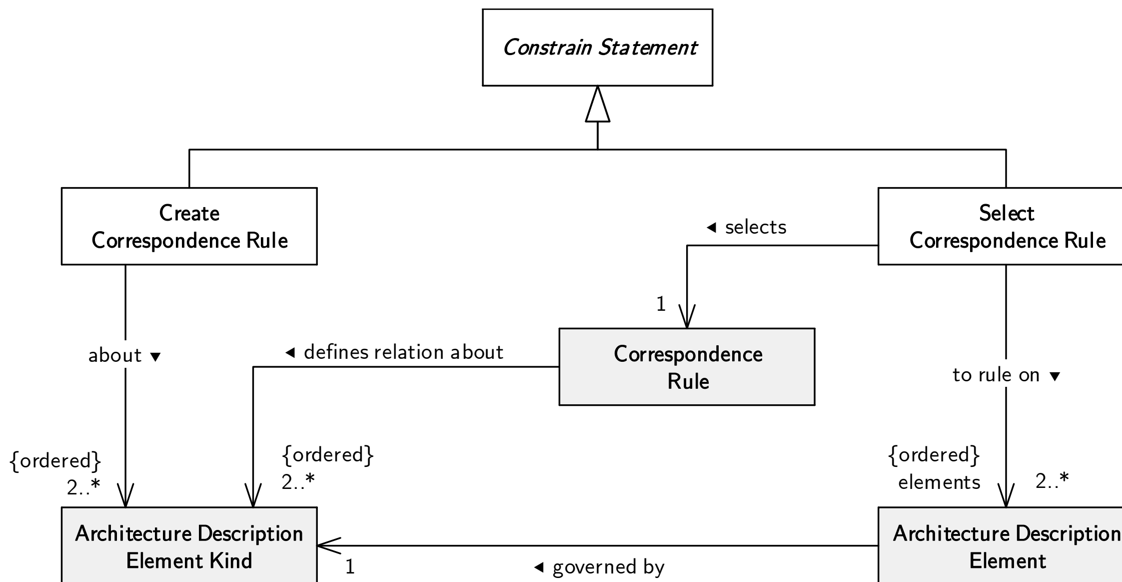
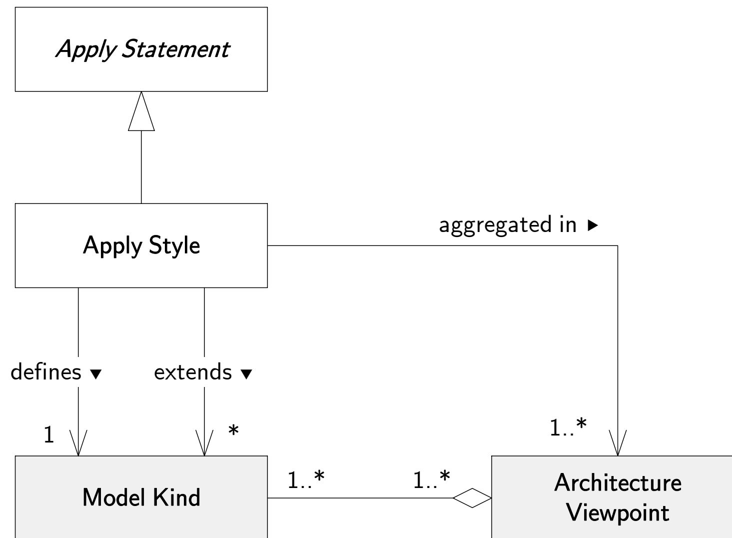


Figure 4.16: *Conceptual model of the* Constrain *statements for correspondence rules.*
The figure illustrates the different kinds of Constrain statements that allow to create and select *correspondence rules*. Constrain statements on architecture model constraints are illustrated in Figure 4.15.

Correspondence rules are used to enforce relations within an architecture description. They represent a rule or assertion that is expected to hold on the interrelated architecture description elements. As we discussed in Section §3.2.2, while a *correspondence* defines a relation by extension, i.e. by enumerating the links between the related elements, a *correspondence rule* defines a relation by intention, i.e. what the architect expects to be true on the related elements. Correspondence rules are reusable assets that are captured by architecture frameworks and architecture description languages. When the architect decides to adhere to any of these architecture description mechanisms, the correspondence rules they define become available for the architect to be applied on any particular *architecture description element* in the architecture description being designed. Then, we define the SelectCorrespondenceRule statement as the concrete means to select a correspondence rule and record it to be enforced on a given set of architecture description elements. As the architect can decide to capture a *correspondence* between architecture description elements, by means of the CreateCorrespondence statement, the architect can also decide to define a new *correspondence rule* and apply it to a given set of architecture description elements. We define the CreateCorrespondenceRule statement as a concrete means to achieve this purpose. Figure 4.16 illustrates the conceptual model for these statements.

Apply Statements

Architecture patterns, tactics and styles are architecture design mechanisms that the architect uses in combination to define the structural and behavioral fundamental characterization of the system-of-interest. These mechanisms capture reusable system-independent architecture knowledge on how to design an architecture. However, they do not conform an architecture description by themselves, but rather, the architect has to *apply* them on architecture models



context CreateModelKindExtension **inv:**
self.modelKind.architectureViewpoints→includesAll(**self**.architectureViewpoints)

Figure 4.17: *Conceptual model of the* ApplyStyle *statement.*
The figure illustrates the ApplyStyle statement that allows to create a system-specific style possibly by extending existing styles captured in some of the *model kinds* in the architecture description.

and correspondences to define specialized kinds of elements and relations, and to populate existing architecture models with instances of these element and relations kinds, according to recurrent well-defined architecture structures and solutions that address specific concerns and whose benefits and liabilities are known beforehand. We define the ApplyStatement as the kind of statements that allows the architect to apply an architecture mechanism on the architecture description being built.

As we discussed in Section §4.1.1, we conceptualize an *architecture style* as the architecture design mechanism for capturing the governing principles for a specific kind of *architecture models*, possibly extending already existing *model kinds*. An architecture style defines the kinds of elements, their properties and relations, that participate in a recurrent or common design solution. Pre-cooked architecture styles are usually provided by architecture frameworks and architecture description languages, in the form of model kinds. By means of a Use statement discussed before, the architect includes these predefined styles in the architecture description being built. In practice, the architect may need to specialize these predefined styles to capture more concrete and system-specific kinds of elements and relations that are then used in architecture models to define the characterizing structure and behavior of the system-of-interest. There are two different approaches to define new styles, strongly depending of the capabilities of the architecture description languages being used. On the one hand, (a) the architecture description language provides constructs for both defining *new specialized types* and defining *instances of types*. On the other hand, (b) the architecture description language provides constructs only for defining *instances of types*. For example, consider an architecture description language for the Component & Connector style. This language can provide constructs to define specific component instances and connector instances that

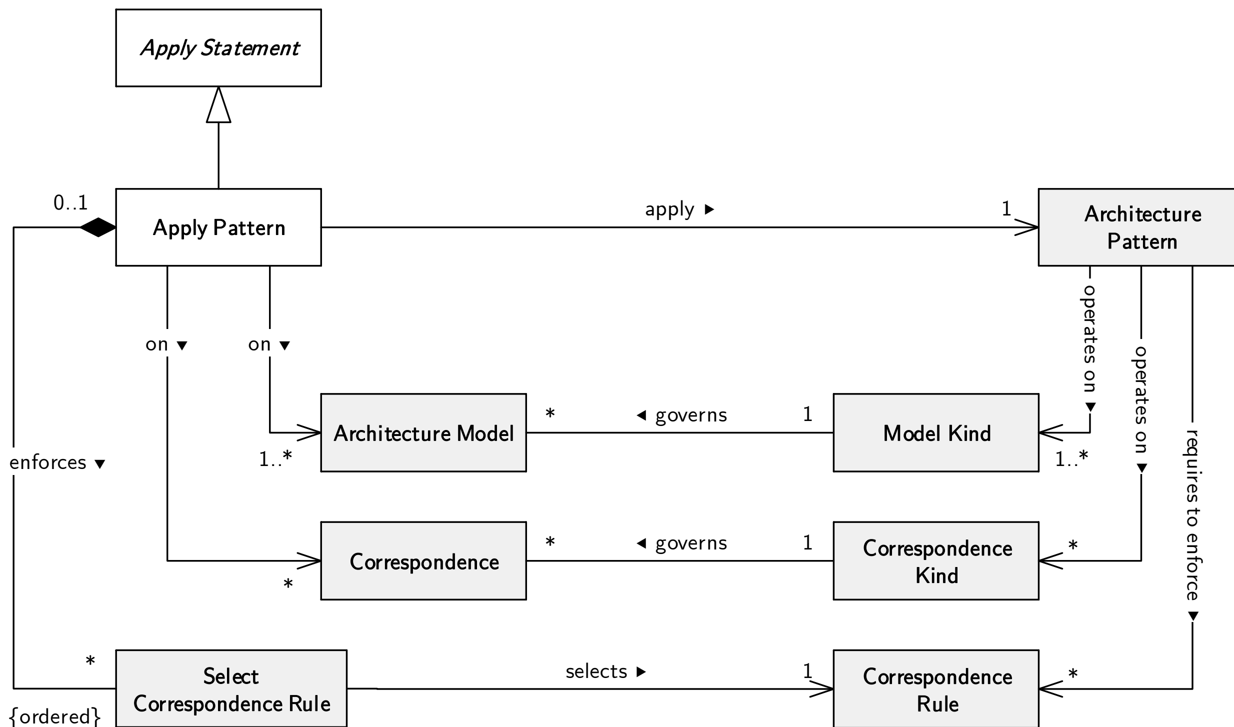


Figure 4.18: *Conceptual model of the* ApplyPattern *statement.*
The figure illustrates the specific kind of ApplyStatement for refining *architecture models* and *correspondences* using an *architecture pattern*. The statement for applying tactics is illustrated in Figure 4.19.

conform an architecture, but can also provide constructs to define new specialized types of components (e.g. Filters, Blackboard, etc.). For this kind of languages (a), to define a new style implies actually to populate an architecture model with new *component type* elements. For the kind of languages in which new instances of component types cannot be created (b), the architect needs then to create a new specialized language that introduces the new types, and to create an architecture model using this specialized language. There is no absolute reason for one being preferable to the other, which kind of language to use depends on the required expressiveness for the architecture description, the skills and experience of the team, as well as the available tool support. Our conceptualization for architecture styles fits the scenario (b) as an architecture style provides a new model kind possibly extending already existing ones. We define the ApplyStyle statement as the concrete means to create a new style in the architecture description. The architect must provide the definition of the new model kind, the existing model kind being extended, and the set of architecture viewpoints that will aggregate the new model kind. Figure 4.17 illustrates the conceptual model for this statement. Nevertheless, it is important to notice that we also support scenario (a), but by means of the ApplyPattern statement. If an architecture model can define both new types and new instances, an architecture pattern can then populate the architecture model creating any of these elements as appropriate.

An *architecture pattern* defines a generic and parameterized solution to a recurrent problem. It embodies how to update the architecture description in order to reflect and include in it the structures and behavior conforming the pattern's solution. An architecture pattern

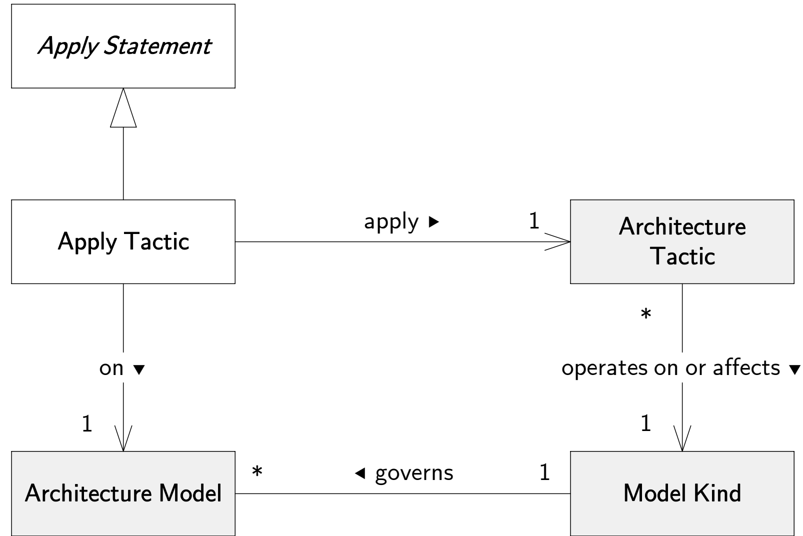


Figure 4.19: *Conceptual model of the* ApplyTactic *statement.*
The figure illustrates the specific kind of ApplyStatement for refining an *architecture model* using an *architecture tactic*. The statement for applying patterns is illustrated in Figure 4.18.

is generic and parameterized in the sense that it is not conceived for a particular *architecture model* or *correspondence*, but rather, it is conceived in terms of *model kinds* and *correspondence kinds*, so as it can then be applied on any architecture model and correspondence governed by those model and correspondence kinds. As we illustrated in Figure 4.6, an architecture pattern *operates* on model kinds and correspondence kinds. As shown in that figure, an architecture pattern also states that specific correspondence rules must be enforced on the architecture description when using the pattern. We define the ApplyPattern statement as the concrete means to apply an architecture pattern. Figure 4.18 illustrates the conceptual model for this statement. When applying a pattern, the architect must indicate the specific architecture models and correspondences to update. Also, the architect must indicate on which of these architecture models and correspondences the correspondence rules of the pattern are to be enforced. To this end, an ApplyPattern statement is composed by a possibly empty set of SelectCorrespondenceRule statement that establishes which correspondence rule applies on which architecture description element.

As we discussed in Section §4.1.1, architecture tactics are used to devise solutions packaged in architecture patterns. However, architecture tactics can be directly applied during architecture design to devise the solution to a particular quality attribute expectation of the system. In other words, architecture tactics are also building blocks for architectures [SK09]. Then, we define the ApplyTactic as the concrete means to apply an architecture tactic in the architecture description. Provided that an architecture tactic has a finer granularity than architecture patterns as they are defined to operate on a single model kind, as illustrated in Figure 4.6, the architect must indicate which is the actual architecture model on which the tactic is to be applied. Figure 4.19 illustrates the conceptual model for this statement.

The ApplyPattern and ApplyTactic statements are the only statements we provide to allow the architect to modify the architecture models and correspondences in the architecture description. Well-defined patterns and tactics provide an unambiguous specification of the

most fine-grained updates to an architecture description, particularly the definition of the structure and behavior contained in architecture models and correspondences. Architecture patterns and tactics are defined as *common* solutions to *recurrent* problems. In our conceptualization, we lighten the *recurring* characteristic of their problem. We expect the architect to capture these fine-grained modifications as if capturing patterns or tactics. By this means, instead of manual modifications, potentially reused solutions are captured and then applied to the particular case of the architecture being built. Consequently, we aim the Apply statements to be used for every fine-grained update regardless if the problem being addressed is recurrent or not.

4.1.3 Architecture Solutions, Decisions & Rationale

Architecture design is the creative and intellectual effort that the architect undertakes to decide the most suitable architecture — among the devised alternatives — that best satisfies the stakeholders' expectations on the system-of-interest. According to J. Tyree in [TA05], a complex architecture probably reflects thousands of decisions, big and small. Even though the original definitions for *software architecture* were more focused on the structural and behavioral aspects of the system, *architecture design* has always been about making the critical decisions with respect to the fundamental characterization of the system. The architecture discipline has undergone a significant paradigm-shift on this subject, in the last decade. The research and practitioner community has detected that the relevance of architecture decisions goes beyond the moment they were made. As a consequence, current techniques for architecture description emphasize the importance of capturing and documenting such decisions and the supporting rationale, in addition to elements characterizing the structure and behavior of the system.

In 1999, G. Booch et al. in [BRJ99] provided one of the first definitions of *architecture* that explicitly considers it as the set of significant *decisions* about the organization of a software system. However, it took a few years for the importance of decisions to pervade the architecture description practice. In 2004, J. Bosch envisioned intensive research focused towards the first-class representation of architecture decisions in architecture descriptions [Bos04]. Also in that year, P. Kruchten presented in [Kru04] an ontology of design decisions, their attributes and relationships, and stated that preserving the architecture decisions and all their interdependencies would support the evolution and maintenance of software systems. Since then, the community has actively developed different techniques for capturing and documenting architecture decisions. U. Heesch et al. provide in [HAH12a] a discussion on these different techniques, identifying three main categories: structured textual templates as proposed in [CBB+10, TA05], annotations to architecture models as proposed in [LJA09], and dedicated *decision models* included in architecture views as proposed in [DC05, KCD09]. Decision models can present the same information as templates and annotations, but by means of well-structured and well-organized dedicated models. U. Heesch et al. argue that these approaches do not satisfy all decision-related concerns and then, they propose in [HAH12a] a documentation framework for decisions by means of four architecture independent architecture viewpoints, which they later extend in [HAH12b] by adding a fifth architecture viewpoint. In their proposal, decision models are no longer companion models to archi-

tecture models in different architecture views, but rather, they are architecture models in purpose-specific architecture viewpoints.

In the conceptual model provided by the IEEE 1471:2000 standard [IEE00] in 2000, the concept of *rationale* was included. It was expected that an *architecture description* included an associated rationale explaining and justifying the structures and behavior captured in the description. The standard provided no further refinement or guidance for this concept. In the revised edition of this standard, even in some of the early drafts previous to the final version of the ISO/IEC/IEEE 42010:2011 standard [ISO11], the concept was defined more precisely. *Rationale* is defined as the explanation, justification or reasoning about the architecture decisions that have been made. According to the standard, the rationale includes the basis for a decision, the alternatives and trade-offs considered, potential consequences and citations to supporting material. Although a definition for the concept *decision* is not actually provided by the standard, its conceptual model determines that a decision is justified by the rationale, that it pertains to a particular set of concerns and might raise new concerns, that it affects multiple architecture description elements, and that they are interrelated according to different kinds of dependency relationships. We define an *architecture decision* as the conscious selection of one single architecture solution, among a set of potential and suitable alternatives, that provides the best long-term system-wide benefits with respect to a set of architecturally significant concerns that must be addressed. *Architecture solutions* are not explicitly considered in the standard or in decision-related publications. As we discussed before in Section §4.1.1, the community captures recurrent generic solutions by means of architecture design mechanisms such as architecture patterns, tactics and styles. However, system-specific solutions tend to be implicitly embedded in the architecture descriptions and commented or discussed in the associated rationale. The first-class representation of solutions in the current state of the practice consists of the *actual* structure and behavior conforming the architecture description. The architecture knowledge on how that structure and behavior was built, in order to make it a shareable and reusable procedure beyond manual Copy & Paste, is not explicitly captured in architecture descriptions. This information is lost during architecture design as practitioners keep (include) what architecture description elements conform the architecture description and why they do, but not how the architect has proceeded to get there. A fine-grained language for architecture description updates, as we defined in Section §4.1.2, would provide such missing first-class representation of the how.

Conceptualization

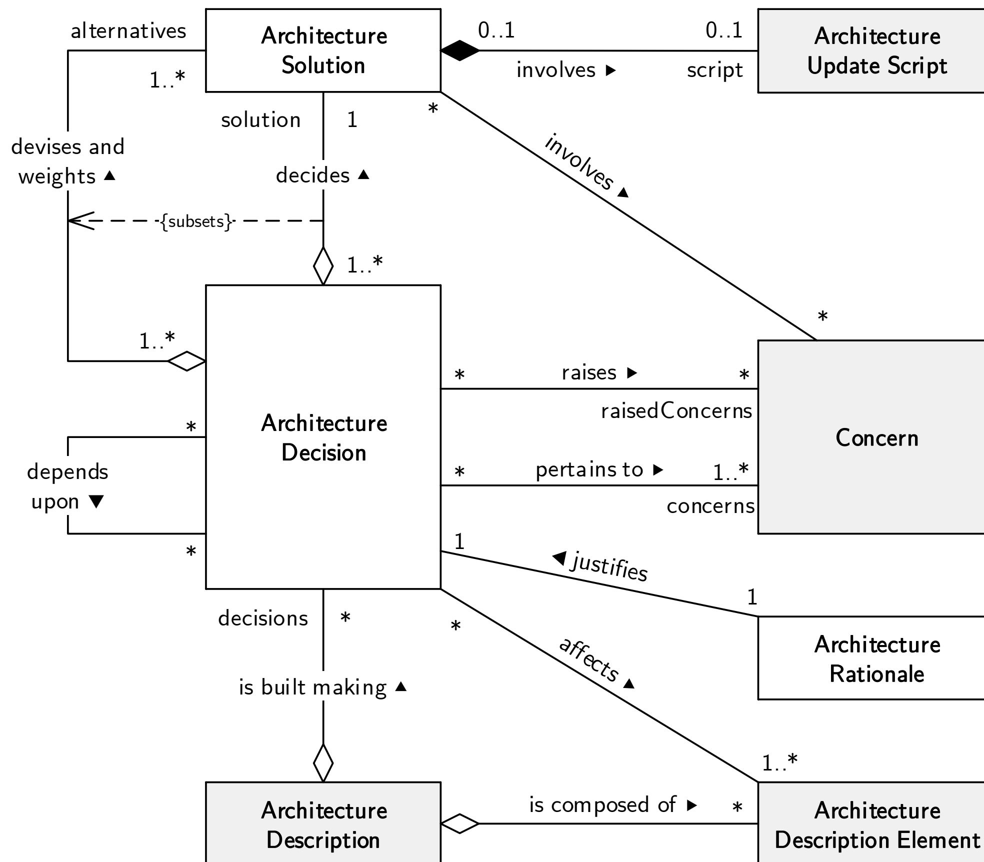
As the community understanding on the architecture description practice evolves, the conceptual model defined by the ISO/IEC/IEEE 42010:2011 standard [ISO11] is refined in order to capture the consensual agreement of the most critical concepts pertaining the practice. By recognizing the importance of architecture decisions for understanding, analyzing, maintaining and evolving an architecture description, the standard then included the concept in the conceptual model. We claim that the more this conceptual model grows in detail, the more it conflicts with the concepts that are actually captured in architecture views and architecture models. Clear examples of this conflict are the concepts *concern* and *stakeholders*. These concepts are used to characterize the other concepts in the architecture description practice, such as architecture viewpoints, architecture views, model kinds, among others.

However, one can think of a purpose-specific architecture viewpoint to capture concerns and stakeholders, possibly including external interoperable systems, business goals, business processes, among others. These latter elements are actually captured by specific architecture viewpoints in some architecture frameworks. The same issue is faced with the concept of *decision*. To require a first-class representation for them in the architecture description, does not actually imply to require a first-class representation in the conceptual model of the practice. For instance, most architectures are based in the notion of Component & Connectors, which are very important concepts in the practice, but they lack a first-class representation in the conceptual model. Then, there is a dual characterization of some concepts, either as a concept in the conceptual model, and as elements populating architecture views and models, or even both at the same time. For instance, while some authors have proposed purpose-specific architecture viewpoints and model kinds for capturing and documenting decisions, as we discussed before, other authors have propose extensions to the conceptual model of the standard. We foresee future research on addressing this duality issue.

As it is, using purpose-specific architecture viewpoints and model kinds to capture architecture decisions is already supported by the standard. It neither imposes restrictions on the purpose of architecture viewpoints, nor on the kind of constructs that populate the architecture models. Then, approaches based on architecture viewpoints, like [HAH12a], can be defined as any other architecture viewpoint would. However, approaches extending the conceptual model must take the corresponding considerations for resolving how these concepts are actually expressed in the structured organization of architecture descriptions. For instance, in Section §3.2.2 we decide to use a purpose-specific model for capturing *concerns*, that is directly aggregated by the architecture description independently of any architecture viewpoint. We proceeded in the same way in the case of *stakeholders*. In the standard, as *architecture decisions* and *rationale* are first-class concepts, practitioners must decide how the instances of these concepts are actually captured in the architecture description. The impact of this decision has an impact on architecture design. The architect must update the architecture description not only to reflect the effect of the architecture decisions made, but also to record such decisions. The representation of architecture decisions then impacts how the architect must proceed.

Our position is that architecture decisions must have a first-class representation in the conceptual model of the standard, instead of being defined as an architecture viewpoint. The standard not only states that architecture viewpoints govern the architecture views of architecture descriptions, but also that the particular architecture viewpoints depend on the practitioner community and are defined and publicly shared by means of architecture frameworks. In other words, no particular architecture viewpoint is mandatory for the standard. This is not the case for architecture decisions, as they are mandatory for the standard and as such, they are part of the conceptual model. As it is the case for *concerns*, practitioners must decide a mechanism to model those decisions, aggregating them in the architecture description. The modeling techniques from those approaches based on architecture viewpoints can still be used. However, they would be conceived as a specific mechanism to capture the first-class concept of *architecture decisions*, not just as another architecture viewpoint.

Figure 4.20 illustrates the conceptual model for ArchitectureDecision and ArchitectureRationale, inspired in the conceptual model of the standard. An *architecture decision* per-



context ArchitectureDecision

def: AllDependedUponDecisions : Set(ArchitectureDecision) =
self.dependsUponDecisions→closure(d | d.dependsUponDecisions)

inv: self.AllDependedUponDecisions→excludes(**self**)

inv: self.solution.script→notEmpty()

def: AllConcerns : Set(Concern) =
self.concerns→union(**self**.raisedConcerns)

inv: self.AllConcerns→includesAll(**self**.alternatives.concerns)

Figure 4.20: *Conceptual model of ArchitectureSolution, ArchitectureDecision and ArchitectureRationale.*

The figure illustrates the conceptual model representing the devised alternative solutions and their effect on the architecture description, the decisions made, and the rationale justifying such decisions. The relations between *decisions*, *rationale*, *concerns* and *architecture description elements* are taken from the conceptual model of the ISO/IEC/IEEE 42010:2011 standard [ISO11].

tains to a set of *concerns*, affects a set of *architecture description elements* and possibly raises new *concerns*. Also, there is a dependency relation between architecture decisions. An architecture decision is justified by its *rationale*. We define a one-to-one relationship between the decisions and rationale, as opposed to the standard that defines a many-to-many relationship. Provided the definition of *rationale* from the standard — rationale includes the basis for a decision, the alternatives and trade-offs considered, potential consequences and citations to supporting material — we expect the architect to justify every decision made. A many-to-many relationship would provide some sort of reusability of the rationale, i.e. the same justification for several decisions. However, if the rationale includes the reasoning with respect to the alternatives and the justification of why the selected one is preferable, it is unlikely that two different decisions weight the same alternatives and make the same selection.

An *architecture decision* is an aggregation of the set of alternative *architecture solutions* that the architect devises and weights in order to address a particular set of significant concerns. Additionally, an architecture decision marks one of this alternatives as the actual solution, while the rationale justifies the trade-offs and the selection made. As opposed to the standard, we provide a first-class representation for *architecture solutions*. An ArchitectureSolution is a complex architecture design mechanism that prescribes or describes the structure and behavior that must be included in the architecture description to address a particular set of concerns. An architecture solution involves an optional ArchitectureUpdateScript — that we defined before in Section §4.1.2 — that specifies how the solution can be reflected in the architecture description being built. While a solution defines what needs to be done, its benefits and liabilities, the architecture update script of the solution defines how to proceed to achieve it. We mark the solution's script as optional to allow the architect to conceive alternatives without going into the details on how to construct them. However, at least the selected solution must provide the script. We state this constraint by means of an invariant in Figure 4.20. To the best of our knowledge, there is no proposal from the research and practitioner community that provides architecture solutions and their corresponding update statements a first-class representation in architecture descriptions.

We expect architecture solutions to be shared and reused among architecture decisions, in order to support the exploration of alternatives in the solution space. Considering the example illustrated in Figure 4.4, a solution that was applied deep in one path of the exploration tree might also be applied in another path, provided that the architecture descriptions on which it is applied are compatible — i.e. the updates enforced by the solution can be performed. This allows us to provide solution reuse between paths. While each architecture decision has its rationale explaining why the selected solution was preferred, each architecture solution has its own rationale explaining why it is a solution, together with its benefits and liabilities. Then, the architecture description must preserve all the decisions made as well as all the solutions considered, as the latter are reused among decisions preferring different alternatives.

4.2 Model-Based Architecture Design

Techniques from the modeling discipline have pervaded the *architecture description practice*. First, modeling techniques are used to capture and communicate the conceptualization of the practice. As we discussed in Section §3.2, the ISO/IEC/IEEE 42010:2011 standard [ISO11] defines the consensual *domain* or *conceptual model* of the practice. We reviewed this model in Section §3.1. This model identifies the main concepts, their properties and their interrelationships, in the context architecture description practice. The importance of the conceptual model is that it provides a well-structured and well-defined vocabulary determining the kind of terms that can be used when talking about architecture descriptions. As stated in [ISO11, Section 4.1], the standard itself uses the conceptual model to express the requirements for *architecture description*, *architecture frameworks* and *architecture description languages*. Moreover, the conceptual model was a key input for us when we define the model-based interpretation of these concepts. Particularly, in Section §3.2.2 we used a slight adaptation of the conceptual model to determine the syntactic domain of our denotational semantics approach to the formalization of our model-based interpretation. Second, the standard involves the application of modeling techniques to the definition of these concepts. For the standard, an *architecture description* is an aggregation of *architecture views*, which in turn are aggregations of *architecture models*. Architecture models are the *representation of* a particular aspect of the system-of-interest, and hence, they are understood as models in the modeling sense of the term. Moreover, the standard introduces the concept of *model kind* as the reusable system-independent asset that captures the principles governing the construction of *architecture models*. As stated in the standard in [ISO11, Section 7], the language defined by *model kinds* can be captured by means of a *metamodel*. Model kinds are not only aggregated in *architecture viewpoints* within an *architecture description*, but also they are used by *architecture frameworks* and *architecture description languages* to capture the reusable architecture knowledge on the conventions on how to build *architecture models*. However, the potential of *models* is not being fully exploited in the architecture description practice yet. For instance, the standard is not committed to a model-based approach to capture architecture descriptions as it is intended to be usable for a range of approaches, including document-centric and repository-based (e.g. wikis). As we discussed in Section §3.2.4, these kinds of approaches to architecture description are a common practice. Models are only a part of an architecture description, as practitioners strongly rely on textual documents to capture and communicate most of the content, including decisions and rationale. Models themselves are frequently used to analyze a given aspect of the system, such as performance, availability or security. However, most models in architecture descriptions tend to be captured as textual descriptions and diagrams. In the current state of the practice, it is mainly the tool support for Architecture Description Languages that promotes the use of models.

To a much larger degree than *architecture description practice*, the *architecture design practice* is still underusing the potential of the modeling discipline. First, (A) there is no consensual conceptual model of the practice. In the current state of the art, various authors have dealt with different aspects of the practice, providing their own conceptual model or even metamodels when tool support was envisioned or developed. For instance, the relationship between *architecture patterns* and *architecture viewpoints* is studied in [AZ05], between *architecture patterns* and *architecture decisions* in [HAZ07], and the relation between *archi-*

tecture patterns and *architecture tactics* in [HA10]. Also, in [CZZ+11] the authors proposed a metamodel for linking *architecture decisions* to other artifacts in the development process life cycle. As we discussed before in Section §4.1, research on capturing and modeling *architecture decisions* has become more extensive recently, since the Software Architecture discipline underwent the paradigm-shift from focusing on the effect of architecture decisions in the architecture description, to focusing on capturing architecture decisions in the architecture description. Second, (B) architecture design processes are sketched, described or defined by means of textual documentation, like the Attribute-Driven Design Method [WBB+06]. Some authors also include some activity diagrams to guide the application [RW05]. However, no modeling techniques are being applied to the rigorous specification of these processes. Third, (C) architecture design is about querying the current models and processing and updating them according to the decisions made. This is mainly a manual method, generally assisted by visualization and edition tools. For the case of architecture description languages, rich tool sets are usually available to deal with concrete syntax, to check well-formedness of artifacts and models, and to analyze properties of the architecture. However, to the best of our knowledge, integrated tool support that deal with the whole representation (description) of architectures, that provides automatic traceability from concerns to decisions to architecture description elements, and that rely on an homogeneous technological support for capturing architecture knowledge on description and design, is not yet available. Custom tool environments like those supporting extensible architecture description languages, like ArchStudio [DAH+07], are the most promising approaches looking forward such a goal. However, automatic architecture description derivation from the specification of architecture decisions, and automatic traceability are not usually part of their offered functionality.

The most notorious application of modeling techniques in the context of architecture design and software development, is the case of generative technologies [TMD09], being Model Driven Architecture (MDA) initiative [OMG03] the most outstanding approach. By this means, an architect can automatically generate implementation artifacts from the architecture description. For instance, a complete definition of the structure and behavior of the system at the architecture level might render the whole implementation of the system. From a less detailed description, a skeleton system and component interfaces might be obtained. Also, in the context of a reusable component library, the implementation of component composition might be derived.

We dealt with case (A) in Section §4.1. We first extended the contextual model for the architecture description practice in order to include and relate the architecture design practice. We illustrated this contextual model in Figure 4.1. Then, based on different proposals from the literature, we defined a conceptual model for the main concepts pertaining architecture design. This conceptual model was defined as an extension of the conceptual model for the architecture description practice, that was reviewed in Section §3.1.1 and illustrated in Figure 3.2. Particularly, in Section §4.1.1 we analyzed architecture patterns, tactics and styles, and we defined the conceptual model establishing the relationship between these concepts to those of *concern*, *model kind* and *correspondence rule*. We illustrated this in Figure 4.6. In Section §4.1.3 we analyzed architecture solutions, decisions and rationale, and we illustrated the conceptual model in Figure 4.20. Case (B) refers to applying modeling techniques to the formal specification of architecture design processes. The modeling community has defined the Software & Systems Process Engineering Metamodel (SPEM) [OMG08] for the

model-based specification of software engineering processes. It provides constructs like *roles*, *artifacts*, *tasks*, *activities*, *workflows*, among others, to precisely define the method content and its life cycle. Currently, there is also tool support available, such as the Eclipse Process Framework (EPF) [Bal07]. To the best of our knowledge, there is still no application of these techniques and tools to formally capture architecture design process. This is out of the scope of our work, as we are concerned with the formal specification of the fine-grained design actions, instead of the coarse-grained activities performed by the architect. Nevertheless, we suggest this as a further line of research. In this section, our main focus is to provide support for case (C): to provide a homogeneous and applicable means for capturing the conceptualization of architecture design concepts. To this end, in Section §4.1.2 we defined the fine-grained *architecture update statements* that provide a conceptualization of the atomic updates that can be performed by the architect while enacting the Decision Making activity of an architecture design process. In what follows, we define our model-based interpretation of this concepts in terms of Model-Driven Engineering, particularly, in terms of the constructs defined by the Global Model Management approach we reviewed in Section §2.3.

In this section, we first define our model-based approach to architecture design as a paradigm-shift from focusing on the architecture description to focusing on the architecture design that lead to the description. Then, we formally define our model-based interpretation of the conceptual model for the architecture design practice, that provides the formal support for our model-based approach. To this end, we extend the denotational semantics approach that we defined in Section §3.2.1. In Section §4.2.2 we formally define the semantic functions and in Section §4.2.3 we define the semantic equations for the concepts in the conceptual model. Finally, we discuss the automation of traceability in Section §4.2.4.

4.2.1 Shifting Focus from Description to Design

In the current state of the practice, the actual *construction* of the architecture description during an architecture design effort is mainly manual, using basic tool support providing visualization and edition capabilities. Document-centric architecture descriptions are captured by one or more textual documents, including diagrams to depict the relevant architecture description elements participating in the architecture. Repository-based architecture descriptions like wikis, offer improved structure and navigation capability, but still rely on text and diagrams to capture, document and communicate the architecture being developed. The architect uses the corresponding editor tools to manually (strictly speaking assisted by tools) create and capture each *architecture model*. In repository-based architecture descriptions in a workspace of an integrated to development environment, with tool support for *architecture description languages*, the language-specific tools are used to create, update and visualize the architecture models expressed in those formal languages. However, architecture models are still developed by hand. The fact that the language provides a formal syntax and semantics — as opposed to natural language — improves the capability of the tool support by means of specialized visualizers and editors, in addition to well-formedness checkers and analysis tools. However, it is the responsibility of the architect to use these tools to appropriately develop the architecture description. Moreover, a companion textual- and diagram-based documentation is usually required and developed.

A manual tool-assisted approach to architecture description development focuses on actually building the architecture description — it is *description-centric*. The actual procedure that the architect follows to build the particular architecture description being developed, is not captured. The knowledge on how to proceed is from and for the architect alone, and it is lost during architecture design. It cannot be shared or reused, and at most it might be guessed from the architecture description as it is implicitly embedded in the structures and behaviors described. By capturing the architecture decisions made and their supporting rationale, the architect records and preserves why the architecture description is shaped in any precise way, but it does not provide any detailed information on the procedure followed to reach such a description. Capturing traceability information from decisions to the impacted architecture description elements is a step forward capturing the procedure. Using this information, any stakeholder can understand which elements were affected and possibly how they were affected, depending on the actual information captured. However, while traceability information is extremely useful for understanding, there is no straightforward approach to take this information and derive a procedure that can be reused when exploring alternative solutions or when undertaking a separate design effort. Moreover, those actual modifications that the architect can derive must be manually replicated in the new scenario. Even if the architect captures such a procedure explicitly, using an informal language or any language without the appropriate tool support will also hinder the reuse of the procedure. It requires an enormous effort to manually capture the procedure and traceability information, in addition to building the actual description. It is generally error prone as the slightest modification must be reflected in both the affected elements and the captured procedure. As a consequence, this kind of effort is rarely undertaken in practice.

As we discussed before, the software architecture community is undertaking a paradigm-shift from considering the architecture description as the set of interrelated architecture elements that characterize the system, to considering it as the set of critical decisions that lead to these architecture elements. We refer by what to the architecture elements that populate an architecture description — from the key question *what elements conform the architecture*. Then, from the original conception of capturing what, the community is shifting to capture why + what, i.e. the critical decisions and the result of their effect in terms of architecture elements. Quoting P. Kruchten in [Kru04], “for many years we have focused on the result, the consequences of the design decisions we made, trying to capture them in the ‘design’ or the ‘architecture’ of the system under consideration, using often graphics. . . . But doing so, we lose some of the knowledge that is attached to the decision itself, and to the decision process: rationale, avoided alternatives, etc.” Explicitly capturing the decisions (the why) is intended to preserve such knowledge. According to the survey on the documentation and use of architecture decisions and rationale published by A. Tang et al. in [TABGH05], although practitioners recognize the importance of documenting the decisions and rationale, most of them face barriers like the lack of standards and tool support, and mainly time and budget restrictions. A standard representation mechanism and the appropriate tool support would clearly ease and improve the adoption of capturing both why + what. However, we claim that a major productivity improvement occur when the architecture decisions not only facilitate understanding, but also when they can be encapsulated, shared and most important reused when exploring alternative solutions or when developing similar products. For us, to capture how the architect proceeds to develop the what is important.

During architecture design, the architect's effort and time is devoted to (i) analyzing requirements and the current architecture description, (ii) devising and weighting alternative solutions and deciding the most suitable solution — possibly requiring a trade-off analysis of the alternatives and renegotiation with stakeholders —, and (iii) updating the architecture description to reflect the decision made. The modifications to the architecture description involve the definition of new structural parts of the description (e.g. model kinds, architecture models), and new architectural elements populating these models, i.e. modifications affect the what. The modifications also include to record the why, i.e. the alternative solutions devised and weighted in (ii), their benefits and liabilities, and the rationale justifying the selection of a particular solution. As we discussed before, all these modifications are carried out manually using the appropriate edition tool support. The ultimate goal is to achieve a new version of the architecture description in which the architecture decision is captured, and the effect of the decision is reflected in the structural organization and the populating elements. The actual procedure followed by the architect to perform such modifications is not captured and consequently, it is lost. We call this knowledge the how. By capturing the how, the stepwise procedure is explicitly recorded, it can be analyzed, modified at the step level, and also important it can be shared and reused. Then, as a first approximation to our approach, we propose to capture why + how + what.

Capturing architecture decisions and rationale requires a great effort, and even more if the architect intends to be thorough in capturing traceability information. Capturing the how also requires a great effort. It is important to notice, however, that the how and the what are actually two perspectives of the same activity. The how is the steps to be performed (the means), and the what is the effect of these steps (the end). Capturing both not only requires an even greater effort, but also introduces redundancy and is error prone. Then, we propose to prioritize capturing the how to capturing the what. One might argue that by focusing on the how, the architect has no actual representation of the architecture to directly work with, just the set of architecture decisions and the corresponding required updates. This would be the case if the generation of the representation is still manual. Informally capturing the how produces no actual benefit in this sense. However, capturing it formally, i.e. in a way that it can be machine-processed, it allows the architect to automatically produce or generate the what from the specified how, even to dynamically see the effect of any change in the how onto the what. Then, formally capturing the how, together with the appropriate tool support, renders unnecessary to capture the what. We are not claiming that the what is not important or unnecessary. The architect devotes a considerable amount of time analyzing the what to decide the next steps to follow. We are claiming that formally capturing the how allows the automatic generation of the what, without any additional effort. We might argue that this assertion is also valid in the opposite direction. In the general case, the complexity of this approach would be similar to that of program derivation. In the particular case in which an initial and final states, i.e. the architecture descriptions before and after the modifications, the set of changes can be extracted in terms of create-update-delete actions. These extracted changes are specific to the particular refinement step. It provides no generalization or parameterization (like the application of architecture patterns) and no appropriate modularization of the changes. As a consequence, the reusable capability for these changes would be extremely limited. Then, we define our model-based approach to architecture design as the process that captures the why and the how in a way that the what can be automatically derived. In short: why + how ⇒ what.

It is important to remark that this approach is not new in the Software Engineering discipline. For instance, practitioners implement test cases in a general-purpose programming language to "program" how to automatically perform unit tests, they use scripts (e.g. Ant) to program how to build large implementations, they use database scripts to program how to create a given schema in a database management system, they use deployment scripts to program how to deploy, install and configure a system in a production environment, etc. What is more, we have used LaTeX to "program" how to build this thesis report. We claim that a new shift in the focus from capturing the why+what to capture the why+how is feasible in the current state of the art, and it is beneficial to the architecture design practice. While the productivity boost might not be significant in a forward architecture design of a system (i.e. the architecture design of a single system from scratch), it is significant in the context of planned reuse when a line or family of products is developed at the same time. We study this scenario of application of our architecture design approach in Chapter §5.

We use Model-Driven Engineering techniques to provide both the foundation and the instrumentation of our architecture design approach why + how ⇒ what. In Chapter §3 we defined a model-based representation for every kind of construct pertaining the architecture description practice. In this chapter, we extend such a representation to cover also our conceptualization of the architecture design practice. In Section §3.1 we reviewed the conceptualization of the *architecture description practice* defined by the ISO/IEC/IEEE 42010:2011 standard. In Section §4.1, we extended that conceptual model to include the main concepts in the *architecture design practice*. This conceptualization provides us with a characterization of the domain to be formalized. In Section §3.2 we formally defined a model-based interpretation of the constructs pertaining the architecture description practice. To this end, we applied the denotational semantics approach to provide a formal denotation of each construct in terms of Model-Driven Engineering techniques. In Section §3.2.1 we defined the semantic functions and in Section §3.2.2 we defined the semantic equations of these functions for the constructs related to the concept of *architecture description*. The semantic equations for *architecture frameworks* and *architecture description languages* were defined in Section §3.2.3. Later, in Section §4.2.2, we extend the semantic functions, and in Section §4.2.3 we define the semantic equations, in order to cover the architecture design constructs. This formalization addresses both structural and behavioral aspects of the constructs. By this means, we are defining a fine-grained architecture design language providing its syntactical constructs (§4.1) and its formal semantics in terms of modeling artifacts (§4.2.3). Consequently, we provide an homogeneous approach to capture architecture knowledge on architecture description and architecture design that is shareable and reusable. Modeling artifacts provide such homogeneous means. We first published our model-based approach in [PBR09]. In this thesis report, we complement that work and we provide the supporting theoretical foundation.

4.2.2 Definition of the semantic functions

In Section §3.2 we formally defined our model-based interpretation of *architecture descriptions*. We applied the denotational semantics approach [NN92, Sch86] to specify the meaning of concepts and concept instances in the conceptual model defined by the ISO/IEC/IEEE 42010:2011 standard [ISO11] for the *architecture description practice*. Our specification ex-

presses such meaning in terms of the modeling constructs available in the current practice of the Model-Driven Engineering discipline. In this section, we continue our specification to cover the formal denotation of the meaning of concepts and concept instances in the conceptual model for *architecture design practice* that we defined before in Section §4.1. As opposed to architecture description, we are not only interested in the representation of those concepts and their instances in terms of the modeling constructs, but also, we are interested in their behavior. Architecture design, and particularly the Decision Making activity, is about both deciding how the architecture is shaped, and reflecting those decisions in the architecture description. Thus, most of the concepts in our conceptual model have a dual interpretation: they are *prescriptive or descriptive* as they state the modifications that will be or were performed by the architect, and they are *applicable or behavioral* as they work as the actual tool that *is being* applied by the architect to exercise the corresponding modifications. This dual characterization is reified in the two kinds of semantic functions that we define later.

As we explained in Section §3.2.1, the application of the denotational semantics approach consists of the definition of (i) the syntactic domain being formalized, (ii) the semantic domain being targeted by the formalization, (iii) the semantic functions, and (iv) the semantic equations for those functions. We provide these definitions in what follows.

(i) Syntactic domain

In the context of the *architecture description practice*, we defined the syntactic domain as the conceptual model specified in the ISO/IEC/IEEE 42010:2011 standard that we reviewed in Section §3.1. That conceptual model provided us the constructs for representing architecture knowledge in the practice. As summarized in Table 3.1 on page 88, we defined $\mathcal{A}^{\mathsf{C}}$ as the set of concepts in the conceptual model, $\mathcal{A}_{\mathsf{c}}$ as the set of instances of those concepts, $\mathcal{A}$ as their union, and $\mathcal{A}^{*}$ as the power set of $\mathcal{A}$.

In Section §4.1, we defined our conceptual model for the *architecture design practice*, that includes the common design mechanisms, the decisions and rationale, and the set of fine-grained update statements that conform architecture solutions. As before, we use the conceptual model as the foundation for the syntactic domain. We extend the definitions for architecture description practice defined Section §3.2.1 to also consider the conceptual model for architecture design practice. Thus, we define $\mathcal{A}^{\mathsf{C}}$ as the set of concepts both of architecture description and architecture design, and $\mathcal{A}_{\mathsf{c}}$ as the set of their instances. As extending the syntactic domain requires to extend the definition of the semantic functions, we define the semantic equations for the additional concepts and concept instances later in Section §4.2.3.

In the context of architecture design, it is important to distinguish a specific subset of concepts that, in addition to provide a descriptive aspect, they also provide a behavioral aspect. Instances of concepts in this particular subset, when applied on an architecture description, produce an effect yielding an updated architecture description. Particularly, every concept pertaining the architecture update statements are actually behavioral concepts. Notice for instance that while an ArchitecturePattern is only descriptive, applying a pattern (ApplyPattern statement) is both descriptive and behavioral. In addition, we consider solu-

$\mathcal{S}^{\mathsf{C}}$	$\overset{\text{def}}{=} \{x \mid x$ is a concept representing an architecture update statement$\}$
	$\equiv \{x \mid x$ is ArchitectureUpdateStatement or one of its sub-metaclasses$\}$
$\mathcal{U}^{\mathsf{C}}$	$\overset{\text{def}}{=} \{x \mid x$ is a behavioral concept from the conceptual model for architecture design$\}$
	$\equiv \mathcal{S}^{\mathsf{C}} \cup \{\mathsf{ArchitectureDecision}, \mathsf{ArchitectureSolution}, \mathsf{ArchitectureUpdateScript}\}$
$\mathsf{X} : \square$	$\overset{\text{def}}{=} \mathsf{X} \in \mathcal{U}^{\mathsf{C}}$
$\mathcal{U}_{\mathsf{c}}$	$\overset{\text{def}}{=} \{x \mid x$ is an instance of a concept in $\mathcal{U}^{\mathsf{C}}\}$
$\mathsf{x} :_{\square} \mathsf{X}$	$\overset{\text{def}}{=} \mathsf{X} : \square \;\wedge\; \mathsf{x} \in \mathcal{U}_{\mathsf{c}} \;\wedge\;$ x is an instance of the concept X

Table 4.2: *Syntactic domain of the semantic functions.*

tions to be behavioral, as an ArchitectureSolution might be performed or carried out in an architecture description to update it according to its update script. Also, an ArchitectureDecision as its selected solution is performed when the decision is made. It is important to notice that their descriptive aspect is still present. A concept presenting both aspects can be understood as the representation of an action that can be performed or was performed (prescriptive or descriptive) and as the representation of an action that is being performed (behavioral).

We define $\mathcal{S}^{\mathsf{C}}$ as the set containing the concept ArchitectureUpdateStatement and all its sub-metaclasses, defined in Section §4.1 and illustrated in Figures 4.8 to 4.19. We define $\mathcal{U}^{\mathsf{C}}$ as the set of all the behavioral concepts in the conceptual model we defined in Section §4.1 for the architecture design practice. $\mathcal{U}^{\mathsf{C}}$ is the finite set defined by extension that includes all concepts in $\mathcal{S}^{\mathsf{C}}$, as well as concepts ArchitectureUpdateScript, ArchitectureSolution and ArchitectureDecision. Notice that ArchitectureRationale is not behavioral. We use $\mathsf{X} : \square$ to denote that $\mathsf{X} \in \mathcal{U}^{\mathsf{C}}$. We define $\mathcal{U}_{\mathsf{c}}$ as the set of all elements or instances of concepts in $\mathcal{U}^{\mathsf{C}}$. $\mathcal{U}_{\mathsf{c}}$ is an infinite set. Given $\mathsf{X} : \square$, we use $\mathsf{x} :_{\square} \mathsf{X}$ to denote both that $\mathsf{x} \in \mathcal{U}_{\mathsf{c}}$ and that it is an instance of the concept X. It is important to notice that, by definition, $\mathcal{U}^{\mathsf{C}} \subset \mathcal{A}^{\mathsf{C}}$ and $\mathcal{U}_{\mathsf{c}} \subset \mathcal{A}_{\mathsf{c}}$. Then, we have that $\mathsf{X} : \square \Rightarrow \mathsf{X} : \circ$ and $\mathsf{x} :_{\square} \mathsf{X} \Rightarrow \mathsf{x} :_{\circ} \mathsf{X}$. Table 4.2 provides a condensed summary of the definition of the syntactic domain.

(ii) Semantic domain

The semantic domain for our model-based interpretation of *architecture design* is the same as the semantic domain we defined in Section §3.2.1 (ii) for the interpretation of *architecture descriptions*. It is conformed by the universe of *model repositories* storing *modeling artifacts*, that are characterized by means of the technology-independent metamodel provided by the Global Model Management approach that we reviewed in Section §2.3. Table 4.3 replicates the definitions from Chapter §3 for ease of reading. It is important to recall that the notation is based on set theory and that we use assertions to state facts on the semantic elements in order to further characterize the denotation.

$$
\begin{aligned}
\mathcal{R} &\stackrel{\text{def}}{=} \{x \mid x \text{ is a modeling artifact}\} \\[1ex]
\mathcal{R}^* &\stackrel{\text{def}}{=} \{x \mid x \text{ is a finite set of modeling artifacts}\} \\
&\equiv \{x \mid x \text{ is a model repository}\} \\
&\equiv \mathcal{P}(\mathcal{R}) \\[1ex]
\mathsf{R} : \blacksquare &\stackrel{\text{def}}{=} \mathsf{R} \text{ is a model repository} \\
&\equiv \mathsf{R} \text{ is a finite set of modeling artifacts} \\
&\equiv \mathsf{R} \subset \mathcal{R} \quad \wedge \quad \mathsf{R} \text{ is finite} \\
&\equiv \mathsf{R} \in \mathcal{R}^* \quad \wedge \quad \mathsf{R} \text{ is finite} \\[1ex]
\mathcal{R}^{\mathsf{E}} &\stackrel{\text{def}}{=} \{x \mid x \text{ is a kind of modeling artifact}\} \\
&\equiv \{x \mid x \text{ is a metaclass in the metaclass hierarchy of } \mathsf{Entity}\} \\
&\equiv \{\mathsf{Entity}, \mathsf{Model}, \mathsf{ExternalEntity}, \\
&\qquad \mathsf{ReferenceModel}, \mathsf{Metametamodel}, \mathsf{Metamodel}, \\
&\qquad \mathsf{TerminalModel}, \mathsf{TransformationModel}, \mathsf{WeavingModel}, \mathsf{Megamodel}\} \\[1ex]
\mathcal{R}^{\mathsf{C}} &\stackrel{\text{def}}{=} \{x \mid x \text{ is a concrete kind of modeling artifact}\} \\
&\equiv \{x \mid x \text{ is a concrete sub-metaclass of } \mathsf{Entity}\} \\
&\equiv \{\mathsf{ExternalEntity}, \mathsf{Metametamodel}, \mathsf{Metamodel}, \\
&\qquad \mathsf{TerminalModel}, \mathsf{TransformationModel}, \mathsf{WeavingModel}, \mathsf{Megamodel}\} \\[1ex]
\mathsf{X} : \bullet &\stackrel{\text{def}}{=} \mathsf{X} \in \mathcal{R}^{\mathsf{E}} \\[1ex]
\mathsf{x} :_{\bullet} \mathsf{X} &\stackrel{\text{def}}{=} \mathsf{x} \in \mathcal{R} \quad \wedge \quad \mathsf{X} : \bullet \quad \wedge \quad \mathsf{x} \text{ is a modeling artifact of kind } \mathsf{X}
\end{aligned}
$$

Table 4.3: *Semantic domain of the semantic functions.*
This table presents the definition of *modeling artifacts* and *model repositories* that conforms the semantic domain of the semantic functions. This table is a replica of Table 3.2 on page 90 that is provided here for ease of reading.

Semantic operators. The semantic domain consists of both the universe of semantic elements and the set of operators that can be applied on these elements. Provided that our semantic domain consists of modeling artifacts and model repositories, the operators determine how these elements can be queried and manipulated. These operators must be provided by the supporting modeling environment and tool being used.

As we explained in Section §3.2.1, *modeling artifacts* are resources that can be uniquely identified. Given any two artifacts a_1 an a_2, it is possible to determine whether $\mathsf{a}_1 = \mathsf{a}_2$ or $\mathsf{a}_1 \neq \mathsf{a}_2$ holds. Additionally, we defined several assertions that characterize the modeling artifacts in terms of Global Model Management constructs. While we use these assertions to determine the properties and characteristics of the modeling artifacts, they can also be considered operators as it must be possible to check whether they hold or not for any particular case. We also defined a *model repository* as a set of modeling artifacts. Consequently, as we explained in Section §3.2.1 and used in Section §3.2.2, set relations such as member inclusion and subset, and set operators such as union, intersection and difference, are also available. For example, given two model repositories $\mathsf{R} = \{\mathsf{a}_1, \mathsf{a}_2\}$ and $\mathsf{R}' = \{\mathsf{a}_1\}$, we have that the

assertion $\mathsf{R} \bigcup \mathsf{R}' = \mathsf{R}$ holds as R' contains only the modeling artifact a_1 which is already stored in R. From a practical point of view, an scenario of the union of repositories takes place when importing a set of modeling artifacts defined elsewhere. Let R be the model repository in the working space and let $\mathsf{R}_{\mathsf{imp}}$ be the model repository which artifacts are to be imported. Then, to *import* means to create a new version R' of the model repository R of the working space, being R' the union of all artifacts already in R and the artifacts in $\mathsf{R}_{\mathsf{imp}}$. In other words, the resulting model repository of the working space contains all current and imported artifacts, and it has no duplication of modeling artifacts given that modeling artifacts have identity and the union uses identity to determine the resulting set.

Transformation engine. In addition to the basic operations for validating assertions and for manipulating sets of modeling artifacts, the Model-Driven Engineering discipline relies on a remarkable operator providing support for the execution of transformations on modeling artifacts stored in a model repository, namely the transformation engine.

As we reviewed in Section §2.2 and specified in Section §2.3 in the context of GMM, a *transformation* specifies the modeler's knowledge on how to manipulate the elements within modeling artifacts for a given purpose. A transformation is reified by a *transformation model*, the special kind of *terminal model* that implements the intended purpose of the transformation. A transformation model is expressed in terms of a *transformation modeling language* defined by the metamodel to which the transformation model conforms to. It captures how to query the input artifacts, how to update the input-output ones, and how to generate the new output artifacts. Each transformation and its implementing transformation model are not defined in terms of the actual artifacts they operate on. Rather, they are defined in terms of *kind* of artifacts, namely, in terms of *reference models* or *external entity kinds* that we reviewed in Section §2.3.1. Thus, a transformation is parametric on the actual artifacts it manipulates and produces. It is a *transformation record* that captures the actual execution of a transformation on specific modeling artifacts. A *transformation engine* is the tool support provided by the modeling environment for executing transformations. Given a *transformation modeling language*, the transformation engine for that language is responsible for executing transformation models expressed in that language, consuming specific input modeling artifacts available in the model repository of the working space and producing output artifacts.

We use the symbol $\mathcal{T}$ to represent transformation engines. Then, given a transformation modeling language $\mathsf{tmm} :_\bullet \mathsf{Metamodel}$, $\mathcal{T}_{\mathsf{tmm}}$ represents the specific transformation engine that is capable of executing transformation models conforming to tmm. For instance, letting $\mathsf{atlMM} :_\bullet \mathsf{Metamodel}$ be the metamodel for the ATL [JK05] model transformation language, then $\mathcal{T}_{\mathsf{atlMM}}$ represents the model transformation engine for ATL capable of executing ATL model transformations. A transformation engine takes one single transformation model and provides a transformation operator that takes modeling artifacts from a model repository and produces a set of resulting modeling artifacts. Formally, the signature of a transformation engine is $\mathcal{T}_{\mathsf{tmm}} : \mathcal{R} \to (\mathcal{R}^* \hookrightarrow \mathcal{R}^*)$. Thus, given a particular transformation model $\mathsf{tm} :_\bullet \mathsf{TransformationModel}$ conforming to tmm, $\mathcal{T}_{\mathsf{tmm}}(\mathsf{tm}) : \mathcal{R}^{m+n} \hookrightarrow \mathcal{R}^{m+q}$ represents the transformation operator for the transformation model tm that takes a tuple of m InOut and n In modeling artifacts, and executes the transformation model tm updating the m InOut and

producing the q Out modeling artifacts. Table 4.4 provides the formal declaration for transformation engines and operators available in the semantic domain.

In the semantic equations of our formalization, we apply transformation operators to create new artifacts or new versions of existing artifacts, by querying existing artifacts in a model repository. For example, let $\mathsf{t}\,\langle\overline{\mathsf{x}_1},\ldots,\overline{\mathsf{x}_m},\mathsf{y}_1,\ldots,\mathsf{y}_n \twoheadrightarrow \mathsf{z}_1,\ldots,\mathsf{z}_q\rangle$ be a transformation where x_i, y_j and z_k are defined in a model repository R and determine the kind of entities of the InOut, In and Out parameters respectively – i.e. x_i, y_j and z_k are either *reference models* or *external entity kinds*. Let $\mathsf{tm} :_\bullet \mathsf{TransformationModel}$ be a transformation model in R that realizes the transformation t. Then, given m a_i entities in R such that $\forall \mathrm{i}, 0 < \mathrm{i} \leq m.\ \ \mathsf{a}_\mathrm{i} \lessdot \mathsf{x}_\mathrm{i}$, and n b_j entities in R such that $\forall j, 0 < j \leq n.\ \ \mathsf{b}_\mathrm{i} \lessdot \mathsf{y}_\mathrm{i}$, the application of the transformation operator $\mathcal{T}_{\mathsf{tmm}}(\mathsf{tm})$ on $\langle \mathsf{a}_1,\ldots,\mathsf{a}_m,\mathsf{b}_1,\ldots,\mathsf{b}_n\rangle$ produces the set of entities $\{\mathsf{a}'_1,\ldots,\mathsf{a}'_m,\mathsf{c}_1,\ldots,\mathsf{c}_q\}$ where each $\mathsf{a}'_\mathrm{i} \lessdot \mathsf{x}_\mathrm{i}$ is the result of updating a_i according to tm, and each $\mathsf{c}_k \lessdot \mathsf{z}_k$ is the output of tm. In other words, the result of the transformation operator is the set of artifacts containing the updated version a'_i of each InOut argument a_i, and each generated Out artifact c_k. Given the initial model repository R in the working space of the modeling environment, we expect the resulting artifacts of executing a transformation to be part of the model repository in the working space. While this is directly provided and correctly handled by the modeling environment and supporting tool set, we need to explicitly represent this capability at the formalization level. In other words, given the initial model repository R, we need the ability to state how the new version R' of R is built from the modeling artifacts in R and from the resulting artifacts of executing transformations. As we discussed above, the *union* operator is used to represent the model repository that contains artifacts from two model repositories. Then, we can use this operator to include in R' all artifacts from R and the new artifacts c_k produced by the transformation. However, the union operator cannot be applied to include those artifacts that were updated, i.e. the a_i InOut arguments. We introduce a substitution operator on model repositories to explicitly represent the capability of the modeling environment to replace the content of an existing modeling artifact in the model repository. Given a model repository R and two versions of the same modeling artifact a and a', $\mathsf{R}[\mathsf{a}/\mathsf{a}']$ represents the model repository in which every $\mathsf{e} \neq \mathsf{a}$ in R is also in the result of the substitution, and in which the content of a in the substitution is replaced by the content of a'. We use $\mathsf{R}[\mathsf{a}_1/\mathsf{a}'_1,\ldots,\mathsf{a}_p/\mathsf{a}'_p]$ for the simultaneous substitution of the content of multiple artifacts $\mathsf{a}_1,\ldots,\mathsf{a}_p$. In practice, the tool support achieves this by replacing the actual resource preserving the modeling artifact but keeping the same resource identifier. In this way, not only the identity of modeling artifacts is preserved, but also the links used in megamodels targeting those modeling artifacts are kept valid.

It is important to remark that it is the responsibility of the supporting modeling tool to enforce the semantics of the operators. Our specification assumes that these operators are available in the semantic domain and that they correctly behave as expected. The semantic domain is a foundational basis for our semantics specification, but its formalization is not the goal of our work.

$\mathcal{T}_{\mathsf{tmm}} : \mathcal{R} \to (\mathcal{R}^* \hookrightarrow \mathcal{R}^*)$ with $\mathsf{tmm} :_\bullet \mathsf{Metamodel}$

is the transformation engine for the transformation language defined by tmm

$\mathcal{T}_{\mathsf{tmm}}(\mathsf{tm}) : \mathcal{R}^{m+n} \hookrightarrow \mathcal{R}^{m+q}$

with $m, n, q \in \mathbb{N}, m + n + q > 0 \ \wedge$

$\mathsf{tm} :_\bullet \mathsf{TransformationModel} \ \wedge \ \mathsf{tmm} :_\bullet \mathsf{Metamodel} \ \wedge \ \mathsf{tm} \triangleleft \mathsf{tmm} \ \wedge$

$\mathsf{tm} \blacktriangleright \mathsf{t} \langle \overline{\mathsf{x}_1}, \ldots, \overline{\mathsf{x}_m}, \mathsf{y}_1, \ldots, \mathsf{y}_n \twoheadrightarrow \mathsf{z}_1, \ldots, \mathsf{z}_q \rangle$

is the transformation operator for the transformation model tm

$\mathcal{T}_{\mathsf{tmm}}(\mathsf{tm}, \langle \mathsf{a}_1, \ldots, \mathsf{a}_m, \mathsf{b}_1, \ldots, \mathsf{b}_n \rangle)$

with $m, n, q \in \mathbb{N}, m + n + q > 0 \ \wedge$

$\mathsf{tm} :_\bullet \mathsf{TransformationModel} \ \wedge \ \mathsf{tmm} :_\bullet \mathsf{Metamodel} \ \wedge \ \mathsf{tm} \triangleleft \mathsf{tmm} \ \wedge$

$\mathsf{tm} \blacktriangleright \mathsf{t} \langle \overline{\mathsf{x}_1}, \ldots, \overline{\mathsf{x}_m}, \mathsf{y}_1, \ldots, \mathsf{y}_n \twoheadrightarrow \mathsf{z}_1, \ldots, \mathsf{z}_q \rangle$

$\forall \mathrm{i}, 0 < \mathrm{i} \leq m. \ \mathsf{a}_\mathrm{i} \lessdot \mathsf{x}_\mathrm{i} \ \wedge$

$\forall j, 0 < j \leq n. \ \mathsf{b}_j \lessdot \mathsf{y}_j$

$\stackrel{\text{def}}{=} \{\mathsf{a}'_1, \ldots, \mathsf{a}'_m, \mathsf{c}_1, \ldots, \mathsf{c}_q\}$

where ($\forall \mathrm{i}, 0 < \mathrm{i} \leq m. \ \mathsf{a}'_\mathrm{i} \lessdot \mathsf{x}_\mathrm{i} \ \wedge \ \mathsf{a}'_\mathrm{i}$ is the result of updating a_i according to tm) $\wedge$

($\forall k, 0 < k \leq q. \ \mathsf{c}_k \lessdot \mathsf{z}_k \ \wedge \ \mathsf{c}_k$ is the output of tm)

$\mathsf{R}' \equiv \mathsf{R}[\mathsf{a}/\mathsf{a}']$ is such that

$\forall \mathsf{e} \in \mathsf{R}. \ \mathsf{e} \neq \mathsf{a} \Rightarrow \mathsf{e} \in \mathsf{R}' \ \wedge$

$\mathsf{a} \in \mathsf{R}'$ but with its content replaced by the content of a'

Table 4.4: *Semantic operators $\mathcal{T}_{\mathsf{tmm}}$ for transformation engines.*

(iii) Semantic function

In the application of the denotational semantics approach, semantic functions are used to map elements in the syntactic domain to elements in the semantic domain. In Section §3.2.1 we defined the semantic functions $\mathcal{M}$ and $\mathcal{M}^*$ to specify the model-based interpretation of the concepts and concept instances related to the practice of *architecture description*. We now define the corresponding semantic functions to formalize the practice of *architecture design*.

For any given syntactic element, its denotation is expressed by (a) precisely indicating specific elements in the semantic domain, or (b) by the application of the available operators on elements in the semantic domain. We already used both cases when we defined the semantic equations for the semantic functions $\mathcal{M}$ and $\mathcal{M}^*$ in Sections §3.2.2 and §3.2.3. For instance, case (a) was used to define the denotation of the Concern concept as the set of specific well-identified models, as defined in Table 3.8. Case (b), usually in combination with case (a), was used in the semantic equations for most of the other concepts and concept instances. For instance, the denotation of the Stakeholder concept is defined as the union of the denotation of Concern with a set of specific well-identified models, as defined in Table 3.9. Then, it is important to notice that operators in the semantic domain are useful even in the

case were a structural mapping is being defined. The semantic functions $\mathcal{M}$ and $\mathcal{M}^*$ are structural in the sense that they do not capture any behavior in the *syntactic* domain, they only map syntactic elements to semantic elements.

In the context of *architecture design*, however, we face a different scenario. As we remarked before, we have a dual interpretation of most of the syntactic constructs pertaining the conceptual model of the architecture design practice. On the one hand, design concepts are *prescriptive* and *descriptive*. When the architect defines one or more alternative solutions, along with the corresponding update scripts to some or all of them, and defines the decision made, these elements are prescriptive as they state the architect's intention that must be reflected in the architecture description. Afterwards, once the required modifications were made, these elements are descriptive as they state the decision made by the architect, the solutions weighted and the one selected, along with the corresponding update scripts that capture how to proceed to modify the architecture description. On the other hand, design concepts are also *behavioral*. The conceptual model we defined include the set of fine-grained modifications that can be performed on an architecture description during the Decision Making activity of architecture design. These modifications are captured by means of update scripts conformed by update statements, and determine how the architecture description is to be updated. The goal of these fine-grained modifications goes beyond prescription and description, we use them to perform the actual modifications. They capture the fine-grained behavior that is to be applied by the architect. Then, our semantics specification must deal with both cases: to provide a representation of the constructs and to perform (execute) them.

The semantic functions $\mathcal{M}$ and $\mathcal{M}^*$ defined in Section §3.2.1 serve the purpose of providing the prescriptive and descriptive representation of syntactic elements in terms of semantic elements, but restricted to the context of the architecture description practice. We extend the definition of these semantic functions to cover also the concepts in the conceptual model for architecture design. To this end, in Section §4.2.2 we extended the syntactic domain $\mathcal{A}$ previously defined in Section §3.2.1. Then, the signature of the semantic functions remains unchanged, they still maps elements from $\mathcal{A}$. The solely difference is that now we consider $\mathcal{A}$ to have a broader scope. The semantic equations already defined for $\mathcal{M}$ and $\mathcal{M}^*$ in Sections §3.2.2 and §3.2.3 are still valid. What remains to be done is to complete the definition by providing the semantic equations for the new kind of elements that we now include in $\mathcal{A}$. These semantic equations are defined later in Section §4.2.3.

In order to formally specify the semantics for the *behavioral* aspects of the constructs, we define the semantic function $\mathcal{D} : \mathcal{U}_{\mathsf{c}} \longmapsto (\mathcal{R}^* \hookrightarrow \mathcal{R}^*)$ that maps concept instances in $\mathcal{U}_{\mathsf{c}}$ to their effect on model repositories. The syntactic domain of $\mathcal{D}$ is the set $\mathcal{U}_{\mathsf{c}}$ which consists of all conceivable concept instances of the behavioral concepts defined in $\mathcal{U}^{\mathsf{C}}$. The codomain of $\mathcal{D}$ is the set of partial functions from model repositories to model repositories. Each concept instance $\mathsf{u} :_{\square} \mathsf{X}$ is mapped to a partial function $\mathcal{D}[\![\mathsf{u}]\!] : \mathcal{R}^* \hookrightarrow \mathcal{R}^*$ that states the effect of instance u on a given model repository, in terms of the application of the semantic operators available in the semantic domain. Given a model repository $\mathsf{R_i} : \blacksquare$, the application of the partial function $\mathcal{D}[\![\mathsf{u}]\!](\mathsf{R_i})$ generates $\mathsf{R_{i+1}} : \blacksquare$ that contains the same and possibly more modeling artifacts than $\mathsf{R_i}$ and where some of the modeling artifacts were modified. Additionally, the resulting model repository also captures the representation of the specific concept instance being used to produce the model repository. We achieve this by applying

the semantic function $\mathcal{M}$. In other words, the partial functions determines how to produce the next version R_{i+1} of a model repository R_i according to the intention of the concept instance being denoted. The denotation targets a *partial* function as we cannot every update to be possible in any model repository. Invalid scenarios include starting an already started architecture design effort, applying a pattern that operates on a model kind that is not being used in the architecture description being built, among others. The semantic equations for $\mathcal{D}$ determine which are the conditions that must hold on the current model repository for the concept instance to be applied.

In order to improve readability, when necessary, in the application of the semantic function $\mathcal{D}$ to a syntactic element u we augment the argument to the assertion that states the concept characterizing the instance. Then, we use $\mathcal{D}[\![\mathsf{u} :_{\square} \mathsf{X}]\!]$ to denote $\mathcal{D}[\![\mathsf{u}]\!]$ and to make explicit that $\mathsf{u} :_{\square} \mathsf{X}$. In order to avoid cluttering in the definitions, we introduce a shorthand notation for the application of the semantic function and the resulting partial functions. We omit the function name and we use juxtaposition for the argument corresponding to the model repository. Then, given $\mathsf{u} :_{\square} \mathsf{X}$ and $\mathsf{R} : \blacksquare$, $\mathcal{D}[\![\mathsf{u}]\!](\mathsf{R})$ is expressed by $[\![\mathsf{u} :_{\square} \mathsf{X}]\!]\mathsf{R}$ as a shorthand, and even $[\![\mathsf{u}]\!]_{\square}\mathsf{R}$ when understanding is not risked.

(iv) Semantic equations

In the case of the semantic function $\mathcal{M}$, we use the same general form for the definition of the semantic equations that we defined in Section §3.2.1. For each concept, two semantics equations are defined, one for the case of the semantic function $\mathcal{M}^{\mathsf{C}}$ on concepts, and one for the semantic function $\mathcal{M}_{\mathsf{c}}$ on concept instances, both of them using the general form:

$$[\![x]\!] \stackrel{\text{def}}{=} \mathsf{R} \text{ such that } \psi$$

where x is a concept or concept instance, and its denotational semantics is a set of modeling artifacts R on which the assertion ψ holds. The assertion ψ characterizes the resulting modeling artifacts, and states their properties and relationships.

In the case of the semantic function $\mathcal{D}$, however, we need a different general form for the definition of its semantic equations:

$$\begin{aligned}[\![\mathsf{u}]\!]_{\square}\mathsf{R} \stackrel{\text{def}}{=}\ & \mathsf{R}' \\ & \text{for } \psi(\mathsf{u},\mathsf{R}) \\ & \text{such that } \psi'(\mathsf{u},\mathsf{R}')\end{aligned}$$

where u is a concept instance in $\mathcal{U}_{\mathsf{c}}$, R is the model repository on which u is applied, and R' is the resulting model repository that reflects the effect of having applied u on R. The assertion ψ predicates about the concept instance u and the modeling artifacts in the model repository R, and states the conditions that must hold on u and R for the partial function denoting u to be well-defined. The assertion ψ' predicates about u and R', and states the conditions that must hold for the resulting model repository R' to be well-defined. For any potential resulting model repository that satisfies ψ', the model repository that includes the minimal set of changes is considered the result. It is important to notice that this is a common

practice of semantic approaches as it reduces the set of assertions required to unambiguously defined the result. If the *minimization selection* is not used, the assertion ψ' would also have to state all the facts that do not hold on the result so as to exclude any other (non-minimal) potential model repository.

In Section §3.2.1 we defined $\nu : (\mathcal{R}^* \bigcup \mathit{Descriptors}) \hookrightarrow \mathcal{F}_{\mathsf{MgMM}}$ as the special case of representation-of function that is total, eager, and renders the most possibly detailed representation for every modeling artifact, according to the assertions that are known to hold on those artifacts. Function ν was declared in Table 3.6. This function was used in the semantic equations for the semantic function $\mathcal{M}$ to produce model fragments capturing all the known characterization regarding modeling artifacts, which is then defined within the megamodels in our specification. For instance, in the semantic equations for ModelKind in Table 3.10, for every model kind mk we used function ν to capture in its representative megamodel mkMgM, the representation of all the modeling artifacts participating in the denotation of mk, together with their properties and relationships. In the semantic equations for $\mathcal{D}$, we use in the assertion ψ' the same function ν to capture in the corresponding megamodels, the new characterization information regarding the modeling artifacts in the resulting model repository. It is important to remark that as a consequence, the megamodels capturing information related to those modeling artifacts about which the assertion ψ' predicates, are expected to be updated as part of the effect denoted by the semantic function $\mathcal{D}$. Even though we do not explicitly state it in the semantic equations for $\mathcal{D}$ to avoid cluttering in the definitions, for each megamodel mgm that is updated to mgm′ due to the new characterization stated in ψ', we assume that the substitution operator is applied to produce the resulting model repository. For instance, given $[\![\mathsf{u}]\!]_{\square}\mathsf{R} \stackrel{\mathsf{def}}{=} \mathsf{R}'$ such that $\psi'(\mathsf{u}, \mathsf{R}')$, we assume the actual result to be $\mathsf{R}'[\mathsf{mgm_i}/\mathsf{mgm}'_\mathsf{i}]$ where the megamodels $\mathsf{mgm_i}$ are those affected by the assertions in ψ'.

The semantic function $\mathcal{D}$ is compositional on the semantic function $\mathcal{M}$, but not vice versa. In the semantic equations of $\mathcal{D}$, compositional applications of $\mathcal{M}$ can take place in the assertion ψ restricting the arguments and the assertion ψ' characterizing the result. The semantic function $\mathcal{D}$ is also compositional on itself. The denotation of instances of complex constructs, such as ArchitectureDecision, ArchitectureSolution and ArchitectureUpdateScript, are expressed in terms of the denotation of the composing concept instances. We restrict the compositional application of $\mathcal{D}$ to occur only in the assertion ψ' that characterizes the result. By this means, our semantic equations do not require any update on the input model repository to determine if the denotation is possible, i.e. to determine whether ψ holds.

4.2.3 Semantics of Architecture Design

The definition of the semantics for the software architecture design practice is twofold: to specify the structural representation of design concepts in terms of modeling constructs, and to specify the behavioral effect of the fine-grained actions that can be performed on architecture descriptions during an architecture design effort. In order to cope with the structural representation, in Section §4.2.2 we extended the definition of the semantic function $\mathcal{M}$ to also cover architecture design concepts. In order to cope with the behavioral effect, we also introduced the semantic function $\mathcal{D}$ that maps behavioral constructs in the syntactic

domain $\mathcal{U}_c$ to their denotation expressed in terms of the model repository resulting from applying the corresponding modifications on the modeling artifacts residing in a working model repository.

In this section, we define the semantic equations of $\mathcal{M}$ with respect to architecture design concepts. Together with the semantic equations defined in Sections §3.2.2 and §3.2.3, they complete the definition of $\mathcal{M}$. Also, we define the semantic equations for $\mathcal{D}$ for instances of behavioral design concepts. Provided that the syntactic domain $\mathcal{U}_c$ of $\mathcal{D}$ is an infinite set, we provide a semantic equation for a generic instance of each behavioral design concept in $\mathcal{U}^C$. For the definition, we use a top-down approach following the compositional structure of the concepts.

Architecture Design Library

In Section §4.1.1 we introduced the concept ArchitectureDesignLibrary as an aggregation of architecture design mechanisms such as architecture patterns, styles and tactics. We illustrated the conceptual model in Figure 4.6. We consider these constructs to be complex work products, i.e. they require multiple artifacts to be completely represented. These constructs conform, and their representing artifacts, conform system-independent reusable assets capturing prefabricated design mechanisms.

From the perspective of our formal semantics for these concepts, we consider them as *descriptive* concepts as their purpose is to capture design mechanisms, and not to represent the application or usage of such mechanisms during an architecture design effort. The application of these mechanisms is achieved by the corresponding *apply* statements of our fine-grained architecture design language. As a consequence, we only define the semantic equations of $\mathcal{M}$ corresponding to these concepts, as no semantic equation of $\mathcal{D}$ is needed. Formally, while these concepts belong to $\mathcal{A}^C$, they do not belong to $\mathcal{U}^C$. We define the denotation of these concepts following the same strategy that we applied in Sections §3.2.2 Section §3.2.3 for the definition of complex concepts in $\mathcal{A}^C$ and their instances in $\mathcal{A}_c$. As they represent complex work products, we require multiple modeling artifacts to completely and correctly denote them. First, (i) each concept instance is mapped to a representative megamodel capturing all the composing modeling artifacts that conforms the representation of the concept instance, their properties and interrelation. The megamodel also captures the relation of the concept instance being mapped, to the linked concept instances according to the associations in the conceptual model. To this end, the representative megamodel of the concept instance includes the representative megamodel corresponding to the denotation of the linked concept instances. Also, (ii) each concept instance is mapped to a terminal model that captures the documentation information on the instance, including administrative data, acronyms, definitions, reference materials, usage examples, application scenarios, etc. Consequently, (iii) the metamodel providing the modeling language for that terminal model is part of the denotation of the concept itself, which provides the infrastructure for the denotation of its instances. Additionally, (iv) as the concept instances are linked to Concern instances, the mapping includes a weaving model annotating the linked model elements in the terminal models capturing the concerns. (v) The corresponding metamodel for the weaving model is part of the denotation of the concept itself. In the case of ArchitecturePattern and

1	$[\![\mathsf{ArchitecturePattern} : \circ]\!]$
2	$\quad\overset{\text{def}}{=} [\![\mathsf{Concern} : \circ]\!] \ \cup\ [\![\mathsf{ArchitectureStyle} : \circ]\!] \ \cup\ [\![\mathsf{ModelKind} : \circ]\!] \ \cup$
3	$\quad [\![\mathsf{CorrespondenceKind} : \circ]\!] \ \cup\ [\![\mathsf{CorrespondenceRule} : \circ]\!] \ \cup$
4	$\quad \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel}, \mathsf{TMM}_\rho :_\bullet \mathsf{Metamodel},$
5	$\quad\ \mathsf{ArchitecturePatternDocMM}_\rho :_\bullet \mathsf{Metamodel},$
6	$\quad\ \mathsf{ArchitecturePatternCWMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
7	$\quad \mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge\ \mathsf{TMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
8	$\quad \mathsf{ArchitecturePatternDocMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
9	$\quad \mathsf{ArchitecturePatternCWMM}_\rho \lhd \mathsf{MMM}_\rho$
10	$[\![\mathsf{ap} :_\circ \mathsf{ArchitecturePattern}]\!]$
11	$\quad\overset{\text{def}}{=} [\![\mathsf{ArchitecturePattern} : \circ]\!] \ \cup$
12	$\quad [\![\mathsf{ap.concerns}]\!] \ \cup\ [\![\mathsf{ap.styles}]\!] \ \cup\ [\![\mathsf{ap.modelKinds}]\!] \ \cup$
13	$\quad [\![\mathsf{ap.correspondenceKinds}]\!] \ \cup\ [\![\mathsf{ap.requiredCorrespondenceRules}]\!] \ \cup$
14	$\quad \{\mathsf{apDocM} :_\bullet \mathsf{TerminalModel},$
15	$\quad\ \mathsf{apCWM} :_\bullet \mathsf{WeavingModel},$
16	$\quad\ \mathsf{apParamMM}_\rho :_\bullet \mathsf{Metamodel},$
17	$\quad\ \mathsf{apTM}_\rho :_\bullet \mathsf{TransformationModel},$
18	$\quad\ \mathsf{apMgM} :_\bullet \mathsf{Megamodel}\}$ such that
19	$\quad \mathsf{apDocM} \lhd \mathsf{ArchitecturePatternDocMM}_\rho \ \wedge\ \mathsf{ap}_\mu \lesssim \mathsf{apDocM} \ \wedge$
20	$\quad \mathsf{apCWM} \lhd \mathsf{ArchitecturePatternCWMM}_\rho \ \wedge\ \mathsf{apCWM} \blacktriangleright \mathsf{apCMW}\langle \mathsf{CM} \rangle \ \wedge$
21	$\quad \forall \mathsf{c} \in \mathsf{ap.concerns}.\ \langle \mathsf{ap}, \mathsf{c} \rangle_\mu \lesssim \mathsf{apCWM} \ \wedge$
22	$\quad \mathsf{apParamMM}_\rho \lhd \mathsf{MMM}_\rho \ \wedge$
23	$\quad \mathsf{apTM}_\rho \lhd \mathsf{TMM}_\rho \ \wedge$
24	$\quad \mathsf{apTM}_\rho \blacktriangleright \mathsf{apT}\left\langle \overline{\mathsf{a}_1}, \ldots, \overline{\mathsf{a}_m}, \mathsf{apParamMM}_\rho \twoheadrightarrow \varnothing \right\rangle$
25	$\qquad$ where $\forall \mathrm{i} \in \mathbb{N}, 0 < \mathrm{i} \leq m.\ (\exists\, \mathsf{mk} \in \mathsf{ap.modelKinds}.\ \mathsf{mkMM} = \mathsf{a_i} \ \vee$
26	$\qquad\qquad \exists\, \mathsf{ck} \in \mathsf{ap.correspondenceKinds}.\ \mathsf{ckWMM} = \mathsf{a_i}) \ \wedge$
27	$\quad \forall \mathsf{s} \in \mathsf{ap.styles}.\ \mathsf{sMgM}_\nu \lesssim \mathsf{apMgM} \ \wedge$
28	$\quad \forall \mathsf{mk} \in \mathsf{ap.modelKinds}.\ \mathsf{mkMgM}_\nu \lesssim \mathsf{apMgM} \ \wedge$
29	$\quad \forall \mathsf{ck} \in \mathsf{ap.correspondenceKinds}.\ \mathsf{ckMgM}_\nu \lesssim \mathsf{apMgM} \ \wedge$
30	$\quad \forall \mathsf{cr} \in \mathsf{ap.requiredCorrespondenceRules}.\ \mathsf{crMgM}_\nu \lesssim \mathsf{apMgM} \ \wedge$
31	$\quad \nu(\{\mathsf{MMM}_\rho, \mathsf{TMM}_\rho, \mathsf{ArchitecturePatternDocMM}_\rho, \mathsf{apDocM},$
32	$\qquad \mathsf{CMM}_\rho, \mathsf{CM}, \mathsf{ArchitecturePatternCWMM}_\rho, \mathsf{apCWM},$
33	$\qquad \mathsf{apParamMM}_\rho, \mathsf{apTM}\}) \lesssim \mathsf{apMgM}$

Table 4.5: *Semantic equations:* ArchitecturePattern.

ArchitectureTactic, the denotation also includes a transformation model that encapsulates the parametric knowledge on how the corresponding terminal models of the associated concept instances are manipulated to reflect the effect of applying the pattern or tactic. We explain in detail the semantic equations for ArchitecturePattern in what follows. The semantic equations for the other concepts are analogous and follows the same strategy explained above.

Table 4.5 defines the semantic equation for the concept ArchitecturePattern (1) and its instances (10). The denotation of ArchitecturePattern : ○ is compositional on the denota-

tion of its associated concepts (2–3), according to the compositional traversal determined by the aggregation and navigable associations in the conceptual model of Figure 4.6. The denotation also includes the customized metametamodel MMM_ρ (4) that determines the technical space, the customized metamodel for megamodels MgMM_ρ (4), a customized model transformation language TMM_ρ (4) for defining model transformations, a customized metamodel $\mathsf{ArchitecturePatternDocMM}_\rho$ (5) for documentation models, and a customized metamodel $\mathsf{ArchitecturePatternCWMM}_\rho$ (6) for weaving models capturing the relationship between concerns and a particular pattern.

The denotation of every concept instance ap :∘ ArchitecturePattern is compositional on the denotation of its concept (11) and on the denotation of all related concept instances (12–13) according to the compositional traversal. It also includes the terminal model apDocM (14) that conforms to $\mathsf{ArchitecturePatternDocMM}_\rho$ (19) and captures the documentation information regarding the pattern ap (19). Analogously to how we proceeded in the denotation for model kinds in Section §3.2.2 declared in Table 3.10, the denotation of ap includes a weaving model apCWM (15) to capture the relationship between the pattern ap and its concerns. apCWM is actually an annotation model containing links to concerns represented in the concern model CM (obtained in the denotation of Concern : ∘), and capturing additional information regarding the relationship to the pattern. The customized metamodel $\mathsf{ArchitecturePatternCWMM}_\rho$ (6) provides the constructs for this information (20). A significant modeling artifact in the denotation of ap is the transformation model apTM_ρ (17). It is expressed in terms of the model transformation language TMM_ρ (23), and realizes the model transformation apT (24) that takes m InOut terminal models and one In parameter model. The transformation model encapsulates the knowledge on how to manipulate the terminal models in order to enforce the pattern, i.e. to reflect its solution in terms of model elements within the terminal models. The parameter model allows to customize the application of the pattern, making the transformation model useful in multiple scenarios. The $m + 1$ parameters of the transformation apT (24) are specified using the metamodels defined in the denotation of the model kinds and correspondence kinds that the pattern operates on. In other words, given that a pattern operates on a particular model kind mk, then the metamodel mkMM defined in the denotation of mk (see Table 3.10) is used as the type of the InOut parameter of apT. Then, when we apply the pattern using the corresponding *update statement*, we use the terminal model amM of any architecture model am that is governed by the model kind mk. We explain this latter when we define the semantic equations for the ApplyPattern statement.

Finally, the denotation of ap also includes a megamodel apMgM that captures the complete representation of the modeling artifacts that participate in the denotation of ap (27–33). In the case of ArchitecturePattern we face an scenario that we have not faced before in the formal semantics specification. As indicated in the conceptual model in Figure 4.6, there are two associations between the concepts ArchitecturePattern and CorrespondenceRule, one indicating the set of correspondence rules that the pattern requires to be enforced once applied, and the other indicating the set of correspondence rules that are directly defined by the pattern. We made this distinction as correspondence rules can be reused in multiple scenarios and hence, a pattern can require to enforce a rule that is defined along the pattern definition, or that is defined elsewhere such as in an architecture framework. The semantic denotation of a pattern ap captures in its megamodel apMgM the representative megamodel for *all* correspondence rules (30), both defined and required, as we know from the conceptual model

that ap.definedCorrespondenceRules ⊆ ap.requiredCorrespondenceRules. In order to preserve in the megamodel the information of which of the correspondence rules is actually being defined, and not only required, we attach metadata to the corresponding element in the megamodel. This is accomplished by the representation-of function ν (30) that produces the model fragment that is expected to be defined within the megamodel apMgM.

Architecture Solutions, Decisions & Rationale

In Section §4.1.3 we defined the conceptual model for architecture decisions and rationale, and we introduced the concept ArchitectureSolution that provides an explicit conceptualization of each alternative solution devised and weighted by the architect when making a decision on how to address a particular set of concerns or requirements. We illustrated the conceptual model in Figure 4.20. We consider these constructs as complex work products conforming system-dependent assets that capture the actual procedure on how to build the architecture description of the system from the start.

From the perspective of our formal semantics, we consider ArchitectureDecision both as an *descriptive* and *behavioral* concept and hence, we require semantic equations for both $\mathcal{M}$ and $\mathcal{D}$ covering this concept and its instances. As we reviewed in Section §4.1.3, various authors have provided different techniques to capture architecture decisions within an architecture description, such as textual templates, dedicated models, and dedicated architecture viewpoints. While we are committed to a model-based representation of architecture descriptions and the related constructs conceived in the architecture description practice, the goal of our semantics is not to mandate nor restrict the techniques used to capture architecture descriptions, and in particular, architecture decisions. Hence, our formalization aims to provide the foundation for these techniques to be reified. However, we are also committed to a paradigm-shift towards a first-class representation on how to build architecture descriptions by means of decisions and solutions, and hence, we need those decisions and solutions to be independently captured so as they can be shared and reused, and the design effort can be repeatable. As a consequence, we consider each architecture decision as a complex work product in itself, and hence, requiring of a set of modeling artifacts to be completely represented in terms of modeling constructs. Thus, we follow the same strategy that we discussed before for the case of architecture design libraries, to define the structural semantics for decisions. Table 4.6 defines the semantic equations for ArchitectureDecision and its instances. Each decision is mapped to a representative megamodel (14) capturing the set of modeling artifacts that participates in the denotation, their properties and relationships (21–22). Also, the megamodel captures the representative megamodel of the alternative architecture solutions (20), and the representative megamodel of the related architecture decisions (19). In addition, each decision is mapped to a documentation model (12) capturing not only administrative and technical information related to the decision, but mainly the rationale justifying the decision. Provided that our conceptualization establishes a one-to-one relationship between decisions and rationale, we embed the documentation of the rationale in the documentation of the decision itself, avoiding the need for an additional artifact (15).

It is important to notice, however, that other techniques can still be captured using the formalization techniques that we already defined and covered in Section §3.2.2. For instance,

approaches using annotation or companion models to architecture models, as in [LJA09], can be captured defining a specific model kind for the companion model, adding such model kind to every architecture viewpoint in use in the architecture description, and having in each architecture view an architecture model governed by the defined model kind. The relation between the companion model and the architecture models can be established using correspondences and correspondence rules. Approaches using dedicated decisions models, as in [DC05, KCD09] can also be addressed by defining a purpose-specific model kind, having and architecture model governed by this kind, and using correspondences and correspondence rules for capturing relationships. In order to include the model kind in the architecture description, a purpose-specific architecture viewpoint and the corresponding architecture view must also be defined. Approaches using a dedicated architecture viewpoints, as in [HAH12a, HAH12b], can be addressed using our formalization for architecture viewpoints directly. Besides, the denotation of architecture decisions defined by our formalization can be interrelated with the information captured by these approaches, either statically using correspondences, or assisted by model transformations to produce or relate elements in our denotation to the elements resulting from the denotation of the approaches.

We consider each decision d also as a behavioral instance that, when applied, have an effect in the current state of the architecture description captured in the model repository of the modeling environment. Table 4.6–(23) defines the semantic equation for the semantic function $\mathcal{D}$ that denotes the effect of making a decision. First, each architecture decision d defines a set of addressed concerns and a set of raised concerns. Provided that these concerns may not be captured yet in the architecture description, the effect of d yields those concerns to be merged into the concerns defined by the architecture description. To this end, we use the transformation operator $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{CMergeTM}_\rho)$ on the transformation model $\mathsf{CMergeTM}_\rho$ defined in the infrastructure modeling artifacts, in order to perform the merged concern model (27). Additionally, the effect of d yields a model repository that includes all the modeling artifacts required to denote d (26) and the the representative megamodel adMgM of the architecture description now refers to the representative megamodel of the architecture decision dMgM (25). By this means we capture the fact that the decision d was made in the design of the architecture being capture by the architecture description ad. Finally, when the architect makes a decision, i.e. the architect selects the most suitable architecture solution from the set of alternatives devised and weighted, that selected solution must be enforced in the current architecture description. Hence, the effect of d is the effect of its *selected* solution. Then, being R' the model repository in which the concerns where updated and the modeling artifacts required to denote d are defined, the effect of d yields the effect of $\mathsf{d.solution}$ on R' (24).

One of the most significant aspects to be captured about decisions is the set of architecture description elements that are affected or impacted by the solution. While this is captured by means of an association in the conceptual model illustrated in Figure 4.20, our semantic equations do not explicitly capture this information. As we discuss later in Section §4.2.4 when we explain how traceability is achieved in our formalization, automatic traceability is one of the benefits of applying a strict model-based approach to architecture description and design. Using modeling artifacts to represent every architecture description element, and using model transformations to represent the manipulation of these model elements, allow us to automatically extract traceability information, such as the impacted architecture

1	$\llbracket \mathsf{ArchitectureDecision} : \circ \rrbracket$
2	$\quad \stackrel{\text{def}}{=} \llbracket \mathsf{Concern} : \circ \rrbracket \;\cup\; \llbracket \mathsf{ArchitectureSolution} : \circ \rrbracket \;\cup$
3	$\qquad \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel},$
4	$\qquad\; \mathsf{ArchitectureDecisionDocMM}_\rho :_\bullet \mathsf{Metamodel},$
5	$\qquad\; \mathsf{ArchitectureDecisionCWMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
6	$\qquad \mathsf{MgMM}_\rho \triangleleft \mathsf{MMM}_\rho \;\wedge$
7	$\qquad \mathsf{ArchitectureDecisionDocMM}_\rho \triangleleft \mathsf{MMM}_\rho \;\wedge$
8	$\qquad \mathsf{ArchitectureDecisionCWMM}_\rho \triangleleft \mathsf{MMM}_\rho$
9	$\llbracket \mathsf{d} :_\circ \mathsf{ArchitectureDecision} \rrbracket$
10	$\quad \stackrel{\text{def}}{=} \llbracket \mathsf{ArchitectureDecision} : \circ \rrbracket \;\cup\; \llbracket \mathsf{d.dependedUponDecisions} \rrbracket \;\cup$
11	$\qquad \llbracket \mathsf{d.concerns} \rrbracket \;\cup\; \llbracket \mathsf{d.raisedConcerns} \rrbracket \;\cup$
12	$\qquad \{\mathsf{dDocM} :_\bullet \mathsf{TerminalModel},$
13	$\qquad\; \mathsf{dCWM} :_\bullet \mathsf{WeavingModel},$
14	$\qquad\; \mathsf{dMgM} :_\bullet \mathsf{Megamodel}\}$ such that
15	$\qquad \mathsf{dDocM} \triangleleft \mathsf{ArchitectureDecisionDocMM}_\rho \;\wedge\; \mathsf{d}_\mu \lesssim \mathsf{dDocM} \;\wedge\; \mu(\mathsf{d.rationale}) \lesssim \mathsf{dDocM} \;\wedge$
16	$\qquad \mathsf{dCWM} \triangleleft \mathsf{ArchitectureDecisionCWMM}_\rho \;\wedge\; \mathsf{dCWM} \blacktriangleright \mathsf{dCMW}\,\langle \mathsf{dCM} \rangle \;\wedge$
17	$\qquad \forall \mathsf{c} \in \mathsf{d.concerns}.\; \langle \mathsf{d}, \mathsf{c} \rangle_\mu \lesssim \mathsf{dCWM} \;\wedge$
18	$\qquad \forall \mathsf{c} \in \mathsf{d.raisedConcerns}.\; \langle \mathsf{d}, \mathsf{c} \rangle_\mu \lesssim \mathsf{dCWM} \;\wedge$
19	$\qquad \forall \mathsf{d}' \in \mathsf{d.dependedUponDecisions}.\; \mathsf{d'MgM}_\nu \lesssim \mathsf{dMgM} \;\wedge$
20	$\qquad \forall \mathsf{s} \in \mathsf{d.alternatives}.\; \mathsf{sMgM}_\nu \lesssim \mathsf{dMgM} \;\wedge$
21	$\qquad \nu(\{\mathsf{MMM}_\rho, \mathsf{ArchitectureDecisionDocMM}_\rho, \mathsf{dDocM},$
22	$\qquad\quad \mathsf{CMM}_\rho, \mathsf{dCM}, \mathsf{ArchitectureDecisionCWMM}_\rho, \mathsf{dCWM}\}) \lesssim \mathsf{apMgM}$
23	$\llbracket \mathsf{d} :_\square \mathsf{ArchitectureDecision} \rrbracket\; \mathsf{R}$
24	$\quad \stackrel{\text{def}}{=} \llbracket \mathsf{d.solution} \rrbracket_\square\; \mathsf{R}'$
25	$\qquad$ such that $\mathsf{dMgM}_\nu \lesssim \mathsf{adMgM} \;\wedge$
26	$\qquad\qquad \mathsf{R}' \equiv \mathsf{R}[\mathsf{adCM}/\mathsf{adCM}'] \;\cup\; \mathcal{M}\llbracket \mathsf{d} \rrbracket$
27	$\qquad\qquad$ where $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{CMergeTM}_\rho, \langle \mathsf{adCM}, \mathsf{dCM} \rangle) = \{\mathsf{adCM}'\}$

Table 4.6: *Semantic equations:* ArchitectureDecision.

description elements, by modeling techniques such as difference between models and the trace output generated by transformation engines when executing model transformations. Hence, by having a current model repository R and obtaining its subsequent version R' by making a decision d and applying the script of the architecture solution selected by d, modeling techniques provide the architect the traceability information both by processing R and R' together with the produced trace information from the transformation engines.

The ArchitectureSolution is also both *descriptive* and *behavioral*, and hence it is formalized by means of both semantic functions $\mathcal{M}$ and $\mathcal{D}$. As illustrated in the conceptual model in Figure 4.20, an architecture solution involves a set of concerns and optionally captures an architecture update script. While the solution states the architect's intention, the script reifies this intention in terms of architecture update statements. The structural denotation of each $\mathsf{s} :_\circ \mathsf{ArchitectureSolution}$ is defined following the same strategy as before. A megamodel

1	$[\![\mathsf{s} :_{\square} \mathsf{ArchitectureSolution}]\!]\ \mathsf{R} \stackrel{\text{def}}{=} [\![\mathsf{s.script}]\!]_{\square}\ \mathsf{R}$
2	for $\mathsf{s.script} \rightarrow \mathsf{notEmpty}()$

Table 4.7: *Semantic equations:* ArchitectureSolution.

capturing the set of involved modeling artifacts, a terminal model providing documentation information including benefits and liabilities, and a weaving model as an annotation of the concerns model CM. Also, the megamodel for s also contains a reference to the representative megamodel of the script reifying the solution, if any. We omit the formal definition of $\mathcal{M}[\![\mathsf{s} :_{\circ} \mathsf{ArchitectureSolution}]\!]$ as it is analogous to that for architecture decisions defined in Table 4.6.

Table 4.7 defines the semantic equation for an architecture solution s from the behavioral perspective. The effect of a solution is actually the effect produced by the architecture update script that reifies the solution (1). However, as an architecture solution may not provide a reifying script, we have to limit the definition only for those cases in which an script is actually defined (2). It is important to notice, however, that in our conceptualization, for every $\mathsf{d} :_{\square} \mathsf{ArchitectureDecision}$, its selected solution d.solution always provides a reifying script as it is stated in the invariant on the conceptual model in Figure 4.20. As a consequence, the effect of d relies on the effect of d.solution, that relies on the effect of d.solution.script that is always defined and hence, the semantic equation in Table 4.7–(1) can be applied.

Architecture Update Scripts

In Section §4.1.2 we introduced the concept of ArchitectureUpdateScript that allows the architect to encapsulate the knowledge on how to update an architecture description to achieve a given purpose. An architecture update script consists of a non-empty sequence of ArchitectureUpdateStatement. The different kinds of statements represent the fine-grained actions that can be performed on an architecture description. We illustrated the conceptual model for scripts and statements in Figure 4.8. From the perspective of our formal semantics for these concepts, we consider them both as *descriptive* and *behavioral*, and hence we require semantic equations for both semantic functions $\mathcal{M}$ and $\mathcal{D}$. Table 4.8 defines the semantic equations for the concept ArchitectureUpdateScript and its instances.

We consider an architecture update script as complex work product conforming a system-dependent asset that can be reused among different architecture solutions and mainly, among different architecture update scripts by means of the CallScript statement. As a consequence, the denotation of a $\mathsf{us} :_{\circ} \mathsf{ArchitectureUpdateScript}$ includes a megamodel usMgM (8) capturing all composing modeling artifacts, and that is used as the representative artifact when referring to the script from the denotation of solutions and call statements. However, we do not consider architecture update statements as work products, but rather, as the constituent elements of architecture update scripts. As we explained in Section §4.1.2, scripts are expressed in the domain-specific architecture design modeling language that we defined for this purpose. We use this same language to provide both conceptualization and denotation for statements. To this end, we define the metamodel USMM_{ρ} (4) which constructs are those de-

1 $\llbracket \mathsf{ArchitectureUpdateScript} : \circ \rrbracket$

2 $\overset{\text{def}}{=} \bigcup_{\mathsf{X} \in \mathcal{S}^{\mathsf{C}}} \llbracket \mathsf{X} \rrbracket \;\; \cup$

3 $\{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel},$

4 $\mathsf{USMM}_\rho :_\bullet \mathsf{Metamodel}, \mathsf{USWMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that

5 $\mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge \;\; \mathsf{USMM}_\rho \lhd \mathsf{MMM}_\rho \;\; \wedge \;\; \mathsf{USWMM}_\rho \lhd \mathsf{MMM}_\rho$

6 $\llbracket \mathsf{us} :_\circ \mathsf{ArchitectureUpdateScript} \rrbracket$

7 $\overset{\text{def}}{=} \llbracket \mathsf{ArchitectureUpdateScript} : \circ \rrbracket \;\; \cup \;\; \llbracket \mathsf{us.statements} \rrbracket \;\; \cup$

8 $\{\mathsf{usMgM} :_\bullet \mathsf{Megamodel}, \mathsf{usM} :_\bullet \mathsf{TerminalModel}, \mathsf{usWM} :_\bullet \mathsf{WeavingModel}\}$ such that

9 $\mathsf{usM} \lhd \mathsf{USMM}_\rho \;\; \wedge \;\; \mathsf{us}_\mu \lesssim \mathsf{usM} \;\; \wedge \;\; \forall \mathsf{s} \in \mathsf{us.statements}.\;\; \mathsf{s}_\mu \lesssim \mathsf{usM} \;\; \wedge$

10 $\mathsf{usWM} \lhd \mathsf{USWMM}_\rho \;\; \wedge \;\; \mathsf{usWM} \blacktriangleright \mathsf{usMW}\,\langle \mathsf{usM}, \mathsf{usMgM} \rangle \;\; \wedge$

11 $\forall \mathsf{s} \in \mathsf{us.statements}.\;\; \forall \mathsf{a} \in \llbracket \mathsf{s} \rrbracket.\;\; \langle \mathsf{s}_\mu, \mathsf{a}_\mu \rangle_\mu \lesssim \mathsf{usWM} \;\; \wedge$

12 $\mathsf{usMgM} \lhd \mathsf{MgMM}_\rho \;\; \wedge$

13 $\forall \mathsf{s} \in \mathsf{us.statements}.\;\; \nu(\llbracket \mathsf{s} \rrbracket) \lesssim \mathsf{usMgM} \;\; \wedge$

14 $\nu(\{\mathsf{MMM}_\rho, \mathsf{MgMM}_\rho, \mathsf{USMM}_\rho, \mathsf{USWMM}_\rho, \mathsf{usM}, \mathsf{usWM}\}) \lesssim \mathsf{usMgM}$

15 $\llbracket \mathsf{us} :_\square \mathsf{ArchitectureUpdateScript} \rrbracket \; \mathsf{R}$

16 $\overset{\text{def}}{=} (\llbracket \mathsf{s}_n \rrbracket_\square \circ \ldots \circ \llbracket \mathsf{s}_1 \rrbracket_\square) \; \mathsf{R}$

17 for $\mathsf{us.statements} = \langle \mathsf{s}_1, \ldots, \mathsf{s}_n \rangle$

Table 4.8: *Semantic equations:* ArchitectureUpdateScript.

fined in Figures 4.8 to 4.19. Then, the denotation of an architecture update script us includes a terminal model usM (8) conforming to USMM_ρ (9), and that captures all the statements in the script (9).

From the perspective of the descriptive denotation of statements, we distinguish three classes. Class (A) includes those statements that are self-contained and need no additional modeling artifact to be denoted, such as StartFromScratch. Class (B) includes those statements that refer to modeling artifacts that are already defined and hence that are available in the model repository in the workspace of the modeling environment or defined in a model repository reachable from the modeling environment, such as CreateEmptyArchitectureModel and UseArchitectureFramework respectively. Class (C) includes those statements that, in addition to requiring already existing elements, they require concrete modeling artifacts to be provided by the architect, such as ApplyPattern. The terminal model usM captures the declaration of the statements, but it does not capture the related modeling artifacts that are present in class (B) and (C). We capture these related modeling artifacts in the megamodel usMgM (13). For statements in class (B), the representative megamodel of the referred concept instances are captured in usMgM. For statements in class (C), in addition to the representative megamodels as for class (B), the statement-specific modeling artifacts provided by the architect are captured in the megamodel. The set of modeling artifacts to be captured are provided by the compositional application of the semantic function $\mathcal{M}$ on the statements. While $\mathcal{M}$ applied to the concept provides the infrastructure artifacts, $\mathcal{M}$ applied to the concept instance provides the instance-specific artifacts. Formally, in Table 4.8–(2) we include the

1	$[\![\mathsf{StartFromScratch} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{StartFromScratch}]\!] \stackrel{\text{def}}{=} \varnothing$
3	$[\![\mathsf{s} :_{\square} \mathsf{StartFromScratch}]\!]\ \varnothing$
4	$\stackrel{\text{def}}{=} \mathcal{M}[\![\mathsf{ArchitectureDescription} : \circ]\!] \ \cup$
5	$\{\mathsf{adDocM} :_{\bullet} \mathsf{TerminalModel}, \mathsf{adMgM} :_{\bullet} \mathsf{Megamodel}\}$
6	such that $\mathsf{adDocM} \lhd \mathsf{ArchitectureDescriptionDocMM}_{\rho} \ \wedge$
7	$\mathsf{adMgM} \lhd \mathsf{MgMM}_{\rho} \ \wedge$
8	$\nu(\{\mathsf{MMM}_{\rho}, \mathsf{MgMM}_{\rho}, \mathsf{TMM}_{\rho}, \mathsf{ArchitectureDescriptionDocMM}_{\rho}, \mathsf{adDocM},$
9	$\mathsf{CMM}_{\rho}, \mathsf{adCM}, \mathsf{CMergeTM}_{\rho},$
10	$\mathsf{SMM}_{\rho}, \mathsf{adSM}, \mathsf{SCWMM}_{\rho}, \mathsf{adSCWM}, \mathsf{SMergeTM}_{\rho}\}) \lesssim \mathsf{adMgM}$

Table 4.9: *Semantic equations:* StartFromScratch *statement.*

infrastructure artifacts for every statement, and we include the statement-specific artifacts in the denotation of the script (7). In this latter case, the modeling artifacts are also captured in usMgM (13). Then, we have script's statements defined in the terminal model usM and their related modeling artifacts captured in usMgM. We still need to capture the relationship between them. To this end, we include a weaving model usWM (8) in the denotation of the script that relates elements in usM and elements in usMgM. For each statement s in the script and for each artifact a resulting from the denotation of s, a link is part of the weaving model (11). We provide an example later when we discuss the StartFromArchitectureDescription statement. We illustrate the example in Figure 4.21.

From the behavioral perspective, the effect of an architecture update script is the composed effect of the statements in the script, in the same order as they are defined in the script. Formally, the composed effect is the functional composition of the semantic function $\mathcal{D}$. Given a model repository R and a script us with $\mathsf{s}_1, \ldots, \mathsf{s}_n$ as its statements, the effect of applying us on R is the effect of sequentially applying each statement s_i on the model repository $\mathsf{R}_{\mathrm{i}-1}$ producing the model repository R_i. Having $\mathsf{R} \equiv \mathsf{R}_0$, the resulting model repository R_n from the application of the last statement is the resulting repository of applying the script (16).

In Figure 4.8 we defined eight kinds of statements that are available in our architecture design scripting language. In what follows, we define the semantic equations for the most representative statement or statements of each of these kinds.

Start Statements. A Start statement is used to set up the starting environment of an architecture design effort, and hence, it can be used just once and at the very beginning. As illustrated in our conceptual model in Figure 4.9, we distinguished three special kinds of Start statements.

The StartFromScratch statement is used to set up a initial environment containing the minimal set of modeling artifacts required to construct an architecture description. Table 4.9 defines the semantic equations for this statement. The architect does not need to provide any additional information in order to apply this statement, as opposed to the other kinds

1	$[\![\mathsf{StartFromArchitectureDescription} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{StartFromArchitectureDescription}]\!]$
3	with $\mathsf{ad}' \equiv \mathsf{s.architectureDescription}$
4	$\stackrel{\text{def}}{=} \{\mathsf{ad'MgM}\}$ such that $\mathsf{ad'MgM} \in \mathcal{M}[\![\mathsf{ad}']\!]$
5	$[\![\mathsf{s} :_{\square} \mathsf{StartFromArchitectureDescription}]\!] \varnothing \stackrel{\text{def}}{=} \mathcal{M}[\![\mathsf{s.architectureDescription}]\!]$

Table 4.10: *Semantic equations:* StartFromArchitectureDescription *statement.*

of Start statement where a prefabricated architecture description or an elsewhere defined reference architecture must be indicated. As a consequence, the semantic function $\mathcal{M}$ defining the structural denotation of StartFromScratch : ○ (1) and any instance $\mathsf{s} :_{\circ}$ StartFromScratch (2) yield empty sets. The behavioral effect of this statement (3) is captured by the semantic function $\mathcal{D}$. As it is the case for every start statement, it can only be applied on an empty model repository (3). The resulting set of modeling artifacts is determined by the structural denotation of an architecture description that we defined in Section §3.1.1 and particularly in Table 3.16. Thus, $[\![s]\!]_{\square}$ yields all the modeling artifacts providing the infrastructure for capturing architecture descriptions (4), that is produced by using the structural denotation of ArchitectureDescription : ○. In particular, it not only provides the reference models defining the necessary modeling languages, but also those modeling artifacts that capture and manipulate the concerns and stakeholders related to the architecture description. $[\![s]\!]_{\square}$ also yields an documentation model adDocM (5) that conforms to the corresponding metamodel (6). This model is initially empty. The architect can later improve the documentation by using the ImproveDocumentationOfArchitectureDescription statement. Finally, the result also includes a representative megamodel adMgM (5) that contains a representation of all the produced modeling artifacts (8–10).

It is important to remark that in the *such-that* section of the semantic equation for $\mathcal{D}$ (3) we refer to modeling artifacts that are part of the result set. For instance, we refer to the metamodel for documentation models $\mathsf{ArchitectureDescriptionDocMM}_{\rho}$ (6), to the concerns model adCM (9) and the stakeholder models adSM (10), that are produced by $[\![\mathsf{ArchitectureDescription} : \circ]\!]$ (4). In order to be strictly formal, these artifacts must also be introduced in the *such-that* section of the semantic equation for $\mathcal{D}$, stating their existence in addition to stating their properties and characteristics. However, we omit this kind of existence assertions to avoid unnecessary cluttering in the definitions when understanding is not compromised. When the modeling artifacts are not introduced in the *such-that* section, their definition can be found by traversing the compositional use of the semantic functions. For instance, in the case of the StartFromScratch statement, the definition of these artifacts can be found by traversing the result of applying $\mathcal{M}$ on the concept ArchitectureDescription (4).

Table 4.10 defines the semantic equations for the StartFromArchitectureDescription statement. In this case, the architect starts the design effort by using a prefabricated architecture description. As illustrated in the conceptual model for the statement in Figure 4.9, this already existing architecture description is conceived in terms of the conceptualization of the architecture description practice that we reviewed and formalized in Section §3.2.2. Formally, a statement instance $\mathsf{s} :_{\circ}$ StartFromArchitectureDescription refers to an instance of a

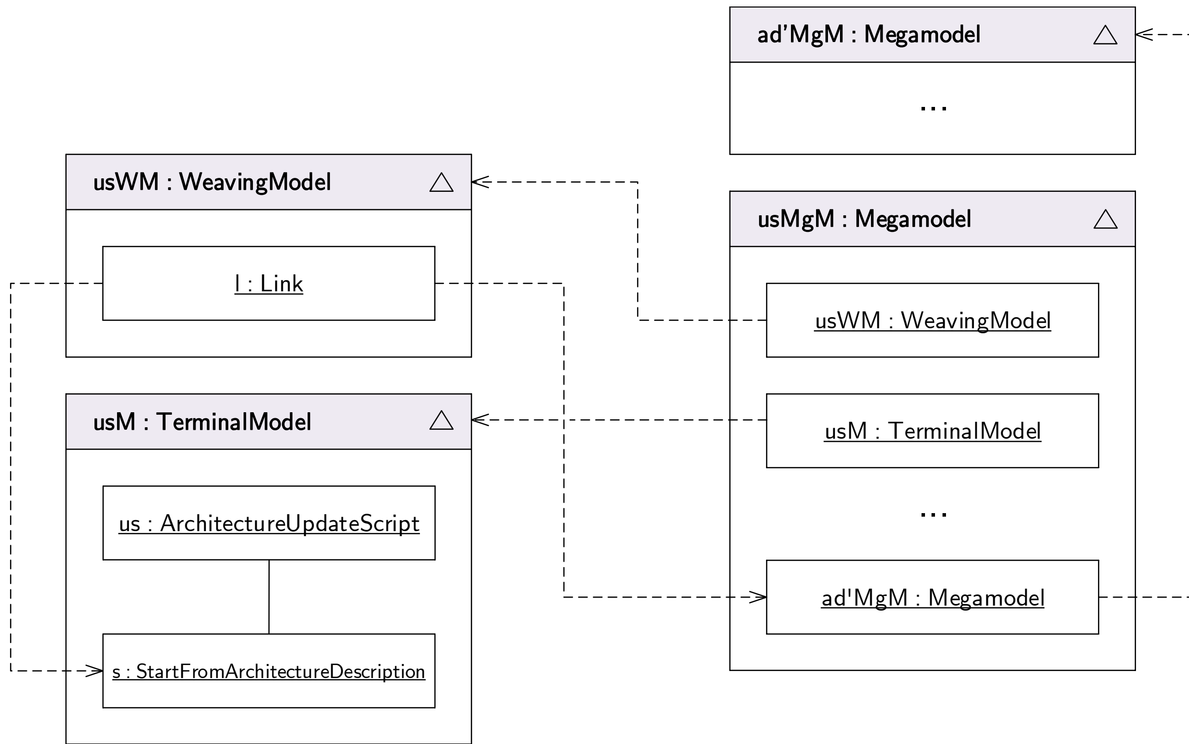


Figure 4.21: *Example of denotation of* StartFromArchitectureDescription *statement.*
The figure illustrates the main modeling artifacts in the denotation of an architecture update script us with a single StartFromArchitectureDescription statement s.

prefabricated architecture description ad′ :∘ ArchitectureDescription. Then, as conceptually we have that s.architectureDescription = ad′, the structural denotation of s must also capture this relationship. As we have homogeneously proceeded in our semantic formalization, we capture these relationships by including the representative megamodel of the referenced concept instance in the representative megamodel of the referencing one. In this case, we refer to the representative megamodel ad′MgM of ad′ :∘ ArchitectureDescription in the representative megamodel of the update script containing the statement s. As a consequence, we have that $\mathcal{M}[\![s]\!]$ yields {ad′MgM} (4), which is then part of the representative megamodel of the script as we defined in Table 4.8–(13). We illustrate in Figure 4.21 an example of these declarations and relationships. Let us be an architecture update script containing a single statement s :∘ StartFromArchitectureDescription. Let ad′ :∘ ArchitectureDescription be the architecture description referred by the start statement s. The figure shows the most significant modeling artifacts in a model repository R in which the script us is defined. It is important to remark that this is not the resulting set of artifacts after applying the semantic function $\mathcal{D}$. On the contrary, this is the set of artifacts describing the potential application of $\mathcal{D}$, i.e. prior to the application of $\mathcal{D}$. In the figure, ad′MgM is the representative megamodel of ad′ and usMgM is the representative megamodel of the script us. As defined in Table 4.8–(14), the megamodel usMgM refers to the terminal model usM capturing the script us and its statements, and to a weaving model usWM capturing the relationship between the statements and the existing modeling artifacts. In particular, usM contains an element representing the script us and its statement s. Also, usWM contains a link relating the model element representing s and the model element representing the megamodel ad′MgM in usMgM. By this means, the denotation captures the relationship between the concept instances s and ad′.

1	$[\![\mathsf{StartFromReferenceArchitecture} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{StartFromReferenceArchitecture}]\!]$
3	with $\mathsf{ra} \equiv \mathsf{s.referenceArchitecture}$
4	$\stackrel{\text{def}}{=} \{\mathsf{raMgM}\}$ such that $\mathsf{raMgM} \in \mathcal{M}[\![\mathsf{ra}]\!]$
5	$[\![\mathsf{s} :_{\square} \mathsf{StartFromReferenceArchitecture}]\!] \varnothing$
6	$\stackrel{\text{def}}{=} \mathcal{M}[\![\mathsf{s.referenceArchitecture.architectureDescription}]\!] \cup$
7	$\mathcal{M}[\![\mathsf{s.referenceArchitecture.architectureDesignLibrary}]\!]$
8	for $\mathsf{s.script} \rightarrow \mathsf{isEmpty}()$
9	$[\![\mathsf{s} :_{\square} \mathsf{StartFromReferenceArchitecture}]\!] \varnothing$
10	$\stackrel{\text{def}}{=} [\![\mathsf{s.script}]\!]_{\square} \mathsf{R}'$
11	for $\mathsf{s.script} \rightarrow \mathsf{notEmpty}()$
12	such that $\mathsf{R}' \equiv (\mathcal{M}[\![\mathsf{s.referenceArchitecture.architectureDescription}]\!] \cup$
13	$\mathcal{M}[\![\mathsf{s.referenceArchitecture.architectureDesignLibrary}]\!])$

Table 4.11: *Semantic equations:* StartFromReferenceArchitecture *statement.*

The behavioral effect of $\mathsf{s} :_{\square}$ StartFromArchitectureDescription yields the complete set of modeling artifacts that is required to represent s.architectureDescription (5). In other words, given the set of modeling artifacts representing the starting architecture description ad′, which is referred by the statement s, this set is imported into the modeling environment as the starting point of the design effort.

Table 4.11 defines the semantic equations for the StartFromReferenceArchitecture statement. As in the previous case, the structural denotation of this statement yields the representative megamodel raMgM of the referred ReferenceArchitecture (4) and its behavioral effect is to import all the modeling artifacts required to completely represent the architecture description of the reference architecture (6–7). In the case that the statement s defines an initialization script (9), the resulting set of modeling artifacts is the result of applying the script (10) to the imported modeling artifacts (12–13).

Identify Statement. An Identify statement is used to capture and refine the concerns and stakeholders in the architecture description. As we illustrated in the conceptual in Figure 4.10, $\mathsf{s} :_{\circ}$ Identify identifies a set of concerns, a set of stakeholders, and their interrelation.

From the semantics perspective, this statement belongs to class (C) as the architect must provide a set of modeling artifacts to fully specify the statement. In particular, as we define in Table 4.12, the semantics expects the architect to define a terminal model iCM (3) to capture the identified concerns, a terminal model iSM (3) to capture the identified stakeholders, and a weaving model iSCWM (3) capturing the interrelation between them (5). These models must conform to the corresponding metamodels that are part of the infrastructure modeling artifacts provided by the denotation of the concepts Concern and Stakeholder. The effect of the statement $\mathsf{i} :_{\square}$ Identify on a model repository R is to update the corresponding models capturing concerns and stakeholders that are part of the architecture description being

1	$[\![\mathsf{Identify} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{i} :_\circ \mathsf{Identify}]\!]$
3	$\stackrel{\text{def}}{=} \{\mathsf{iCM} :_\bullet \mathsf{TerminalModel}, \mathsf{iSM} :_\bullet \mathsf{TerminalModel}, \mathsf{iSCWM} :_\bullet \mathsf{WeavingModel}\}$ such that
4	$\mathsf{iCM} \triangleleft \mathsf{CMM}_\rho \quad \wedge \quad \mathsf{iSM} \triangleleft \mathsf{SMM}_\rho \quad \wedge$
5	$\mathsf{iSCWM} \triangleleft \mathsf{SCWMM}_\rho \quad \wedge \quad \mathsf{iSCWM} \blacktriangleright \mathsf{iSCMW}\langle \mathsf{iSM}, \mathsf{iCM}\rangle$
6	$[\![\mathsf{i} :_\square \mathsf{Identify}]\!]$
7	$\stackrel{\text{def}}{=} \mathsf{R}[\mathsf{adCM}/\mathsf{adCM}', \mathsf{adSM}/\mathsf{adSM}', \mathsf{adSCWM}/\mathsf{adSCWM}']$
8	such that $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{SMergeTM}_\rho, \langle \mathsf{adSM}, \mathsf{adCM}, \mathsf{adSCWM}, \mathsf{iSM}, \mathsf{iCM}, \mathsf{iSCWM}\rangle)$
9	$= \{\mathsf{adSM}', \mathsf{adCM}', \mathsf{adSCWM}'\}$

Table 4.12: *Semantic equations:* Identify *statement.*

designed, namely adCM, adSM and adSCWM. This update is achieved by using the transformation model $\mathsf{SMergeTM}_\rho$ defined in the infrastructure modeling artifacts for the Stakeholder concept in Table 3.9. This transformation captures the knowledge on how to merge a set of concerns and stakeholder models into another set of concerns and stakeholder models. In the case for the i statement, the corresponding transformation operator $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{SMergeTM}_\rho)$ available in the semantic domain is applied to produce an updated version of the concerns and stakeholders models of the architecture description (8–9). Then, the resulting model repository is the same model repository R where the content of these models is updated with the content of their new version (7).

Use Statements. A UseStatement allows the architect to determine the architecture frameworks and architecture description languages that the architecture description being designed must adhere to, and hence, to determine how the architecture description is structurally organized in terms of architecture viewpoints and model kinds. As we illustrated in the conceptual model in Figure 4.11, these statements belong to class (B) as the UseArchitectureFramework statement and the UseArchitectureDescriptionLanguage statement must indicate the ArchitectureFramework and the ArchitectureDescriptionLanguage to use, respectively. In the latter case, however, we restrict the statement to refer to architecture description languages that provide an architecture viewpoint for each provided model kind. Table 4.13 defines the semantic equations for the UseArchitectureFramework statement. The equations for the UseArchitectureDescriptionLanguage statement are analogous.

From the descriptive perspective, the denotation of $\mathsf{s} :_\circ \mathsf{UseArchitectureFramework}$ result in the representative megamodel afMgM of the architecture framework af to use or adhere to (4). From the behavioral perspective, the effect of s on a model repository R is complex. First, given that an architecture framework af captures a set of concerns and stakeholders that characterize the architecture viewpoints and model kinds that af defines, those concerns and stakeholders need to be incorporated into the architecture description. Similarly to how we proceeded in the case of the Identify statement, we use the transformation operation $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{SMergeTM}_\rho)$ available in the semantic domain to merge the concerns and stakeholders captured by af, and denoted in the corresponding modeling artifacts afCM, afSM and afSCWM, into the set of concerns and stakeholders captured by the architecture description

1 $[\![\mathsf{UseArchitectureFramework} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$

2 $[\![\mathsf{s} :_{\circ} \mathsf{UseArchitectureFramework}]\!]$

3 $\quad \text{with } \mathsf{af} \equiv \mathsf{s.architectureFramework}$

4 $\quad \stackrel{\text{def}}{=} \{\mathsf{afMgM}\} \text{ such that } \mathsf{afMgM} \in \mathcal{M}[\![\mathsf{af}]\!]$

5 $[\![\mathsf{s} :_{\square} \mathsf{UseArchitectureFramework}]\!]\mathsf{R}$

6 $\quad \text{with } \mathsf{af} \equiv \mathsf{s.architectureFramework}$

7 $\quad \stackrel{\text{def}}{=} \mathsf{R}[\mathsf{adCM}/\mathsf{adCM}', \mathsf{adSM}/\mathsf{adSM}', \mathsf{adSCWM}/\mathsf{adSCWM}'] \;\cup$

8 $\quad\quad \mathcal{M}[\![\mathsf{af}]\!] \;\cup\; \mathsf{R}_{\mathsf{views}}$

9 $\quad\quad \text{such that } \mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{SMergeTM}_\rho, \langle \mathsf{adSM}, \mathsf{adCM}, \mathsf{adSCWM}, \mathsf{afSM}, \mathsf{afCM}, \mathsf{afSCWM} \rangle)$

10 $\quad\quad\quad = \{\mathsf{adSM}', \mathsf{adCM}', \mathsf{adSCWM}'\} \;\wedge$

11 $\quad\quad\quad \mathsf{afMgM}_\nu \lesssim \mathsf{adMgM} \;\wedge$

12 $\quad\quad\quad \forall \mathsf{avp} \in \mathsf{af.architectureViewpoints}.$

13 $\quad\quad\quad\quad \mathsf{avpMgM}_\nu \lesssim \mathsf{adMgM} \;\wedge$

14 $\quad\quad\quad\quad \{\mathsf{avDocM} :_{\bullet} \mathsf{TerminalModel}, \mathsf{avMgM} :_{\bullet} \mathsf{Megamodel}\} \subseteq \mathsf{R}_{\mathsf{views}}$

15 $\quad\quad\quad\quad \text{where } \mathsf{avDocM} \triangleleft \mathsf{ArchitectureViewDocMM}_\rho \;\wedge$

16 $\quad\quad\quad\quad\quad \mathsf{avMgM} \triangleleft \mathsf{MgMM}_\rho \;\wedge$

17 $\quad\quad\quad\quad\quad \nu(\{\mathsf{MMM}_\rho, \mathsf{MgMM}_\rho, \mathsf{ArchitectureViewDocMM}_\rho,$

18 $\quad\quad\quad\quad\quad\quad \mathsf{avDocM}, \mathsf{avpMgM}\}) \lesssim \mathsf{avMgM} \;\wedge$

19 $\quad\quad\quad\quad\quad \mathsf{avMgM}_\nu \lesssim \mathsf{adMgM}$

Table 4.13: *Semantic equations:* UseArchitectureFramework *statement.*

being design, denoted in the corresponding modeling artifacts adCM, adSM and adSCWM (9–10). The resulting terminal models adCM', adSM' and adSCWM' are used to replace the content of the original terminal models in R (7). Second, the set of modeling artifacts that participates in the denotation of af, formally $\mathcal{M}[\![\mathsf{af}]\!]$, is included in the resulting model repository (8). Third, the representative megamodel of af is captured in adMgM to state the fact that the architecture description ad being designed uses the architecture framework af. Also, every architecture viewpoint defined by af is set as an architecture viewpoint of ad (12–13). The correspondence rules defined by af are not captured by the megamodel adMgM of the architecture description. As we explained in Section §3.2.2, while the architecture viewpoints of the used architecture frameworks are directly captured by the architecture description, the correspondence rules defined by architecture frameworks are not. The architecture description captures those correspondence rules directly defined by the architecture description, and those that are selected to be applied. The set of correspondence rules that are available to be selected are those directly defined by the architecture description and also those defined by the architecture frameworks used. Consequently, the correspondence rules defined by af must not to be represented in adMgM.

Our architecture design scripting language does not provide a statement to create architecture views. As conceptualized by the ISO/IEC/IEEE 42010:2011 standard and illustrated in Figure 3.2, an architecture description contains one architecture view for each architecture viewpoint it uses. As a consequence, when using an architecture framework

and thus including additional architecture viewpoints to the architecture description, the corresponding architecture views must also be added. This is also the responsibility of the UseArchitectureFramework statement. Then, for each architecture viewpoint avp defined by af, an architecture view av is included in the architecture description ad. This fact is captured in the semantic equations in Table 4.13. For each architecture viewpoint avp (12), a terminal model avDocM (14) is created to capture the documentation of the corresponding architecture view, as well as a megamodel avMgM (14) to capture the set of modeling artifacts participating in the denotation of the architecture view (17–18). This representative megamodel avMgM is captured in the megamodel of the architecture description adMgM to represent the fact that the architecture view av is part of the architecture description ad. We call $\mathsf{R}_{\mathsf{views}}$ the set of artifacts of all the created views. The set $\mathsf{R}_{\mathsf{views}}$ is part of the repository resulting (8).

Create Statements. The CreateStatement allows the architect to create and add to the architecture description new architecture models and new correspondences, using the CreateArchitectureModel and CreateCorrespondence statements respectively. Figures 4.12 and 4.13 illustrate the conceptual model for the different kinds of create statements.

The first kind of CreateArchitectureModel statement allows the architect to create an empty architecture model. The CreateEmptyArchitectureModel statement belongs to class (B) as the architect must indicate which existing model kind governs the architecture model to create, and which existing architecture views aggregate it. No additional information needs to be provided by the architect. Table 4.14 defines the semantic equations for this statement. From the descriptive perspective, the denotation of $\mathsf{s} :_{\circ}$ CreateEmptyArchitectureModel results in the representative megamodels defined in the denotation of the related concept instances, particularly the model kind and every architecture view (3). From the behavioral perspective, the effect of s on a model repository R is to append to R the minimal set of modeling artifacts required to denote an architecture model am (8). In particular, the resulting model repository includes a documentation model amDocM which is initially empty and can be later improved using the corresponding ImproveDocumentation statement. It also includes a terminal model amM which purpose is to preserve the elements that describe the system from the perspective captured by the architecture model. This terminal model conforms to the metamodel mkMM_{ρ} defined by the governing modeling kind mk indicated by the statement s. Additionally, the resulting repository includes a megamodel amMgM that captures all the composing modeling artifacts in the denotation of am (13–14). This representative megamodel amMgM is captured in the megamodel avMgM of each architecture view that the statement s indicates as an aggregating architecture view (15). It is important to notice that this minimal set of modeling artifacts is derived from the denotation of ArchitectureModel defined in Table 3.11. Also, the semantic equation for $\mathcal{D}$ can only be applied when the indicated model kind and architecture views are already part of the architecture description being built (9). If this is not the case, the architect needs to previously include the corresponding model kinds and architecture views, mainly by means of the UseStatement that we discussed before.

The second kind of CreateArchitectureModel statement allows the architect to create an architecture model from a template. In this case, the statement belongs to class (C) as the architect needs to provide, in addition to the model kind and architecture views, which template to apply together with the parametric information to use when instantiating the template.

When we defined the semantic equations for $\mathcal{M}[\![\mathsf{ModelKind} : \circ]\!]$ in Table 3.10, we included in the denotation of each model kind mk a set of transformation models mktTM^j_ρ that capture the knowledge on how to create an initial version of the terminal model amM, that conforms to mkMM_ρ, for an architecture model am governed by mk. As defined in Table 3.10–(33), the transformation takes a terminal model providing information to the placeholders in the template, formally a terminal model conforming to $\mathsf{mktPlaceholderMM}^j_\rho$, and produces a terminal model conforming to mkMM_ρ. The CreateArchitectureModelFromTemplate statement is analogous to the CreateEmptyArchitectureModel statement, except that it uses this transformation model to create the initial terminal model amM instead of yielding an empty one. The semantic equations for this statement are analogous to those of CreateEmptyArchitectureModel defined in Table 4.14, except that $\mathcal{M}[\![\mathsf{s}]\!]$ also yields a terminal model $\mathsf{sPlaceholderM}^j$ that conforms to $\mathsf{mktPlaceholderMM}^j_\rho$, and that the effect of s yields a terminal model amM that is the result of applying the transformation operator $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{mktTM}^j_\rho, \langle \mathsf{sPlaceholderM}^j \rangle)$.

The CreateCorrespondence statement belongs to both classes (B) and (C), depending on whether the *correspondence kind* for the correspondence to be created is provided or not. Notice that as defined in the conceptual model in Figure 4.13, the multiplicity of the association from CreateCorrespondence and CorrespondenceKind is 0..1, i.e. it is optional. As we defined in the semantics for $\mathcal{M}[\![\mathsf{Correspondence} : \circ]\!]$ in Table 3.14, each correspondence c is denoted by a documentation model cDocM, a megamodel cMgM and a weaving model cWM that captures the links or relations between the architecture description elements that are defined in the woven models of cWM. This weaving model conforms to a customized metamodel cWMM_ρ that determines the information attached to the links in the correspondence. When a *correspondence kind* is indicated, this customized metamodel is defined in the denotation of the correspondence kind itself. When it is not indicated, however, the architect must provide it along with the create statement. Then, the structural denotation of $[\![\mathsf{s} :_\circ \mathsf{CreateCorrespondence}]\!]$ is the set of terminal models $\mathsf{m}_1, \ldots, \mathsf{m}_n$ that participate in the denotation of the architecture description elements of the correspondence to create, and also, either the representative megamodel of the correspondence kind or a metamodel for the weaving model. The effect of statement s is analogous to the effect of the CreateEmptyArchitectureModel statement that we defined in Table 4.14. In this case, the resulting repository also includes a documentation model and a megamodel, and includes an empty weaving model cWM conforming to the provided or already existing metamodel cWMM_ρ. Also, the representative megamodel cMgM is directly captured in the representative megamodel adMgM of the architecture description itself.

Improve Documentation Statements. The ImproveDocumentationStatement allows the architect to improve, refine and update the documentation of any of the complex work product that is directly defined by an architecture description. As we defined in the conceptual model in Figure 4.14, our architecture design scripting language provides one statement of this kind for updating the documentation of the architecture description itself, architecture views, architecture models, correspondences and correspondence rules. These statements belong to class (C) as the architect must provide, along with the statement, the details of the update to be done to the corresponding documentation. The semantic equations for all kinds of ImproveDocumentationStatement are practically the same. We define those corresponding to the ImproveDocumentationOfArchitectureView statement in Table 4.15. From the

1	$[\![\mathsf{CreateEmptyArchitectureModel} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{CreateEmptyArchitectureModel}]\!]$
3	with $\mathsf{mk} \equiv \mathsf{s.modelKind}, \mathsf{avs} \equiv \mathsf{s.architectureViews}$
4	$\stackrel{\text{def}}{=} \{\mathsf{mkMgM}\} \;\cup \bigcup_{\mathsf{av} \in \mathsf{avs}} \{\mathsf{avMgM}\}$
5	such that $\mathsf{mkMgM} \in \mathcal{M}[\![\mathsf{mk}]\!] \;\wedge\; \forall \mathsf{av} \in \mathsf{avs}.\; \mathsf{avMgM} \in \mathcal{M}[\![\mathsf{av}]\!]$
6	$[\![\mathsf{s} :_{\square} \mathsf{CreateEmptyArchitectureModel}]\!]\mathsf{R}$
7	with $\mathsf{mk} \equiv \mathsf{s.modelKind}, \mathsf{avs} \equiv \mathsf{s.architectureViews}$
8	$\stackrel{\text{def}}{=} \mathsf{R} \;\cup\; \{\mathsf{amDocM} :_{\bullet} \mathsf{TerminalModel}, \mathsf{amM} :_{\bullet} \mathsf{TerminalModel}, \mathsf{amMgM} :_{\bullet} \mathsf{Megamodel}\}$
9	for $\mathcal{M}[\![\mathsf{mk}]\!] \subset \mathsf{R} \;\wedge\; \forall \mathsf{av} \in \mathsf{avs}.\; \mathcal{M}[\![\mathsf{av}]\!] \subset \mathsf{R}$
10	such that $\mathsf{amDocM} \triangleleft \mathsf{ArchitectureModelDocMM}_{\rho} \;\wedge$
11	$\mathsf{amM} \triangleleft \mathsf{mkMM}_{\rho} \;\wedge$
12	$\mathsf{amMgM} \triangleleft \mathsf{MgMM}_{\rho} \;\wedge$
13	$\nu(\{\mathsf{MMM}_{\rho}, \mathsf{TMM}_{\rho}, \mathsf{ProblemMM}_{\rho}, \mathsf{ArchitectureModelDocMM}_{\rho},$
14	$\mathsf{amDocM}, \mathsf{amM}, \mathsf{mkMM}_{\rho}, \mathsf{mkMgM}\}) \lesssim \mathsf{amMgM} \;\wedge$
15	$\forall \mathsf{av} \in \mathsf{avs}.\; \mathsf{amMgM}_{\nu} \lesssim \mathsf{avMgM}$

Table 4.14: *Semantic equations:* CreateEmptyArchitectureModel *statement.*

descriptive perspective, their denotation yields the representative megamodel of the concept instance which documentation is to be improved. For instance, the megamodel avMgM that is representative in the denotation of the architecture view av (4). In the particular case of the ImproveDocumentationOfArchitectureDescription statement, no representative megamodel is needed as the documentation to be improved is that of the architecture description being built in the working model repository. Also, the denotation yields a transformation model sTM that encodes how to update the corresponding documentation model. In the case of architecture views, the transformation model sMT realizes a model transformation that takes a documentation model conforming to $\mathsf{ArchitectureViewDocMM}_{\rho}$ as an InOut parameter (5). Thus, from the behavioral perspective, the effect of s on a model repository R yields the same model repository where the content of the documentation model is replaced by the new content (9). To this end, the transformation operator $\mathcal{T}_{\mathsf{TMM}_{\rho}}(\mathsf{sMT})$ is applied on the documentation model to be updated, producing its new version which includes the corresponding updates (11).

Constrain Statements. A ConstrainStatement allows the architect to impose a constraint on the architecture description elements composing the architecture description. As we defined in Section §3.2.2, we consider two kinds of constraints: those that apply on elements within an architecture model, and those that apply on elements defined in different parts of the architecture description. While the former is called a *constraint*, the latter is actually called a *correspondence rule*. The ConstrainStatement can be classified whether it operates on a single architecture model, or it operates by means of correspondence rules. Figures 4.15 and 4.16 defined the conceptual model for each kind, respectively. As shown in these figures, in both groups we can further distinguish two categories of statements, one for creating a

1	$[\![\mathsf{ImproveDocumentationOfArchitectureView} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{ImproveDocumentationOfArchitectureView}]\!]$
3	with $\mathsf{av} \equiv \mathsf{s.architectureView}$
4	$\stackrel{\text{def}}{=} \{\mathsf{sTM} :_{\bullet} \mathsf{TransformationModel}, \mathsf{avMgM}\}$
5	such that $\mathsf{sTM} \triangleleft \mathsf{TMM}_{\rho} \quad \wedge \quad \mathsf{sTM} \blacktriangleright \mathsf{sMT} \left\langle \overline{\mathsf{ArchitectureViewDocMM}_{\rho}} \twoheadrightarrow \varnothing \right\rangle \quad \wedge$
6	$\mathsf{avMgM} \in \mathcal{M}[\![\mathsf{av}]\!]$
7	$[\![\mathsf{s} :_{\square} \mathsf{ImproveDocumentationOfArchitectureModel}]\!]\mathsf{R}$
8	with $\mathsf{av} \equiv \mathsf{s.architectureView}$
9	$\stackrel{\text{def}}{=} \mathsf{R}[\mathsf{avDocM}/\mathsf{avDocM}']$
10	for $\mathcal{M}[\![\mathsf{av}]\!] \subset \mathsf{R}$
11	such that $\mathcal{T}_{\mathsf{TMM}_{\rho}}(\mathsf{sTM}, \langle \mathsf{avDocM} \rangle) = \{\mathsf{avDocM}'\}$

Table 4.15: *Semantic equations:* ImproveDocumentationOfArchitectureView *statement.*

new element (either constraint or correspondence rule) and one selecting an existing one to be applied on particular elements.

From the descriptive perspective, statements in the *create* category belong to class (C) as the architect needs to provide specific modeling artifacts that represent the constraint to be created. For instance, in the case of CreateConstraint, in addition to indicating the architecture model am to constrain, the architect provides a transformation model amcTM that encapsulates how to process the terminal model amM in the denotation of the architecture model am, and how to produce a problems model, i.e. a terminal model one conforming to the $\mathsf{ProblemMM}_{\rho}$ metamodel defined in the infrastructure of an architecture description. This kind of transformation model is part of the semantic denotation of an architecture model, as we defined in Table 3.11–(21). In the case of a correspondence rule, the architect needs to provide a similar transformation model, but, in this case, it is defined in terms of the metamodels of the terminal models capturing the elements the correspondence rule constrains. This kind of transformation model is part of the semantic denotation of correspondence rules, as we defined in Table 3.15–(18). From the behavioral perspective, the effect of a statement s in the *create* category is to append the corresponding modeling artifacts to the model repository and to update the corresponding megamodels to consider the new constraint. For instance, the CreateConstraint statement adds the transformation model to the model repository and adds it to the representative megamodel of the architecture model being constrained. Formally, we have that $\mathsf{amcTM} :_{\bullet} \mathsf{TransformationModel} \in \mathsf{R}'$ and that $\mathsf{amcTM}_{\mu} \lesssim \mathsf{amMgM}$. In the case of the CreateCorrespondenceRule statement, as it is a complex work product, a documentation model and a representative megamodel must be added to the repository, in addition to the transformation model. Also, the representative megamodel is registered in the representative megamodel of the architecture description.

From the descriptive perspective, statements in the *select* category belong to class (B) as the architect only needs to provide on which elements to operate on. In the case of the SelectConstraint statement, the architect indicates the model kind defining the constraint and the architecture model to be constrained. In the case of the SelectCorrespondenceRule

1	$[\![\mathsf{SelectCorrespondenceRule} : \circ]\!] \stackrel{\mathsf{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{SelectCorrespondenceRule}]\!]$
3	with $\mathsf{cr} \equiv \mathsf{s.correspondenceRule}, \mathsf{es} \equiv \mathsf{s.elements}$
4	$\stackrel{\mathsf{def}}{=} \{\mathsf{crMgM}\} \cup \bigcup_{\mathsf{e_i} \in \mathsf{es}} \{\mathsf{eM_i}\}$
5	such that $\mathsf{crMgM} \in \mathcal{M}[\![\mathsf{cr}]\!]$
6	$\forall \mathsf{e_i} \in \mathsf{es}.\ \mathsf{eM_i} :_{\bullet} \mathsf{TerminalModel} \wedge \mathsf{eM_i} \in \mathcal{M}[\![\mathsf{e_i}]\!]$
7	$[\![\mathsf{s} :_{\square} \mathsf{SelectCorrespondenceRule}]\!]\mathsf{R}$
8	with $\mathsf{cr} \equiv \mathsf{s.correspondenceRulees} \equiv \mathsf{s.elements}$
9	$\stackrel{\mathsf{def}}{=} \mathsf{R} \cup \{\mathsf{scrMgM} :_{\bullet} \mathsf{Megamodel}\}$
10	for $\mathsf{crTM}_{\rho} \in \mathcal{M}[\![\mathsf{cr}]\!] \wedge \mathsf{crTM}_{\rho} \blacktriangleright \mathsf{crMT}\langle \mathsf{MM}_1, \ldots, \mathsf{MM}_n \twoheadrightarrow \mathsf{ProblemMM}_{\rho}\rangle \wedge$
11	$\forall \mathrm{i}, 1 \leq \mathrm{i} \leq n.\ \mathsf{eM_i} :_{\bullet} \mathsf{TerminalModel} \wedge \mathsf{eM_i} \in \mathcal{M}[\![\mathsf{e_i}]\!] \wedge \mathsf{eM_i} \triangleleft \mathsf{MM_i}$
12	where $\mathsf{e_i} \equiv \mathsf{es} \rightarrow \mathsf{at}(\mathrm{i})$
13	such that $\mathsf{scrMgM} \triangleleft \mathsf{MgMM}_{\rho} \wedge$
14	$\mathsf{crMgM}_{\nu} \lesssim \mathsf{scrMgM} \wedge$
15	$\forall \mathsf{e} \in \mathsf{es}.\ \mathsf{eM}_{\nu} \lesssim \mathsf{scrMgM} \wedge$
16	$\mathsf{scrMgM}_{\nu} \lesssim \mathsf{adMgM}$

Table 4.16: *Semantic equations:* SelectCorrespondenceRule *statement.*

statement, the architect indicates the correspondence rule to apply and the set of elements on which it is to be applied. Table 4.16 defines the semantic equations for the SelectCorrespondenceRule statement. Formally, the denotation of an statement s consists of the representative megamodel crMgM of the selected correspondence rule, and the terminal models $\mathsf{eM_i}$ belonging to the denotation of the elements indicated in s.elements (4). From the behavioral perspective, the effect of a statement s is to update the corresponding megamodels to reflect the fact that the constraint is selected. In the case of SelectConstraint statement, this is analogous to the CreateConstraint statement except that the transformation model to be registered in the representative megamodel amMgM of the architecture model being constrained is already defined in the indicated model kind, instead of being directly provided by the architect. In the case of the SelectCorrespondenceRule statement, a new megamodel scrMgM (9) is required in order to capture the correspondence rule being applied (14) and the terminal models on which the correspondence rule is applied (15). Notice that this new megamodel scrMgM, that is appended to the model repository R (9) and represented in the megamodel adMgM of the architecture description (16), is actually the way that a *selected correspondence rule* is captured in the denotational semantics of architecture descriptions, as we defined in Table 3.16–(28–32).

Apply Statements. The ApplyStatement allows the architect to use or apply an architecture design mechanism on the architecture description being built, particularly architecture styles, patterns and tactics. These mechanisms can be defined in an already existing architecture design library or they can be directly defined by the architect in the context of a particular design effort.

As we defined in Section §4.1.2, there are three different kinds of ApplyStatement: ApplyStyle, ApplyPattern and ApplyTactic. The ApplyStyle statement allows the architect to include in the architecture description a new system-specific model kind by extending already defined model kinds aggregated in the architecture viewpoints in use in the architecture description. This statement is not meant to include already existing model kinds, which is actually the purpose of the UseStatement for including model kinds defined in the context of architecture frameworks or architecture description languages. Rather, the ApplyStyle statement allows the architect to create a new model kind, thus providing a system-specific modeling language to be later used by some architecture models in the architecture description. The conceptual model for this statement is defined in Figure 4.17. From a descriptive perspective, the ApplyStyle statement belongs to class (C). The architect not only has to indicate which are the already existing model kinds to extend and the already existing architecture viewpoints to aggregate the new model kind, but also the set of modeling artifacts that denote this new model kind. Then, the structural denotation of $\mathsf{s} :_{\circ}$ ApplyStyle yields the representative megamodels of the extended model kinds $\mathsf{mkMgM}_\mathsf{i}$, the representative megamodels avpMgM_j of the architecture viewpoints that aggregate the new model kind, and the set of modeling artifacts defined in Table 3.10–(14). Particularly, a documentation model mkDocM, a metamodel mkMM defining the modeling language by extending the metamodels mkMM_i of the model kinds mk_i indicated in s, and a representative megamodel mkMgM formed by the set of all artifacts in the definition of the new model kind mk. In addition, template transformation models and constraint transformation models can be provided by the architect. From the behavioral perspective, the effect of $\mathsf{s} :_{\Box}$ ApplyStyle is to include the defined modeling artifacts in the resulting model repository, and registering the representative megamodel mkMgM of the new model kind in the representative megamodel avpMgM_j of the architecture viewpoints indicated by s. This effect is similar to that of the UseArchitectureFramework statement that we defined in Table 4.13. It is important to remark that the effect of a ApplyStyle statement only affects the available constructs to use, while all architecture models and correspondences remain unchanged.

As opposed to ApplyStyle, the ApplyPattern and ApplyTactic statements are meant to update the architecture models and the correspondences so as to reflect in them how to address a particular set of concerns or requirements. The conceptual model for these statements is defined in Figures 4.18 and 4.19 respectively. As we discussed in Section §4.1.1, patterns and tactics captures system-independent parameterized solutions that, to be applied, the architect must provide the corresponding configuration for those parameters reflecting the architect's intention and the specifics of the architecture being designed. As we discussed before and defined in Table 4.5 for architecture patterns, the most significant modeling artifact in the structural denotation of a pattern – and analogously of a tactic – is a transformation model that encapsulates the knowledge on how to update terminal models conforming to specific metamodels defined in the denotation of either the model kinds or the correspondences that the architecture pattern operates on. While the pattern is conceived in terms of the metamodels, the application of the pattern is conceived in terms of the actual terminal models on which the pattern must be applied. This is captured by the structural denotation of $\mathsf{s} :_{\circ}$ ArchitecturePattern defined in Table 4.17–(2). The definition for ApplyTactic statement is analogous, even simpler, as a tactic only operates on model kinds. As specified by the semantic equation in Table 4.17, $\mathcal{M}[\![\mathsf{s}]\!]$ yields the representative megamodel of the pattern being applied (4), but also the representative megamodel of the architecture models amMgM

1	$[\![\mathsf{ApplyPattern} : \circ]\!] \stackrel{\text{def}}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{ApplyPattern}]\!]$
3	with $\mathsf{ap} \equiv \mathsf{s.architecturePattern}, \mathsf{ams} \equiv \mathsf{s.architectureModels}, \mathsf{cs} \equiv \mathsf{s.correspondences}$
4	$\stackrel{\text{def}}{=} \{\mathsf{apMgM}\} \cup \bigcup_{\mathsf{am} \in \mathsf{ams}} \{\mathsf{amMgM}\} \cup \bigcup_{\mathsf{c} \in \mathsf{cs}} \{\mathsf{cMgM}\} \cup \{\mathsf{sParamM} :_{\bullet} \mathsf{TerminalModel}\}$
5	such that $\mathsf{apMgM} \in \mathcal{M}[\![\mathsf{ap}]\!] \wedge$
6	$\forall \mathsf{am} \in \mathsf{ams}.\ \mathsf{amMgM} \in \mathcal{M}[\![\mathsf{am}]\!] \wedge$
7	$\forall \mathsf{c} \in \mathsf{cs}.\ \mathsf{cMgM} \in \mathcal{M}[\![\mathsf{c}]\!] \wedge$
8	$\mathsf{sParamM} \lhd \mathsf{apParamMM}_\rho$
9	$[\![\mathsf{s} :_{\square} \mathsf{ApplyPattern}]\!]\mathsf{R}$
10	with $\mathsf{ap} \equiv \mathsf{s.architecturePattern}, \mathsf{ams} \equiv \mathsf{s.architectureModels}, \mathsf{cs} \equiv \mathsf{s.correspondences}$
11	$\stackrel{\text{def}}{=} \mathsf{R}''$
12	for $\mathsf{apTM}_\rho \in \mathcal{M}[\![\mathsf{ap}]\!] \wedge \mathsf{apTM}_\rho \blacktriangleright \mathsf{apT}\left\langle \overline{\mathsf{MM}_1}, \ldots, \overline{\mathsf{MM}_n}, \mathsf{apParamMM}_\rho \twoheadrightarrow \varnothing \right\rangle \wedge$
13	$\langle \mathsf{m}_1, \ldots, \mathsf{m}_n \rangle$ the tuple of the terminal models $\mathsf{amM_i}$ or $\mathsf{cWM_i}$ according to the
14	order in the definition of apT, where $\mathsf{m_i} \lhd \mathsf{MM_i}$ and
15	either $\mathsf{amM_i} \in \mathcal{M}[\![\mathsf{am_i}]\!]$ or $\mathsf{cWM_i} \in \mathcal{M}[\![\mathsf{c_i}]\!]$
16	such that $\mathcal{T}(\mathsf{apTM}_\rho, \langle \mathsf{m}_1, \ldots, \mathsf{m}_n, \mathsf{sParamM} \rangle) = \{\mathsf{m}'_1, \ldots, \mathsf{m}'_n\} \wedge$
17	$\mathsf{R}' = \mathsf{R}[\mathsf{m}_1/\mathsf{m}'_1, \ldots, \mathsf{m}_n/\mathsf{m}'_n] \wedge$
18	$\mathsf{R}'' = [\![\mathsf{scr}_q]\!]_\square(\ldots([\![\mathsf{scr}_1]\!]_\square(\mathsf{R}'))\ldots)$
19	for $\mathsf{scr}_j \in \mathsf{s.selectCorrespondenceRules}$

Table 4.17: *Semantic equations:* ApplyPattern *statement.*

and correspondences cMgM on which the pattern is applied (4). However, from the descriptive perspective, statement s actually belongs to class (C), i.e. it requires the architect to provide statement-specific modeling artifacts in the definition of the statement. The main reason is that architecture patterns are not only parameterized in the actual architecture models and correspondences on which they can be applied, but also in the specifics of the application of the pattern, captured by the parameters metamodel $\mathsf{apParamMM}_\rho$ defined in Table 4.5–(22). Then, the architect must also provide a terminal model sParamM (4) conforming to this metamodel (8) as defined in Table 4.17.

From the behavioral perspective, the effect of $\mathsf{s} :_{\square} \mathsf{ApplyPattern}$ on a model repository R is defined in two stages. First, the transformation operator $\mathcal{T}_{\mathsf{TMM}_\rho}(\mathsf{apTM}_\rho)$ is applied on the terminal models $\mathsf{m}_1, \ldots, \mathsf{m}_n$ and the terminal model sParamM, obtaining a new version of the InOut terminal models: $\mathsf{m}'_1, \ldots, \mathsf{m}'_n$ (16). Then, the model repository R is updated by replacing the content of these models with the content of their new version (17) obtaining an intermediate model repository R'. Second, the correspondence rules selected by the pattern's application must be reflected in R'. As defined in the conceptual model for ApplyPattern in Figure 4.18, the statement s indicates a possibly empty sequence of correspondence rules that must be applied on the architecture models and correspondences operated by the application of the pattern. Then, the set of SelectCorrespondenceRule statements in s.selectedCorrespondenceRules is sequentially applied on R' (18–19) according to the behavioral semantics that we defined in

1	$[\![\mathsf{CallScript} : \circ]\!] \stackrel{def}{=} \varnothing$
2	$[\![\mathsf{s} :_{\circ} \mathsf{CallScript}]\!]$
3	with $\mathsf{us} \equiv \mathsf{s.script}$
4	$\stackrel{def}{=} \{\mathsf{usMgM}\}$ such that $\mathsf{usMgM} \in \mathcal{M}[\![\mathsf{us}]\!]$
5	$[\![\mathsf{s} :_{\square} \mathsf{CallScript}]\!]\ \mathsf{R} \stackrel{def}{=} [\![\mathsf{s.script}]\!]_{\square}\ \mathsf{R}$

Table 4.18: *Semantic equations:* CallScript *statement.*

Table 4.16–(7). Finally, the resulting model repository R'' is the resulting model repository of the statement s (11).

Call Script Statement. The CallScript statement allows the architect to reuse architecture update scripts. While the script attached to an architecture solution provides the complete implementation of the architect's intention represented by the solution, smaller and simpler scripts can be defined to accomplish less ambitious goals. By means of the call script statement, the architect can factorize larger scripts into smaller ones, and can reuse them as many times as appropriate. Sequential execution (embedded in the definition of script) and script calls are the only control statements that we include in our architecture design scripting language. Additional control statements like conditionals and loops are proposed as future work. The CallScript statement belongs to class (B) as the architect needs to provide only the script to be called. Table 4.18 defines the semantic equations for this statement. From the descriptive perspective, the denotation of $\mathsf{s} :_{\circ} \mathsf{CallScript}$ results in the representative megamodel of the called script (4). From the behavioral perspective, the effect of s on a model repository R is actually the effect of the called script on R (5). It is important to remark that this simple and direct definition for the effect of the CallScript statement is possible as the statement does not support passing parameters and the language does not provide the capability of defining local variables. Hence, there is no need to preserve the contextual state of the execution of a script beyond the actual model repository being operated on. The CallScript statement actually works like macro-substitution is programming language theory.

Architecture Design

In the introduction to Section §4.1 we discussed iterative and incremental approaches to architecture design consisting of three major activities that takes place iteratively, involving to understand the problem, to solve it, and to evaluate the solution. We illustrated this process in Figure 4.2. We identified the Decision Making activity – to solve the problem – as the core of the process. This activity is enacted by the architect or architect team, who devises and weights alternative solutions to address a significant set of concerns and requirements, decides the solution that provides best system-wide long-term benefits, and updates the architecture description to reflect the decision made. The iterative nature of the process renders the architecture description to be built incrementally. Starting from an initial possibly empty architecture description, it is successively updated and refined, decision after decision, until the the architect and the significant stakeholders agrees that

the designed architecture achieves the expected functionality and provides the best quality, constrained by the characteristics of the development effort, such as time, budget, skills, tools, etc. We illustrated this incremental nature of architecture descriptions in Figure 4.3, where a *sequence* of architecture descriptions is produced by successive decisions. As we discussed before, architecture design is rarely so lineal. Architecture design progresses by trial-and-error, going forward and backward while exploring the alternative solutions devised by the architect. While in some case to explore a potential solution involves simply sketches, in some other cases the architecture description is actually updated so as the alternative can be presented to significant stakeholders and discussed appropriately. We illustrated the exploration of the solution space in Figure 4.4, where a *tree* of architecture descriptions is produced by making different decisions. In the figure, while each rectangle represent a version of the architecture description, each arrow represent a different decision that yields to different architecture descriptions.

In this section we introduced our model-based approach to architecture design. The core principle of our approach is to shift the architect's focus from directly building the architecture description, to capture how the architecture description is built. We called our approach $\underline{\text{why}} + \underline{\text{how}} \Rightarrow \underline{\text{what}}$ as the architect's focus is on capturing why the architecture description is built in the way it is, along with how the architecture description must be built, rendering the what – the architecture description – to be the result of exercising the how. As we discussed before, it is by capturing the how formally that we achieve automation in the construction of the what. The formal specification that we defined in this section provides this capability. We use modeling constructs and techniques not only to capture architecture design concepts, but only to enact them. While the semantic function $\mathcal{M}$ formally defines how every concept and concept instance is denoted or representing in terms of modeling artifacts, the semantic function $\mathcal{D}$ formally defines how to enact the architecture decisions in terms of modeling techniques, i.e. in terms of the semantic operators available in the modeling environment, such as model repository manipulation and the execution of model transformations.

Let $\mathsf{d}_1, \ldots, \mathsf{d}_n$ be a sequence of architecture decisions captured according to the conceptualization we defined in Section §4.1.3. Thus, each $\mathsf{d}_\mathsf{i} :_\circ \mathsf{ArchitectureDecision}$ identifies a selected architecture solution that provides and architecture update script representing how the architecture description must be updated. Then, starting from an empty model repository $\mathsf{R}_0 \equiv \varnothing$, conforming the initial version of the architecture description, the successive application of $\mathcal{D}[\![]\!]\mathrm{d}_\mathrm{i}$ captures the iterative nature of the architecture design process. Each d_i is applied on the current version of the architecture description built so far, represented by $\mathsf{R}_{\mathsf{i}-1}$, and produces a new current version R_i. The succession of R_i conforms the incremental nature of the architecture description being designed. The resulting model repository R_n yields the final and complete architecture description. Formally:

$$\mathsf{R}_n = ([\![\mathsf{d}_n]\!]_\square \circ \ldots \circ [\![\mathsf{d}_1]\!]_\square)\varnothing$$

It is important to remark that any partial sequence $\mathsf{d}_1, \ldots, \mathsf{d}_k$ renders the architecture description R_k conforming the result of the k-th iteration. This architecture description can be checked for inconsistencies by applying the corresponding transformation models that represent constraints in the architecture models or that represent correspondence rules.

These transformations generate *problems* terminal models that containing the issues found. Besides, each intermediate architecture description is analyzed and evaluated by the architect team and the significant stakeholders. Also, in the next iteration $k + 1$, the architect takes into account the results of the stakeholders' evaluation, the inconsistencies found, a set of concerns or requirements pending to be addressed, and the architecture description R_k itself to devise solutions and to make the next decision d_{k+1}.

As we mentioned above, when the architect explores alternative solutions, different decisions are made and hence different architecture descriptions are obtained. Thus, the design effort is not simply a succession of architecture decisions, but a tree of decisions where a branch conforms the actual architecture description built. Each edge of that tree is represented by a decision d_i, and each branch of that tree, formed by a sequence of decisions, renders an architecture description.

Consequently, by capturing architecture decisions, solutions and scripts – the why + how as formally specified by the semantic function $\mathcal{M}$, the application of the semantic function $\mathcal{D}$ produces the architecture description of the architecture designed – the what. Automation is achieved in the sense that the captured architecture decisions, solutions and scripts relies on modeling techniques to produce the architecture description. By applying the corresponding model transformations, the resulting model repository contains the architecture description built. Further automation can be achieved by means of an interpreter of the architecture description scripting language. Such an interpreter would take the modeling artifacts corresponding to $\mathcal{M}[\![\mathsf{d}]\!]$, interpret the architecture update script defined by the selected solution of d, and apply the corresponding model transformations, and thus generate the architecture description. The semantic function $\mathcal{D}$ provides the formal specification of such an interpreter, and modeling techniques such as model repository management and transformation engines provides the building blocks for its implementation.

4.2.4 Traceability

Traceability information is preserved or must be derived from the architecture description itself. It is the responsibility of the architect, or the supporting tool, to include the minimal information that are enough to answer the traceability questions that are expected in the development project. There are several levels of traceability, and some of them are already covered by the conceptual model for architecture descriptions.

The relationship between stakeholders and concerns allows the architect to determine the coverage on the architecture description that a particular stakeholder or stakeholder category has. As the standard uses concerns to characterize architecture viewpoints and model kinds, this relationship captures the coverage of a stakeholder, at least at a coarse-grained level of abstraction. For instance, given that an architecture view is governed by a single architecture viewpoint, and an architecture model is governed by a single model kind, the traceability from concerns to these kind of elements is derived by transitivity. From the perspective of our model-based interpretation, we pay special attention not only to capture the relation from the corresponding concepts to concerns, but also, to include any possible knowledge already captured in architecture frameworks and ADLs in the architecture description when

the architect decides to adhere to them. To this end, the UseStatement statements merges the concerns, stakeholders and their interrelationships into the models being captured in the architecture description.

A fine-grained traceability between architecture description elements is captured by means of correspondences and enforced by means of the selected correspondence rules. With them, the architect capture relations between elements, either by extension or by intention. Our model-based interpretation also covers these constructs, provides automation to the validation of the selected correspondence rules (the selection is achieved by constrain statements), and the validation by means of the transformation to Problem defined by correspondence rules. At the level of a single architecture model, selectable constraints serves a similar purpose.

In the context of architecture design, we follow a similar approach to allow the characterization of the elements in terms of concerns. Thus, patterns, tactics, styles, solutions and decisions are categorized in terms of concerns, and our model based interpretation also captures it. Also, the semantic equations for $\mathcal{D}$ also merges the corresponding concerns into the concerns identified in the architecture description. By this means, we can trace from any given concern, to the decisions that deals with it or that raises it, the solutions that involves it, and the used architecture design mechanisms (patterns and tactics) that affects it.

A fine-grained traceability requirement is posed on architecture decisions. As stated in the standard, and illustrated in Figure 4.20, it is important to capture which are the architecture description elements that were affected by making each particular decision. To capture this in practice is hard as it requires a significant effort, and a manual approach is generally error prone or tend to be incomplete or out of date. Our model based interpretation automates this task. In our conceptualization, the architect focus on capturing the modifications to be performed on the architecture, instead of capturing its effect first-hand. The architecture description is automatically updated (by means of the modeling environment tool support - i.e. the reification of the operators in the semantic domain) by providing the update script. Thus, not only reusability is improved as the same script can be applied in multiple scenarios instead of manually repeating the required steps, but also traceability. Let us explain this in what follows.

As we explained in Section §3.2.2 when we studied the concept Architecture Description Element when introducing the semantics for correspondences, ADEs can refer to any element of the architecture description, either a concept instance of the conceptual model, or an element defined in an architecture model which type is introduced by viewpoints and model kinds. Then, there are two levels, modeling artifacts and model elements within modeling artifacts. As we analyzed before, our model-based interpretation also represent modeling artifacts in the repository as model elements of the corresponding megamodels. Hence, it would be enough to consider the case for model elements. However, it is useful to study them separately. In the case of concept instances of the conceptual model that correspond to modeling artifacts, the definition of the decision, its solution and the scripts are expressed in terms of them. In other words, our conceptual model for design is combined with the conceptual model for description, and hence, we know which artifacts are referred by definition. However, concepts are complex work products and generally involved several modeling artifacts. The denotation of these concept instances using $\mathcal{M}$ defines which are the actual

modeling artifacts affected. Additionally, we can use the modeling environment to detect the affected artifacts. For instance, having $\mathcal{D}[\![\mathsf{u}]\!]\mathsf{R}_i = \mathsf{R}_{i+1}$, we have that the difference $\mathsf{R}_{i+1} - \mathsf{R}_i$ defines the set of modeling artifacts that were created. Clearly, all model elements defined within these modeling artifacts were also created. Also, for every $m_i \in \mathsf{R}_i$ with its corresponding new version $m_{i+1} \in \mathsf{R}_{i+1}$, $m_{i+1} - m_i$ (the model difference) determine the elements that were created, deleted or modified (their properties were changed). The scenario for deletion usually occur when the elements have no identifier or its identifier changes, which appears as a deletion and a creation. Also, as m_{i+1} was obtained by means of a model transformation on a given set of models (not necessarily m_i), the traceability information produced by $\mathcal{T}$ also determines the relationship between modeling artifacts and model elements in R_i and R_{i+1}. From the technological point of view, this traceability is a weaving model that preserve links between elements in the input models to the impacted elements in the output models.

4.3 Contributions & Discussion

In this chapter we addressed the problem of the lack of a homogeneous means to capture and capitalize architecture knowledge on architecture design. Thus, the goal and main contribution of this chapter is the definition of such a homogeneous means, making such knowledge shareable, reusable, tool-friendly and directly applicable.

As opposed to architecture description, the community lacks a contextual model of the practice providing the main concepts and their relation to their real-world context. To this end, we have extended the contextual model defined in the ISO/IEC/IEEE 42010:2011 standard [ISO11] to consider the architecture design practice. Succinctly, an architect follows an architecture design process to design the architecture of a system-of-interest, and to construct and update its representation captured in an architecture description. While there is no consensus on a single architecture design process, the community agree of three coarse-grained activities involving to understand the problem, to solve it, and to evaluate the solution. Our position is that while the architect's tasks required by these coarse-grained activities may vary from one architecture design process to another, there exists an underlying set of fine-grained activities or architecture design steps that the architect performs when actually building the architecture description of the architecture being design. In fact, in this chapter we developed a scripting language for architecture design. We defined a conceptual model including constructs for every update that can be performed on an architecture description. Such statements are not simple create-update-delete actions on architecture description constructs, but rather they are at the level of abstraction of the architecture design practice. We included statements for starting an architecture description from different sources, for reusing architecture description knowledge by adhering to architecture frameworks and architecture description languages, for capturing additional information and improving the documentation of elements, and mainly for specializing architecture styles and applying architecture patterns and tactics. Also, inspired on current research on the discipline, we reviewed and conceptualized architecture patterns and tactics in terms of the conceptual model for architecture description practice, making them system-independent reusable assets that can be applied in different architecting scenarios.

The purpose and usefulness of this conceptualization goes beyond capturing and describing the architect's steps towards the design of an architecture description. The community recently underwent a paradigm-shift towards capturing architecture decisions within the architecture description. In the current state of the art, in addition to the set of interrelated architecture description elements shaping the architecture, methods and techniques are available to also capture why the architecture is shaped in that specific way. Architecture decisions and rationale are now explicit in architecture descriptions. In this chapter we referred to this approach as why + what. We claim that the architect's knowledge on *how* the architecture is actually built is lost during architecture design. Thus, we proposed a shift in the architect's focus from building the architecture description, to capture the architecture solutions that render the architecture description. In this chapter we called our approach why+how ⇒ what. Thus, our architecture design approach makes architecture design explicit rendering architecture description implicit and automatically derived. Consequently, for us, the purpose of architecture decisions and architecture solutions is prescriptive as they express the intention of the architect for improving the architecture description built so far, and applicable as they serve the architect as a tool for actually creating the next version of the architecture description from the current version. Thus, our fine-grained architecture design scripting language provides the foundation for the enactment of the coarse-grained activities of architecture design processes, mainly in the context of the Decision Making stage.

In order to reify our approach and to provide a homogeneous means to capture architecture design knowledge, we defined a model-based interpretation of the conceptual model, covering the fine-grained update statements, architecture decisions and solutions, and reusable system-independent architecture patterns and tactics. We extend our denotational semantics for architecture description to deal also with architecture design concepts. Hence, by means of the conceptualization and its formalization, we provide a first-class representation of architecture decisions and solutions making them prescriptive and applicable, which conforms one of the core contributions of this work.

Reuse of Architecture Solutions & Decisions. Our model-based approach to architecture design requires the architect to explicitly capture architecture decisions and architecture solutions in a way that the architecture description is automatically generated. We defined an *architecture update script* as a sequence of modifications to an architecture description, namely *architecture update statements*, that allows the architect to encapsulate and modularize fine-grained modifications that must occur together. We defined an *architecture solution* as an architecture design mechanism that prescribe and describe the structure and behavior that must be included in the architecture description to address a particular set of concerns. A solution is reified by means of an architecture update script that captures the actual modifications needed. We defined an *architecture decision* as an aggregation of alternative solutions, where one of them is the actual solution decided. A decision is justified by a rationale and is reached by weighting the benefits and liabilities of the alternative solutions. Scripts capture *how* the architecture description is updated, solutions give them purpose, decisions capture trade-offs, selection and justification.

Scripts are mainly a modularization technique for updates, allowing the architect to structure the captured modifications and use them in more than one scenario. The architect can

progressively restructure scripts and updates into smaller scripts when an opportunity for reuse is faced. This provides a productivity improvement as by reusing scripts, the architect requires less effort for producing alternative solutions when exploring the solution space. However, reuse of scripts is localized to the architecture being design. From a structural perspective, scripts expect the architecture description to be shaped in a particular way — e.g. certain architecture model must exist, certain architecture description elements must be populating some architecture models, etc.—. However, scripts do not depend on the whole set of modeling artifacts and model elements populating those artifacts, just on some of them. Then, while the expected modeling artifacts are available in the model repository and the expected model elements are defined in those artifacts, the script can be applied. This is actually the single kind of precondition for applying scripts. From the intention perspective, scripts are mostly system-dependent as they update the architecture description by including system-specific information — such as concerns, stakeholders, descriptions, etc.— or by applying patterns and tactics with the system-specific parameterization. Even though it might be possible, particularly for small scripts, we do not expect system-independent reuse for them. We delegate the responsibility of system-independent reuse to architecture patterns and tactics. In our approach, every modification to architecture models and correspondences are performed by means of patterns or tactics, even though the problem being solved is not actually a recurrent one. Thus, the architect first captures the intended modification in a system-independent manner, such as a pattern or tactic — that are reified by model transformations —, and second, the architect captures the script by which that pattern or tactic is applied using the system-specific information — reified by the application of model transformations.

Architecture solutions are also system-dependent as they rely on scripts to express the intended modifications. Architecture solutions can be reused when the same problem need to be addressed more than once, on architecture descriptions that are similarly structured and populated. The latter condition is actually imposed by the solution's script, if the script cannot be applied in a new scenario, the solution cannot be reused there either. The former condition, however, refers to the fact that solutions are not just modifications, they are intended to address a problem, presenting pros and cons while doing it. One might argue that the same specific problem is not solved twice in the same architecture design effort, however this is not actually the case. When exploring alternative solutions, the same problem may and surely does arise in more than one branch of the tree resulting from the exploration of the solution space. Thus, the same solution may be used in every branch where the problem is faced and where the script is applicable.

In the context of forward architecture design of a single system — i.e. the recursive refinement of an initial architecture towards a satisfying architecture — architecture decisions are hard to reuse. To achieve so, the same problem must be faced more than once, the same solutions must be applicable and the same tradeoffs and selection must be accurate. This scenario may be only possible for the final decisions, not for the initial ones. As architecture decisions actually conform the nodes of the tree of alternative solutions built during architecture design, to reuse an architecture decision implies to reuse the whole sub-tree, from the decision to the leaf nodes — i.e. the final decisions. Hence, a single decision cannot be reused, but rather it must be reused together with all subsequent decisions made.

Consequently in forward architecture design of a single system, while the reuse of patterns, tactics and scripts, provides a productivity improvement with respect to no reuse, reuse of architecture decisions is rarely possible and reuse of architecture solutions is mainly opportunistic. Being opportunistic, reuse depends on the ability of the architect to classify, catalogue and search existing solutions when exploring alternatives. Concerns can be used for cataloguing and look up. However, the large number of concerns and solutions tend to make it ineffective. In practice, organizing architecture solutions is not the primary focus of the architect, and hence, opportunities for reuse might be missed. While the architect intentionally modularizes scripts to favor reuse of the small ones, the architect cannot straightforwardly achieve the same for solutions and decisions. Then, the main obstacle for effective reuse is the fact that reuse is not being planned. This is the main focus of Chapter §5 when we use a planned reuse approach to architecture design.

Architecture Design Scenarios. Architecture design can take place in different scenarios in which an architect or team of architects, together with significant stakeholders, rely on software architecture principles and techniques to address a specific goal by means of designing a software architecture to deal with size and complexity of the endeavor. In this chapter we have discussed the *forward architecture design* scenario where the architect designs the architecture of a system of interest in order to meet the stakeholder expectations. We defined a model-based approach to architecture design in this scenario, allowing the architect to use different starting points for the architecture design effort, such as from scratch, from an existing architecture description, or from a reference architecture. As we discussed before, while reuse of architecture decisions and solutions is possible, it is mainly opportunistic. To overcome this limitation, in Section §5 we thoroughly discuss and address the scenario promoting *planned reuse* which can benefit from the explicit representation of architecture decisions and solutions, along with their reification in terms of architecture update scripts.

A recurrent architecture design scenario is when architecture design is an after-the-fact endeavor. In this case, stakeholders have a system of interest that is already implemented and possibly available in a production environment. This is the case for legacy systems or for recent systems that have suffered significant changes in its implementation and configuration, and the drift from the actual artifacts with respect to the architecture description of the system, if it exists, is considerable. This scenario is known as *architecture reconstruction* or *architecture discovery*, where the architecture description of an existing system is extracted from the set of artifacts available, usually the source code, build scripts and occasionally designs that are directly realized as code. Architecture reconstruction involves the extraction of information from the available resources, the categorization and refinement of the extracted information, and its consolidation as an architecture description. As noted in [KOV03], tool support is crucial to make architecture reconstruction feasible. Our model-based approach to architecture design can partially assist in this process, but it is not aimed to this particular scenario. As our approach is based on the ISO 42010 standard for architecture description, our conceptualization the architecture description practice, and mainly our formalization in terms of modeling artifacts can guide practitioners on the set of resulting artifacts that can be produced by tools enacting the reconstruction. For instance, the architect can use the structural statements of our architecture design scripting language to produce the initial architecture description. Particularly, using any StartStatement allows the architect to set

the initial set of modeling artifacts, using a UseStatement allows the architect to include in the architecture description the architecture frameworks and architecture description languages that are expected in the reconstructed architecture description. Also, by means of CreateStatements particular architecture models and correspondences can be added to the description, and by means of ConstrainStatements expected constraints can be enforced on the description, which are later automatically evaluated by the modeling environment on any state of the architecture description being reconstructed. Finally, ApplyStatements can assist the architect to encode the refinement of the reconstructed models, once the core information has been injected into the architecture models by means of external tools. Besides, as our approach relies on modeling techniques, they can be used to assist the processing of the resource artifacts and the extraction of information from them. Particularly, this is the main goal of the MoDisco Project [Ecl14c] under development by the Eclipse community: to provide an extensible framework to develop model-driven tools to support software modernization, such as quality assurance, documentation, improvement and migration. Thus, our model-based approach and formalization, together with model-based purpose-specific tools – such as those that can be developed in the context of MoDisco – provide the architect the basis for tool support for an architecture reconstruction effort.

Another recurrent architecture design scenario is for the migration of an existing system from a current infrastructure to a target one. In this case, stakeholders have a system of interest already in production, and they need to migrate this system to a new infrastructure, possibly addressing quality attributes that were not addressed by the original system. The need of migration can have multiple sources, such as economical – moving to the cloud due to the expensive operation and maintenance of on-premises production environment –, technological – moving from one technological platform to another due to change in providers –, to quality-based – moving to a new environment to improve robustness and scalability. Architecture-centric software migration uses the architecture of the actual system and the foreseen architecture of the target one, to define the strategy for migration. Usually, migration cannot be done overnight, mainly for core organizational systems, and hence, multiple steps are required which span for a considerable period of time. In this scenario, architects plan alternative paths of migration, devising intermediate architecture to be achieved during the migration project. This migration strategy takes the form of a direct acyclic graph, from a source corresponding to the current architecture, to a target node corresponding to the target architecture. Intermediate nodes are potential architectures to build – to which the software is to be migrated to –, and arrows represent the actual transformations required to go from one state to the other. Clearly, this graph can be adapted during the timespan of the project due to changes in the expectations, to recovery of new information, and to the expertise gained when performing one of the migration steps. While our model-based approach is not targeted to this particular scenario, is can assist in this process. The incremental nature of our process, and the ability to capture and identify alternative solutions that can be chosen from, can be used for an architecture migration effort. The architect can use architecture decisions, solutions and scripts to capture each arrow of the migration, reusing solutions and scripts when possible and appropriately, deriving the architecture descriptions automatically. By focusing on *how* to update the architecture description, inconsistencies between the intermediate ones are easier to detect. When an architecture solution and its corresponding script is changed, the automatically derived architecture also changes. Then, subsequent solutions and scripts (in the path of the migration graph) might fail to be enacted

as the expected state of the architecture description is no longer available. By this means, the architect detects which subsequent states in the migration must be reconsider, and more importantly, which part of the architecture description needs to be reexamined – mainly the part that is the source of the failure of the enactment of the scripts.

Chapter 5

Model-Based Software Product Line Architecture

The software architecture research and practitioner community has been slowly moving towards explicitly capturing architecture decisions together with the characterizing structure and behavior of the system. To a smaller degree, and mainly in the presence of tool support, traceability from requirements to decisions to architecture description elements is also captured. We recognize the significance of explicitly capturing decisions, which spans system conception, construction and evolution. Hence, one of the main principles of the model-based approach to architecture design that we defined in Section §4.2 is based on *explicitly* capturing the architecture decisions along with their impact in the architecture description. To *make decisions* during architecture design is inevitable because the design process is an intellectual and creative activity that is performed according to the architect's knowledge, skills, expertise and experience. Besides, a large set of external factors also impacts architecture design, such as the time and budget restrictions of the project, the understanding of the external stakeholders, the skills and experience of the development team, the business processes to assist or automate, the standards and regulations available or imposed by external authorities, the probability of change and evolution, the potential growth in scale, the stakeholder expectations on quality attributes, the interoperability capabilities of related external systems, the software and hardware platform capabilities of the production environment, and the available tool support for architecting, designing, implementing, testing and deploying, among others. As a consequence, in the current state of the art, fully automating the architecture design process is improbable.

The architect must be aware of any change in the environment, i.e. in the external factors driving architecture design, in order to accommodate to the new reality in any way possible. In the case of changes during the early stages of the development project, the architect "simply" refactorizes the architecture description in order to reflect the necessary changes. A large number of changes tend to corrode the consistency between the captured decisions and architecture description elements. Changes must be reflected in both of them. In the case of potential long-term changes, the architect applies different tactics to improve the maintainability of the architecture, specially on those architecture description elements where changes are more likely to occur. In the case of changes on a long-lived system, a

Chapter Contents

5.1 Architecture in Software Product Lines **255**
5.1.1 Variability 261
5.1.2 Product Line Architecture 267
5.1.3 Product Architecture 273
5.2 Model-Based Architecture in Software Product Lines **275**
5.2.1 Model-Based Approach 277
Conceptualization of Product Line Architecture Design 278
Conceptualization of Product Architecture Design 284
5.2.2 Semantics of Product Line Architecture Design 286
Semantic Functions 287
Semantic Equations 289
5.3 Contributions & Discussion **292**

partial or complete refactoring of the system might be needed, i.e. the system need to be re-designed and re-implemented in order to accommodate a new business and/or technological reality. Explicitly capturing architecture decisions helps the architect in understanding why the architecture was designed in any specific way, and hence, to make more informed decisions when the architecture needs to evolve.

Capturing what architecture description elements populate the architecture description and why they do, is essential and requires a large effort from the architect. However, as we studied in Chapter §4, by this means the architecture description misses the information on how the architect proceeded to create the architecture description in the way it is shaped. The benefits of capturing the how is not actually the automatic generation of the what. Automation is actually a requirement because otherwise the effort to capture why, how and what renders the approach impractical. Having the architect focusing on the why and explicitly capturing how to proceed to reflect that why, makes the automatically-generated what to avoid all inconsistencies with respect to the why. However, the principal benefits goes beyond consistency and even tool independence. It is actually the ability to *reuse* both the why and the how with less or no effort, and consequently improving productivity. In the context of forward architecture design of a single system, *reuse* improves the efficiency and efficacy of devising and weighting alternatives. It provides a cheaper way to explore combinations in the solution space, and also to use the same fragment of a solution in more than one alternative. In the case of system evolution, *reuse* allows the architect to apply any potential solution or alternative devised when analyzing the potential long-term changes of the system during architecture design. In the case of architecture migration, *reuse* allows the architect to apply the same solution fragment in more than one of the paths that are evaluated. In any case, changes to the architecture forces the architect to review the explicitly captured how and by changing it appropriately, the architect can ensure that the what is also updated appropriately — i.e. the automatically-generated architecture description actually reflects all the changes.

However, the *reuse* of decisions and solutions in these scenarios is mostly opportunistic. The architect must detect that a decision or solution previously defined and used, might be helpful to deal with the problem at hand. Also, such a solution must be generic enough — or otherwise equal — to be reused in the new context. Our model-based approach promotes the use of architecture patterns to generically capture any modification to the architecture models and correspondences, even when the modification is not actually a recurrent problem-solution case. By this means, when the solution per se is not reusable (i.e. applying a pattern), the pattern might be reused by applying it with different arguments or configuration information. As in any other context of the Software Engineering discipline where *reuse* is used to improve productivity and quality, and to reduce cost and time-to-market, *planned reuse* provides the best long term results and benefits, but with a larger upfront investment. Then, planning the reuse of decisions and solutions in architecture design implies to develop a single core of several similar architectures, dealing with their commonalities and variabilities, allowing other architects to cost-effectively generate particular architectures by reusing the core one. The Software Product Line approach, one of the most successful approaches to planned reuse in software development, brings to our model-based approach to architecture description and design the suitable scenario for planned reuse of architecture decisions and solutions, where the benefits of explicitly capturing the how — rendering the what implicit — is reflected in the full automation of the generation of particular architectures from a core architecture. This is our main goal in this chapter.

This chapter is structured as follows. In Section §5.1 we first review the motivation for variability in software development and we discuss planned reuse as the cost-effective approach to improve quality, productivity and reliability in software development. Second, we review Software Product Lines (SPLs) as the most notorious approach embracing planned reuse in the context of the development of multiple related (similar) systems. Third, we define our conceptual model for variability in core assets that are reused among the similar products, being the architecture description one of the essential core assets in a SPL. Also, we define our contextual model for the practice of architecture description and design for SPLs. Section §5.2 introduces our model-based approach to software product line architecture description and design, and formalizes this approach in terms of Global Model Management techniques by applying and extending the formalization we defined in Section §3.2 and Section §4.2. Section §5.3 concludes this chapter with contributions and discussion.

5.1 Architecture in Software Product Lines

Software-intensive systems have pervaded our daily lives, from personal electronic gadgets, to house appliances, to transportation, to the work environment. Indeed, organizations in the most diverse industries and of any scale, depend on complex software-intensive systems not only for management and operations, but also to reach and provide their market segment with enriched and empowered products and services. The constant evolution of markets, the increasing demand for highly customized products and services, and the heterogeneity in the user profiles and preferences, in their localization, in the capabilities of devices, and in the supporting software and hardware platforms, have triggered an exponential growth in the complexity and variability of modern software solutions. Now, software development

organizations have to produce many complex variants even in scenarios where a single product used to suffice. Developing each variant from scratch and independently, and maintaining and evolving them separately, are not cost effective approaches and have poor time to market. As a consequence, organizations have a strong motivation to investigate ways of reusing common parts to create new software systems from existing software assets [Jéz12].

The benefits of reuse in Software Engineering have been recognized for decades, being the improvement in quality, productivity and reliability, and the reduction of the development effort, cost and time to market, the most notorious ones [AMC+06]. Since the influential paper of M. D. McIlroy [McI68] on mass-produced software components, the community have actively researched and proposed a plethora of approaches for reuse, mainly targeting the design and implementation disciplines. Some of them have slowly pervaded the practice in industry, mainly patterns, object-orientation, component reuse, application frameworks and more recently generator-based techniques and service-orientation. *Software reuse* is the utilization of previously developed artifacts across the development of multiple systems [FVBS09], both produced by in-house development or acquired from third parties. The range of available reuse techniques is such that, in most situations, there is the possibility of some reuse. However, simply putting components or artifacts together is usually unsuccessful and frequently results in negative impacts to project schedule and total effort [GAO95]. An unplanned ad-hoc approach is only opportunistic and at most has a short-term effectiveness. It is based on the assumption that developers can, at any point in the project, look back and find an already existing piece that fit the problem at hand. But reuse is not something that just happen [Tra88]. For an artifact to be reusable, it must be conceived to provide the right abstractions, genericity, and the degree of variability and configuration that fit into a higher-level envisioned structure, and later to be developed or acquired by the organization. Successful reuse becomes increasingly harder as a development project progresses [Fin88]. For reuse to be successful it must be planned from the onset of the project, or even before at an organizational level. Systematic reuse is driven by a careful and well-coordinated planning process [Lim98]. The IEEE 1517:2010 standard [IEE10] defines systematic reuse as the practice of reuse according to a well-defined, repeatable process, and provides a high level framework for reuse activities. However, planned reuse is not free. It requires an upfront investment to understand the technical opportunities and limitations, and to acquire or develop the core assets to be available for reuse. Tight budgets or investment capabilities make organizations reluctant or incapable of planning reuse, usually preferring the short-term effectiveness of occasional ad-hoc reuse to the long-term benefits of planned reuse. Even in favourable scenarios, planned reuse is not pursued due to the lack of team skills and the availability of the appropriate tool support. Also, and according to I. Sommerville in [Som06], lack of reuse is mainly a managerial issue. Managers prefer known risks associated to unplanned or no reuse, to the unknown risks of investing in and planning the future reuse.

Nevertheless, in the current market scenario of a continuous demand for lower costs, faster deliveries, and better quality, in addition to the inevitable need for multiple variants and customizations of a software system, planned reuse is positioned as the foundational principle for success. Two of the more effective approaches for planned reuse are [Som06] Software Product Lines (SPL) [Nor99] and software product families [Lin02]. Both emerged as a paradigm shift towards the development of sets of related systems instead of individual systems. These approaches embrace the ideas of mass customization and planned reuse, focusing

on efficiently producing and maintaining multiple related software products, exploiting their commonalities while managing their variability. *Software product line* and a *software product family* are different concepts, although some authors tend to use them interchangeably. Both techniques promotes planned reuse and the development of core assets that conform the basis for developing member products. On the one hand, a software product line is targeted and designed to satisfy a specific market and it defines a set of related capabilities that can be included in the products of the product line. Generally, one product of the product line is acquired and installed on a given work environment with the required set of capabilities for that environment. On the other hand, a software product family is an integrated suite of products, each of them having different capabilities [CE00]. Generally, a subset or all products in the product family are acquired and installed in a work environment. Our work is positioned in the context of Software Product Lines. However, the ideas and techniques presented in this chapter can be straightforwardly ported to product family development scenarios.

Product lines are nothing new in industry, for instance avionic, phone and food companies, exploit the commonality of their products in different ways [NCB+14]. Bluntly, a product line is a set of products that together address a particular market segment or fulfill a particular mission. In the case of software systems, *software product lines* has been shown to enable order-of-magnitude improvements in quality, time to market, cost, and productivity, compared to one-at-a-time software system development [CN02]. Several definitions for the term Software Product Line can be found in the research literature. From a market-driven perspective, for L. Northrop in [Nor02] a *software product line* is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way." From a technological perspective, J. Bosch states in [Bos00] that "a software product line consists of a product line architecture and a set of reusable components designed for incorporation into the product line architecture; in addition, the product line consists of the software products developed using the mentioned reusable assets." These two definition share some similarities and the difference in perspective allows to grasp the concept to its broad sense.

The fundamental idea of a software product line (SPL) approach is to undertake the development of a set of products as a single, coherent development process, where the individual products are developed from a core base of reusable assets. The products in the SPL must present commonalities to a considerable extent in order to enable and justify the development of the *core asset base*. This base consists of a collection of *artifacts* that have been specifically designed for use across the set of products. Core assets include the architecture and its description, specifications, software components, tools such as component or application generators, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions, among others [CN02]. Thus, instead of describing or representing a single software system, the artifacts in the core asset base of the SPL describe or represent the set of products for the domain targeted by the SPL. Although it may be possible to create core assets that can be used across the individual products without any modification, it many cases some adaptations are required to make them usable in a particular product [BC05]. Thus, the core assets also identify the variability between the individual products [CHW98]. However, the variability must be such that it does not risk the effective reuse of the core

assets and that does not compromise the existence or development of the SPL [DS99]. Reuse of core assets is key to productivity and quality gains [Jéz12]. In other words, a SPL succeeds when the commonalities shared by the individual products can be exploited to achieve economies of production.

The software product line approach supports large-scale reuse during software development. According to J. Poulin in [Pou97], as opposed to traditional reuse-in-the-small approaches, reuse in software product lines can even be as much as the 90% of a member product. For J. McGregor et al. in [MMYJ10], with reuse percentages running above 50%, an organization typically recoup the extra cost of making assets reusable after two or three products. Thus, even a product line of a few products can be very profitable. However, the benefits of a software product line approach does not come for free as it requires an extra upfront investment [LSR07]. This investment is required to overcome the common barriers of adoption that organizations face, such as the needed changes in the organization structure, the development process, the skills of the development team, the commitment to a global and complete view of the product line captured by its scope, domain and architecture, and the fact that the benefits are not actually visible in the short-term [BCK03, HT03, LSR07]. Different adoption schemes exist, from a big-bang approach where a completely new product line is set up by developing a reusable infrastructure for the whole range of products right from the start, to an incremental approach where the reusable assets developed are those needed to support the new few upcoming products, excluding highly uncertain potential products [SV02]. Nevertheless, organizations having a solid business case targeting a particular market segment, for which one or some similar products were developed, can justify the costs of developing a software product line.

Once an organization transited from single-product development to software product line development, it evolves through a number of maturity levels. As defined by J. Bosch in [Bos02], the first level is a *standardized infrastructure* determining the technological support, such as operating systems and commercial components such as database management systems, document content management systems, and even third-party domain-specific components. The second level is a *standardized platform* that, in addition to an standardized infrastructure, it captures the functionality that is common to all products and applications, and include self-developed components for the functionality that is not provided by the infrastructure. Third, the *software product line* level is reached when the common functionality is captured by shared artifacts and components. The final level is the *configurable product base* in which the domain is relatively stable and the development moves toward one configurable product base that is configured into the product bought by the consumer. The optimal approach for a company depends on the maturity of the organization and on the maturity of the application domain. The higher the maturity level, the greater the benefits obtained.

A *software product line development process* is a software development process that describes the construction of both the core asset base of the product line and the derivation of particular products from this core base. Such a development process is both business-centric and architecture-centric [LSR07]. It is *business-centric* in the sense that it aims to connect the construction of the software product line to the long-term strategy of the business. It entails the analysis of the business domain and the key decisions about which products to

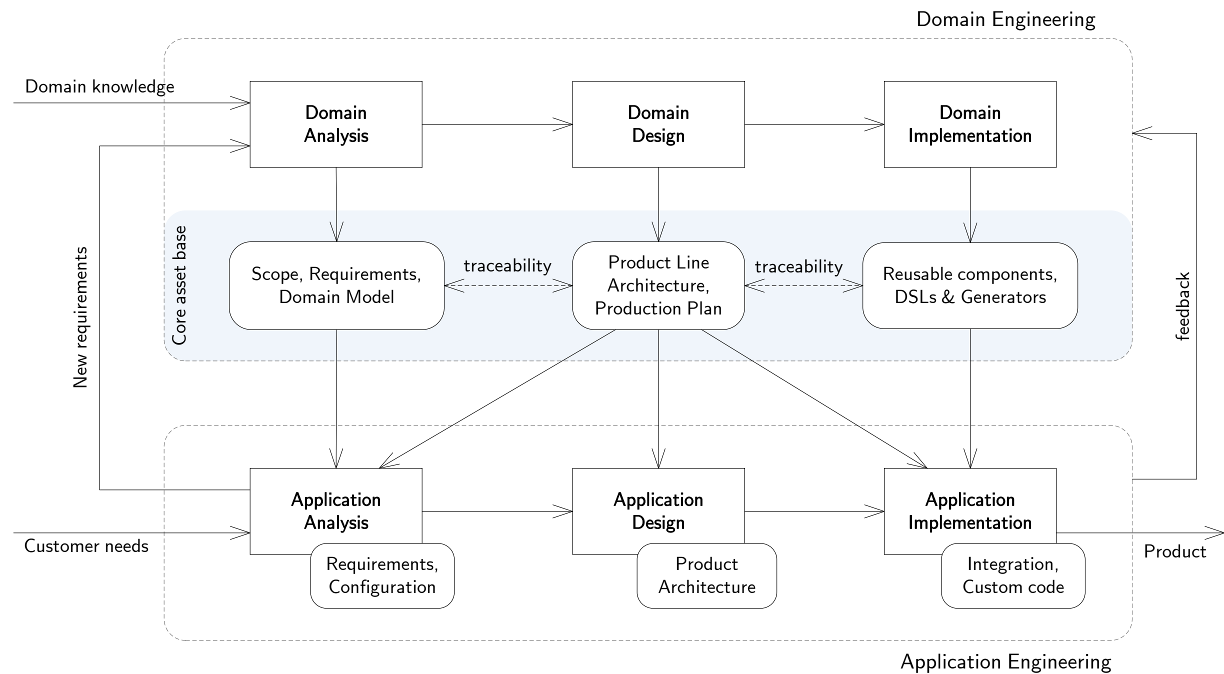


Figure 5.1: *Software Product Line Development Process.*

The figure illustrates the general Software Product Line process, as it can be found in the research literature [CE00, Nor02]. Major activities are show in squared boxes and major artifacts in rounded-corner boxes. Horizontal arrows indicate control flow, making each sub-process sequential as in the waterfall model. Vertical arrows indicate data flow, illustrating the provision of artifacts from one activity to another; some of these artifacts are not shown in the figure.

include in the portfolio, which functionality they provide, and which are the quality expectations for these products. A SPL development processes is *architecture-centric* in the sense that it relies on a common product line architecture that captures the key decisions on how the products address the architecturally-significant concerns. In addition, the product line architecture plays a central role in capturing the commonality and variability between the products in the product line. Although there are some approaches that do not rely on a product line architecture to SPL development, like [SH04], they do not achieve the higher maturity levels for SPLs [Bos02].

Several processes have been proposed in the last decade [CE00, CN02, Nor02, PBL05], and most of them share the same organization. A SPL development process consists of two intertwined sub-processes. *Domain Engineering* or *development for reuse*, focuses on the identification, definition, development and maintenance of the core asset base. *Application Engineering* or *development with reuse*, focuses on the development of final products, using the core assets, to address customer requirements. Figure 5.1 illustrates this common organization. The SEI's framework to SPL [NCB+14] identifies a third intertwined sub-process to provide *management* of the SPL development effort. Both Domain and Application Engineering follows an iterative life cycle, that not necessarily occur synchronized. Each iteration follows a waterfall life cycle [HT03] consisting of Analysis, Design and Implementation activities.

The *Domain Engineering* sub-process consists of collecting, organizing, and storing past

experiences in building systems in a particular domain, in the form of reusable assets and providing adequate means for reusing them for building new systems [Per07]. This sub-process is in charge of identifying, defining, developing and maintaining the common and variable parts of the core assets that populate the *core asset base*. The input to the *Domain Analysis* activity is the domain knowledge and expertise available to the development team. Such knowledge come from diverse sources, such as customers requirements, foreseen future requirements, domain experts, business and technical documentation, prototypes, existing processes and systems, among others. The goal of the activity is to determine the *domain scope*, to identify and capture the set of reusable and configurable *requirements*, and to create the *domain model*. These artifacts are then used in the *Domain Design* activity which designs the *Product Line Architecture* to address the critical requirements in the domain, and to serve as the architecture of reference for the products in the product line. A *production plan* is also defined to capture, specify and communicate how the core assets are used to develop a product in the product line [CM02]. The main goal of the *Domain Implementation* activity is to implement and test reusable core assets to be used to produce the base implementation of products in the line. These core assets can be actual implementations that end up forming part of the product implementations, such as reusable components, interfaces, protocols, database scripts, compilation scripts, integration scripts, deployment scripts, configuration scripts, etc. These core assets can also be tools to be used by the implementers of the actual products, such as domain-specific programming languages, graphical user interface generators, component generators, component-integration generators, etc. The output of this activity is the population and refinement of the core assets at the implementation level.

The *Application Engineering* sub-process, also known as product derivation, consists of building the actual products (applications) based on the artifacts produced by the Domain Engineering sub-process and exploiting the product line variability. The major input to the *Application Analysis* activity are the Domain Analysis artifacts. Commonality and variability must be communicated to stakeholders to allow them to decide which variant of the product line they require. Additional product-specific requirements and constraints can be provided by the stakeholders. In this case, these requirements must be analyzed to evaluate to what extent they deviate from the product line scope, how they fit in the context of the reference requirements of the product line, and if they can be supported by the product line architecture. Requirement renegotiation and reformulation might be needed in order to reconcile both set of requirements. If such a reconciliation cannot be achieved, it must be decided whether the desired product is actually a member of the product line, or if it is not and must be developed independently. The particular variant selected is captured by the *product configuration*, that define which alternative must be chosen for each variation point of the product line. The *Application Design* activity uses the product configuration and the product line architecture to develop the *product architecture*. The product architecture is further refined in order to realize those customer requirements that were not covered by the Domain Design activity. The *Application Implementation* activity is responsible for building the completely operational application. According to the guidance provided by the production plan and the product line architecture, the reusable implementation assets are selected and included in the application implementation. Generators are used to produce additional components and to facilitate integration and deployment. Additional components participating of the product architecture, that are exclusive to the product and not part of the product line's domain, are also developed.

Domain Engineering and Application Engineering are intertwined sub-process. Decisions, artifacts and domain-specific tools are fed from Domain to Application Engineering. In turn, the experience obtained and the artifacts produced during the development of a particular product are used as feedback to Domain Engineering. Subsequent iterations of the Domain Engineering activity considers this feedback to re-examine the scope and domain model, to re-evaluate and refine the product line architecture, and if applicable, to create abstract and generic versions of the developed product-specific components to be included as core implementation assets. In order to achieve the economies of production, the investment on Domain Engineering must be outweighed by the benefits of deriving individual products in Application Engineering. According to S. Deelstra et al. in [DSB05], this goal is not easy to reach in practice as deriving individual products simply from shared software assets is a time-consuming and expensive activity. First, low investments in Domain Analysis result in a product line scope that does not consider the broad spectrum of potential products. Consequently, a large amount of effort is required to develop each product due to the deviation of its requirements from those captured and addressed by the product line. Second, low investments on technology result in reference core assets that must be manually adapted into product assets, and core implementation assets that have poor configuration or integration capability requiring additional effort to make them operable in a final product. A large investment in Domain Engineering would allow a more detailed Domain Analysis, with broader coverage, rendering a software product line in which practically all commonality and variability have been identified and addressed. Also, a large investment in technology renders the automation of the derivation of product assets from core assets. Thus, in this scenario, nearly no effort would be required for Application Engineering.

Our work is focused on the Software Architecture practice in the context of Software Product Line development. As we reviewed before, the architecture of a SPL plays a critical role as it not only addresses the significant concerns of the products in the product line, but also captures the commonality and variability in the architectures of these products. In this section, we first discuss in Section §5.1.1 the concept of *variability*, not in the context of the whole SPL, but rather in the context of a single core asset. In Section §5.1.2 we review *product line architectures*. We define the context of its practice and we discuss how it is described and designed. This section extends the set of constructs that we discussed in Section §4.1.2 in the context of single-system development. In Section §5.1.3 we proceed analogously for the case of *product architectures*.

5.1.1 Variability

Software Product Line (SPL) is the approach for planned reuse that focuses on effectively and efficiently developing, maintaining and evolving multiple related software products, exploiting their commonality while managing their variability, and anticipating required and potential variants for building actual products. The *commonality* among products is understood as the set of assumptions known to be true for each member of the product line [CHW98]. As opposed to commonality, *variability* among products is understood as the set of assumptions about how the members of the product line differ from each other [CHW98, WL99]. According to F. Bachmann et al. in [BC05], "the goal of variability in a software product line is to

```
context Feature
  def: RequiredFeatures : Set(Feature) =
       self.constraints.to

  def: AllRequiredFeatures : Set(Feature) =
       self.RequiredFeatures→closure(f | f.RequiredFeatures)

  inv: self.AllRequiredFeatures→excludes(self)
```

Figure 5.2: *Metamodel for Feature Models.*
The figure illustrates the metamodel defining the modeling language for feature models. Our metamodel is a refinement of that proposed by K. Czarnecki et al. in [CHE04].

maximize the return of investment for building and maintaining products over a specified period of time or number of products." As we discussed before, low costs, rapid time-to-market, and high quality products are means to maximize that return of investment. For K. Pohl et al. in [PBL05], variability in a software product line enables the development of customized products by reusing predefined, adjustable artifacts. In a more general sense, *variability* is the ability of a process, a system, an artifact, or a development environment to support the production of a set of artifacts that differ from each other in a preplanned fashion [BC05]. Then, from a software perspective, *variability* is the ability of an artifact to be configured, customized, extended, or changed for use in a specific context [GBS01], by means of preconceived mechanisms. Consequently, *variability* not only characterizes the product line as a whole, but also it pervades every individual artifact conforming the *core asset base* of the product line. Variability can be considered in two orthogonal dimensions [PBL05]. *Variability in time* refers to the different versions of the same artifacts that are valid at different times. It is strongly related to change and evolution, and dealing with it usually involves version control and configuration management. *Variability in space* refers to the different variants of the same artifact at the same time. It is strongly related to devising the different application scenarios for the artifact and building it to accommodate all of them. Dealing with variability in space is the goal of the software product line approach [Erw10].

Using *features* has been one of the most popular techniques for modeling variability. K. Kang et al. in [KCH+90] defines *feature* as "a prominent or distinctive user-visible aspect, quality or characteristic of a software or system." As stakeholders usually refer to product characteristics in terms of the *features* that the product has or delivers [Jac04], it is natural to express commonality and variability between products in terms of features, and to use them in the context of software product line development. Thus, K. Czarnecki et al. in [CE00] define *feature* as "a system property relevant to some stakeholder used to capture commonalities or discriminate among systems in a family." K. Kang et al. in [KCH+90] propose to use a *feature models* to model the variability in a software product line. A feature model consists of hierarchical decomposition of features. The root feature refers to the complete system which is progressively decomposed into more refined features (internal nodes). Although feature models have the advantage of being clear and easy to understand, they lack expressiveness to model relations between variants or to explicitly represent variable parts (also known as variation points) in core assets. As a consequence, several extensions were added to the original feature models proposed by K. Kang by the research and practitioner community, to enrich their expressiveness, particularly with the purpose of using feature models beyond requirements. J. M. Jézéquel review in [Jéz12] the most relevant extensions. In particular, in Figure 5.2 we illustrate our modeling language for *feature models* by means of a metamodel, which is actually a refinement of that presented by K. Czarnecki et al. in [CHE04]. According to our metamodel, a *feature model* is composed by a non-empty set of *root* features, being one of them the main root feature representing the complete system. Root features are used for modularization purposes. Each *feature* has a distinctive name, and can be either one of the *root* features, a *solitary* feature or a *grouped* feature. A solitary feature is an ungrouped feature that represents a single characteristic of the system. It defines a cardinality in terms of a lower and upper bound, where a lower bound of zero makes the feature optional and an upper bound greater that one makes it possible to instantiate the feature multiple times in a configuration. A *grouped* feature is that that participate in a group. Every feature can have a set of *members*, i.e. it can be decomposed or refined. The members of a feature can be *solitary* features, *references* to root features defined in the feature model, or *groups*. A *group* defines a cardinality that determines the lower and upper bounds for the number of *group members* that can participate in a configuration. *Group members* are either *grouped* features or *references*. A *group* is characterized by an *operator* that specifies if the group provides exclusive alternatives (XOR), inclusive alternatives (OR) or inclusive members (AND), i.e. whether the configuration can select one, several or all of the group members. Other constraints and dependencies between are captured by means of *constraints*, being *requires* and *excludes* that most common ones.

Features can play different roles in a software product line process. In the context of Domain Engineering, features are used during Domain Analysis to capture the commonality and variability between the products of the product line [KCH+90]. Thus, feature conforms the unit of evolution of the product line, capturing the optional and potential requirements to be designed and implemented [SHTB07]. In the context of Application Engineering, customers and product developers selects the group of features that must be provided and delivered by the product being built. From this configuration of features, product developers carefully coordinate the construction of the product using the complicated mixture of parts of different core assets — such as the architecture or component implementations — that are involved in the selected features [Gri00]. However, *feature models* variants can only provide

a hierarchical structuring of high-level product functionalities [CHS08], with practically no connection with the actual software products or core assets. This limitation induced the community to look for more expressive mechanisms for representing variability and connecting it to the core assets.

According to J. M. Jézéquel in [Jéz12], a software product line can be fully modeled as an *asset model* that models the set of core assets (i.e. the reusable elements that are used for the development of new products), and a *variability model* that represents the commonality and variability between product line members. Since the standard languages are generally not developed to explicitly capture variability, artifacts — and particularly models — in software product lines are usually expressed by extending the standard languages or annotating the artifacts. As studied in [HMPO+08], there are two different approaches to deal with variability in core assets: amalgamated and separated. The *amalgamated approaches* extend the languages with specific constructs for expressing variability. In the case of modeling languages, such extensions are defined at the metamodel level, and thus a new combined metamodel is developed. Then, practitioners build core assets by means of a mixed textual syntax or graphical notation, using constructs from the base language altogether with those for expressing variability. Amalgamated approaches can be realized by ad hoc extensions to existing languages, by generic extensions that can be woven (at the metamodel level) into any language, and by defining purpose-specific ad hoc languages per se. *Separated approaches* keep the representation of variability separated from the representation of the core assets. The languages for capturing core assets remain unchanged, and variability is captured by means of a separated language. Elements in the variability model relate to elements in the core asset by referencing them in some specific way. The key characteristic of these approaches is the clear separation of concerns, both at the language level and at the artifact level. In addition to enable designers to focus either on building the core assets or in capturing their variability, these approaches also enable to use multiple variability models for the same core asset. Separated approaches have paved the way for the ongoing OMG's standardization effort on Common Variability Language (CVL) [Hau13, OMG14a]. There are three kinds of separated approaches: orthogonal variability modeling, those based on feature models, and those based on decision modeling. K. Czarnecki et al. in [CGR+12] provides a comparison of variability modeling approaches in the two latter kinds. A review of both amalgamated and separated approaches, as well as the different kinds of those approaches, was developed by J. M. Jézéquel in [Jéz12].

Variability management is the key characteristic that distinguishes Software Product Line Engineering from other software development approaches [BFG+01]. It is central to both Domain and Application Engineering [HP03] and it covers the entire life cycle [PBL05], from requirement elicitation [HT03], to product line architecture design [SWPH09, TH02, ZPJ+11], to product derivation [DSB05, ROR11], to product testing [NTJ06]. As stated by F. Bachmann et al. in [BC05], mismanaging variability in a software product line may result in adding unnecessary variability, implementing variation mechanisms more than once, selecting incompatible or awkward variation mechanism, and missing required variations. *Variability management* is the activity of managing the variability in the multiple artifacts that are used for building the member products of a software product line. Variability management takes place throughout the life cycle of the software product line. It encompasses the activities of identifying and capturing the commonality and variability among the required and potential

Figure 5.3: *Conceptual model of variability management for core assets.*

The figure illustrates our conceptual model that captures the key concepts, and their relations, pertaining the practice of variability management in software product lines. Our model is based on the conceptual model presented by F. Bachmann et al. in [BC05]. We refine that model by naming associations, revising the cardinality of association ends and including those missing, and by establishing the relation between variation mechanism and variants. Also, we extend their model by considering the attached process as a first-class concept — which replaces the *process* and *condition* concepts included by the authors —, and by including *product assets* to the model. In the figure, we also try to improve the layout of the concepts.

products, explicitly representing them in the artifacts, managing their inter-dependencies, and supporting their instantiation [LSR07, SJ04]. L. Chen et al. report in [CABA09, CAB11] a systematic review of research results on variability management and their effectiveness. Variability management is a non-trivial undertaking as it involves extremely complex and challenging tasks, which need to be supported by appropriate approaches, techniques, and tools [BFG+01, SD07].

F. Bachmann et al. in [BC05] define a conceptual model of variability management for core assets, aiming to provide guidelines to core asset creators on how to model variability

explicitly so as management becomes feasible. That conceptual model conforms the basis for ours, which we illustrate in Figure 5.3. A *core asset* is formed by exactly one common part and a possibly empty set of variable parts. The *common part* is the part of the core asset that is shared by the product assets instantiated from the core asset, and then it is used as is. The *variable parts*, also known as *variation points*, represent those parts of the core asset where the product assets may vary. Each variable part offers one or more variation mechanisms. A *variation mechanism* captures the mechanism by which product developers can obtain a variant for the variable part. Thus, it helps to control the required and possible adaptations of the core assets and supports the product developer in their task. A variation mechanism can take different forms, from actual elements to populate the variant, to configuration information for actual elements, to templates, to generator tools, to examples, among others. Possible variation mechanisms have been identified and captured by the research community [AG01, JGJ97]. However, the concrete variation mechanisms can be determined only in the context of a concrete product line development [BC05], as they depend on the team skills, the technology available, time and budget constraints, and their impact on quality attributes such as maintainability and performance. Organizations tend to product their own catalog according to their concrete scenario. In addition, variability mechanisms strongly depends on the actual kind of the core asset. There are two main categories of variation mechanisms. A *creation mechanism* provides product developers with the means to create a new variants. The created new variants are captured as available variants for the variable part, which can be selected afterwards by other product developer by means of a *selection mechanism.* In Figure 5.3 we use navigability to emphasize the fact that the creation mechanism is available prior to the variants created using it, and that the selection mechanism is used to select an already existing variant. A *variant* is the result of exercising a variation mechanism. It consists of a single common part that participates in the product asset that uses the variant. A variant can also have variable parts, with their corresponding variations mechanisms and variants. Each *core asset* has an *attached process* that specifies how the core asset must be used in the development of actual product assets, i.e. the attached process guides the instantiation of product assets from the core asset. Product constraints, production constraints, and the production strategy influence the definition of attached process. An attached process is performed using a set of steps, in the particular order appropriate for the process, in order to instantiate the product asset. At each *attached process step*, a specific *variation mechanism* is applied to address the corresponding *variable part.* Each step defines a *condition* that must be met to render the step applicable. The condition is expressed in terms of the preferences or configuration of the expected product assets, and possibly in terms of actual elements participating in the product asset. An attached process usually specifies the automated tool support for accomplishing these steps. However, the ability of automation strongly depends on the formality of the languages used to capture both the core asset and the attached process steps. A *product asset* is a concrete instantiation of a *core asset* in which all variable parts have been resolved according to particular preferences or configuration. The product asset is the result of applying the attached process of the core asset. The product asset is then conformed by the combination of a set of *common parts*, particularly, the common part of the *core asset* itself, and the common parts of all used *variants.*

We review and discuss the concept of product line architecture next in Section §5.1.2. Later in Section §5.2 we define our model-based approach to software product line architecture

description and design. There, we explain how the product line architecture description, as a core asset, can be understood in terms of the concepts of the conceptual model for variability management that we defined in this section and illustrated in Figure 5.3.

5.1.2 Product Line Architecture

Software architecture plays an essential role in achieving intellectual control over the sophistication and complexity of medium- and large-scale software systems, providing practitioners a means for dealing with complexity, managing and controlling development goals and risks, and assessing quality. In the context of software product lines, the role of software architecture is even more important. It allows practitioners to capture the commonalities and variabilities present in the different products [OM07]. According to R. Macala et al. in [MSG96], an architecture-centric approach to software product line development makes it *better* as it promotes designs that improve requirements consistency and traceability, *cheaper* as it promotes the definition of interfaces and control, reduces the need for non-essential or redundant implementations, and helps communication among stakeholders, and *faster* as once developed components are architecturally compliant, they can be reused with little effort. Architecture-centric development process enable developers to achieve the higher maturity levels for software product lines [Bos02]. The product line architecture is the main artifact of the software architecture practice in the context of software product line development.

A *product line architecture* addresses the critical functionality, the expectations on quality attributes, the constraints and the potential risk of failure that are common to all the products in the product line. In addition, it identifies and captures those parts where the products differ, and explicitly provides the variation mechanisms to support the diversity among the products. The variation mechanisms are the means of the product line architecture to specify how particular variants can be produced, while maximizing the shared part in the implementation [JRL00]. F. Bachmann et al. discuss in [BB01b] variability at the architecture level. R. Taylor et al. in [TMD09] defines a *product line architecture* as "the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation." J. Bosch presented in [Bos02] three maturity levels for product line architectures. At the lower lever, *under-specified* product line architectures capture the commonalities between the products to avoid re-specification, to provide a basic frame of reference, and to allow substantial freedom in product architecture derivation. The higher level of *specified* product line architectures capture commonalities and variabilities, allowing product developers to exploit variation points to determine product specific functionality. At the highest maturity level, *enforced* product line architectures capture all commonalities and variabilities of the product line in a way that products do not need nor are allowed to change the enforced architecture. Higher levels ease automation to the cost of freedom. The development organization must decide which level best fit the product line under development. A product line architecture is a *core asset* in the core asset base of the software product line. It serves as the basis for defining the software architecture of each particular product in the line. Along with the *production plan*, the product line architecture guides the construction of the actual product implementation.

The product line architecture is designed in Domain Design activity of the Domain Engineering sub-process. The architect uses the *scope*, the *domain model* and the *reference requirements* produced during Domain Analysis, as the input to architecture design. The product line architecture is devised, captured and communicated, playing a key role in guiding the enactment of the other activities of the development sub-processes. In Domain Implementation, in addition to the acquisition or implementation of the components that are common to all products, the variation mechanisms *decided by the architect* are reified. To this end, generic, configurable and customizable components are implemented so as to be later adapted into product components. Also, domain-specific languages together with integrator and generator tools are developed in order to support the more complex variation mechanisms that provide higher levels of automation. The product line architecture impacts all activities of the Application Engineering sub-process. Application Analysis uses the product line architecture to support communication to stakeholders on the actual capability of the product line to deliver variability. Product analysts must be aware of the required effort to produce variants in order to accommodate time-to-market and budget restrictions, as well as customer expectations on the product. To select the actual configuration of the product to be developed, it is essential to know to what extent automation is available by the variation mechanisms, as whether those mechanisms are already implemented and available in the core asset base. Also, the product line architecture helps product analysts to understand and measure the deviation of particular customer requirements or expectations with respect to those already addressed by the architecture. Application Design uses the product line architecture to develop an architecture that is specific for the product. To this end, the product architect uses the product configuration and product-specific requirements produced during Application Analysis to select the specific variation mechanisms that provide the best product-wide long-term benefits for the product construction and evolution. The architect uses the attached process to the product line architecture guides the instantiation of that architecture into a product architecture where all potential variability is resolved. By this, the product architecture is developed capturing all decisions on which variation mechanisms to select and which specific variants must be used or implemented to build the actual product. Application Implementation uses the product line architecture, as a reference to the particular product architecture developed, to exercise the variation mechanisms and to implement any custom code required for missing components or for integration.

Designing the architecture for a software product line differs from designing the architecture for a single product. The product line architecture must be designed, evaluated and communicated in terms of a large number of potential products, many of which may never be realized [TMD09]. The complexity of product line architecture design lies not only in the critical decisions that must be made to address functionality and quality expectations, but also and mainly, in their integration with the expected variability of the software product line [KM04]. This is actually a challenging task as quality attribute requirements or expectations may also vary between different products in the product line [NI07]. According to L. Bass et al. in [BCK03], product line architecture design is concerned with the set of explicitly allowed variations, along with also satisfying the behavioral and quality goals. Thus, identifying the variable parts is the architect's responsibility, as it is to provide the built-in variation mechanisms for achieving them. Those variations can be substantial, as they can reflect on the functionality, behavior, quality attribute expectations, supporting software platform and middleware, supporting network and hardware platform, scale factors,

etc. According to L. Northrop et al. in [NCB+14], choosing the appropriate variable parts and variation mechanisms is among the most important tasks for the architect. The impact of this kind of decisions goes beyond the technical aspects of variability. Deciding on available variable parts is deciding on the potential products that participate in the product line. Deciding on the available variation mechanisms and the support for automation is deciding on the tradeoff between the upfront investment in Domain Engineering and the one-at-a-time investment in Application Engineering to deliver new products.

Provided that product line architecture design is actually software architecture design, and that a product line architecture has similarities to the architecture for a single product, some authors [BCK03, Bos00, GS04] rely on the architecture design methods and practices that we reviewed in Section §4.1. In order to deal with variability, they propose models and methodological guidelines in order to identify variation points and to decide variation mechanisms. In particular, SEI's approaches to product line architecture design [BB01a, BCK03, NCB+14] rely on the Attribute-Driven Design method (ADD) [WBB+06]. We study ADD later in Section §6.1.2 when we exemplify how to capture architecture design knowledge using our model-based conceptualization and formalization. to architecture design of a single system. Other authors, however, have gone beyond the general considerations on variability on established architecture design methods, and have defined purpose-specific design methods and models for software product lines. M. Matinlassi provided in [Mat04] a comparison of these methods, and concluded that while there are distinguishable ideologies supporting them, the methods do not overlap even though all of them target product line architecture design. There are four distinct kind of methods [TLY12], feature-oriented, process-driven, component-based, and quality-driven.

A representative *feature-oriented method* is Feature-Oriented Reuse Method (FORM), defined in [KKL+98] as an extension to the Feature-Oriented Domain Analysis (FODA) method defined in [KCH+90]. FORM uses feature modeling to discover, understand, capture and communicate commonalities and variabilities of the product line. Features are separated in terms of services, operating environments, domain technologies, and implementation techniques. The feature model is then used to define a parameterized reference architecture which is organized in three different viewpoints (subsystem, process and module) that have intimate association with the features. Subsystems are used to package services and to allocate them in a distributed environment. Subsystems are further decomposed into processes considering the operating environments. Modules are defined based on the domain and implementation features, and serves the basis for creating reusable components. A representative *process-driven method* is the Family-Oriented Abstraction, Specification and Translation (FAST) [WL99]. FAST is sourced in industry practices and targets the development of families of products. It divides the process of a product line into three sub-process, domain qualification, domain engineering and application engineering. Domain qualification captures the general business needs and creates an economic model hat estimate the number and value of family members and the cost to produce them. Domain engineering generates a language for specifying family members, an environment for generating them from their specifications, and a process for producing them using the environment. Application engineering generates specific members to meet customer requirements. Two representative *component-based methods* are Component-Oriented Platform Architecting (COPA) [OMAO00] and KobrA [ABM00]. The specific goal of COPA is to achieve the best possible fit between business, architecture, pro-

cess and organization — that resulted in the BAPO approach [Omm02]. COPA contains three sub-process: product family engineering in charge of the family architecture, platform engineering in charge of reusable assets, and product engineering in charge of generating products. COPA starts with a business phase analyzing customer needs and expectations, which are the input to the architecting phase. The architecting phase uses five views (customer business, application, functional, conceptual and realization), and produces an architecture that describes (what), justifies (why) and guides (how) the construction of products, according to the definition of lightweight architecture provided in [Mul13]. KobrA is both a component-based incremental product line development approach and a methodology for modeling architectures, and supports the Model-Driven Architecture (MDA) [OMG03] approach to software development. KobrA defines a complete product line engineering process, being framework engineering and application engineering the most relevant sub-processes. In framework engineering, the specification of components is described by a set of models, such as interaction, structural, activity and decision models. The latter is used to capture variability. The purpose of application engineering is to implement the framework (consisting of the component specifications) in order to derive member products from the family. A representative quality-driven method is the Quality-Driven Architecture Design and quality Analysis (QADA) [VTT14], a method for both software architecture design and evaluation. QADA describes the architecture design activity of a development process, providing also the description language to use in artifacts. QADA starts with a requirement engineering phase to collect the driving ideas and technical properties of the system to be designed, as well as functional and quality requirements. QADA uses three viewpoints (structural, behavioral and deployment) at two levels of abstractions (conceptual and concrete). The conceptual level covers conceptual components, relationships and responsibilities, and it is targeted to the high level stakeholders of the product line (e.g. architect, manager). The concrete level is aimed for software engineers and designers. Analysis in the conceptual level deals with variability and quality attribute analysis. Analysis in the concrete level deals with customer value and scenario satisfaction.

According to M. Matinlassi in [Mat04], these methods do not compete with each other as each of them has a special goal or ideology. As the support from commercial tools is not enough for product line architecture design, some of these methods provide their own specific extensions to those tools. From our point of view, in order to provide systematization to the design activity and the product derivation, or even automation, these methods not only refine the specific tasks to be followed by the architect, but also they restrict the specific architecture description to use. While some of them forces a particular architecture framework in terms of specific viewpoints, others require the use just one particular language, such as the Unified Modeling Language (UML).

The *description* of a product line architecture follows the same structural organization of the architecture of a single system. As we reviewed in Section §3.1, the contextual model of the ISO/IEC/IEEE 42010:2011 standard [ISO11] on the architecture description practice, that we illustrated in Figure 3.1, an *architecture description* expresses one or more *architectures* exhibited by a set of systems. Even though the conceptual model of the standard restricts the architecture description to a single architecture of a single system, we discussed in Section §3.1.1 that this limitation is not actually necessary: an architecture description expresses or represents the architecture of a system, family or line of systems. The struc-

tural organization in terms of architecture views and architecture models is considered and followed by the research literature regarding architecture for software product lines. For instance, as we discussed before, product line architecture design methods tends to enforce the organization of the product line architecture description in terms of a specific set of views that is adequate (and required) by the design method.

However, the application of single-product architecture description techniques are not enough to describe product line architectures, as it is also relevant to deal with variability. The Software Engineering discipline has had to deal with variations, evolving artifacts, forking, branching and merging for years, particularly in the context of source code [TMD09]. Version control and configuration management systems (e.g. CVS and SVN) are used for this purpose. As we discussed before, these tools are suitable for variability in time. Version control can be a viable option for versioning architecture descriptions as a whole, helping the architect to track the evolution of architectures over time. However, as stated by R. Taylor et al. in [TMD09], this approach does not work well for creating or maintaining product line architectures. Managing a product line architecture means establishing a core set of design decisions and then combining those core decisions with variation points consisting of additional decisions. In general, variation points spread throughout and across architecture models, and this is why architecture description languages embed variation points with models. Because version control works at the level of artifacts (files), it is difficult to use them to manage elements within files. The research and practitioner community have devoted a large effort to adjust or define architecture description languages (ADLs) in order to embed support for variability and make them suitable for software product lines, both with a general purpose or specific to particular domains. We reviewed ADLs in Section §3.1.2. We discuss next some notorious ADLs aimed for software product lines.

Koala [OLKM00] was developed by Philips Electronics to specify and manage the software systems for their consumer electronic products. We have introduced Koala in Section §3.1.2 as a first generation ADL targeting the domain of embedded and real-time systems. Koala defines components as the unit of design, development and reuse. Each component can be refined by contained components, and provides and requires interfaces which are a set of semantically related functions. Variability is captured by means of switches and optional interfaces. While switches are the constructs to represent variation points, optional interfaces allows the definition of several components providing similar services. Koalish [ASM03] is focused on configurable software product families, the specific class of product families where its members can be developed with no need for component integration code. Koalish is based on Koala, and it adds as a new variation mechanism the ability to select the number and type of contained components, and writing constraints that must hold for individual members of the family. A configuration model explicitly captures the set of valid product configurations of the family. ADLARS [BBSK05] is an ADL developed for describing product line architectures, particularly in the context where the system's variability is captured by means of a feature model that identifies mandatory, optional and alternative features. ADLARS proposes three views, system-, task- (concurrency) and component-level (structure) view, augmented by interaction themes to capture behavior. ADLARS supports relationships between the system's feature model and the architectural structures to capture variability in the architecture. ALI [BSK+08] is an extensible ADL based on ADLARS. ALI aims to address the limitations the authors found in the application of other ADLs to indus-

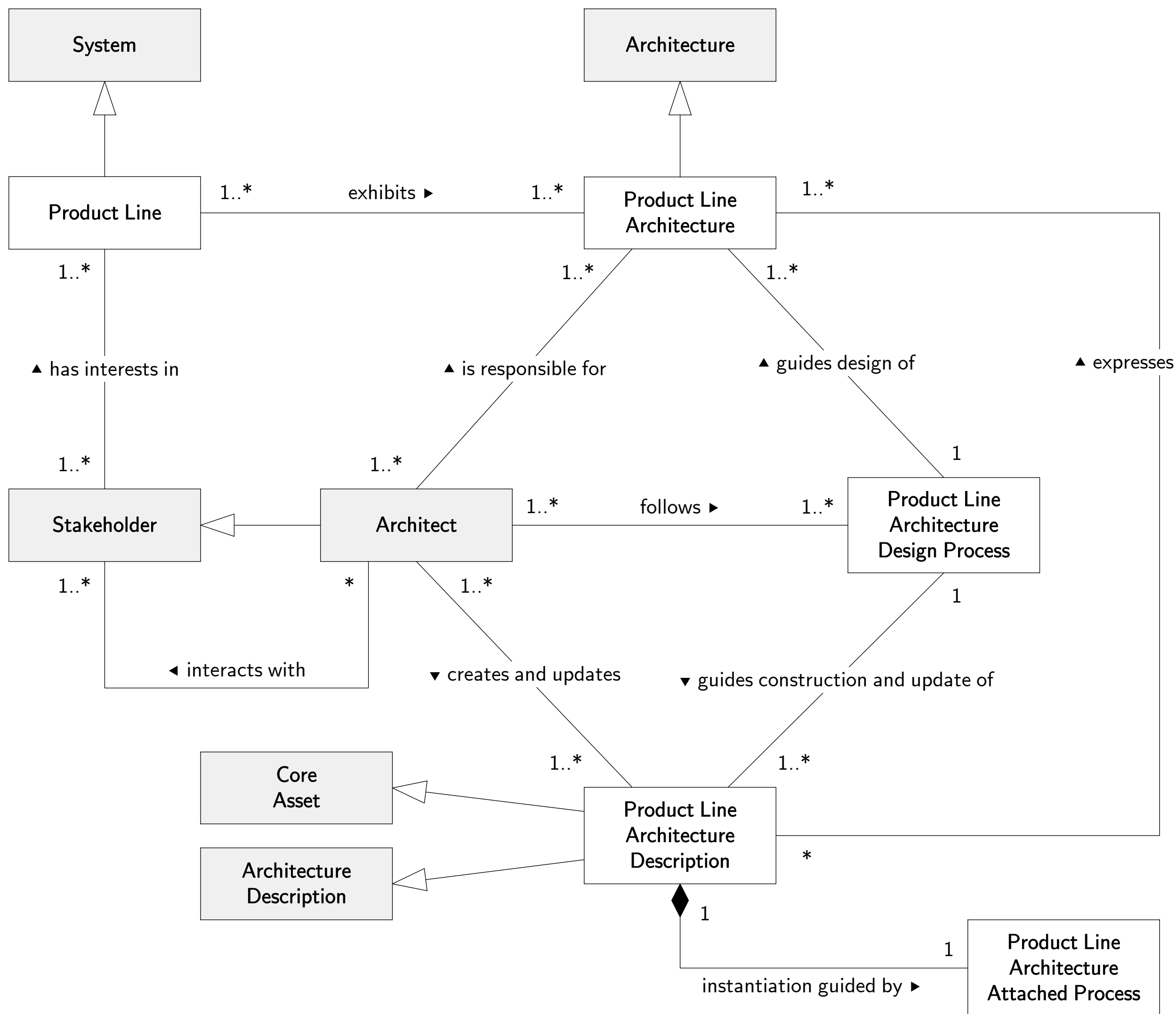


Figure 5.4: *Context of the product line architecture practice.*
The figure illustrates the main concepts pertaining to the context of the practice of the product line architecture description and design.

trial scenarios, such as scalability, over constraining syntax, single-view architectures, and the lack of tool support. The drivers of ALI are flexibility in interface descriptions, architecture pattern description, formal syntax for meta-information and linking capability between the feature and architecture spaces. The support for variability is inherited from ADLARS. xADL [DHT05] is an infrastructure for the development of XML-based ADLs, that facilitates the definition of customized ADLs for particular domains. We have introduced xADL before in Section §3.1.2 when we discussed extensible ADLs. xADL supports variability through optional elements that may or may not be included in products in the line, variant types to enable selection among a set of alternative elements, and optional variant types combining both. A relevant characteristic of xADL is its tool support with ArchStudio [DAH+07], an Eclipse-based environment of integrated tools for modeling, visualizing analyzing and implementing architectures, that we mentioned in Section §4.2.

In order to capture and provide a contextual model of the practice of product line archi-

tecture description and design, we extend the contextual model presented in Section §4.1 for architecture design, that we illustrated in Figure 4.1. In turn, that contextual model is an extension of that provided by the ISO/IEC/IEEE 42010:2011 standard for the architecture description practice, that we discussed in Section §3.1 and illustrated in Figure 3.1. We illustrate the contextual model for product line architecting in Figure 5.4. We conceive the *product line* itself as the actual system which architecture is being designed and described. The product line exhibits a *product line architecture*, which is the special case of *architecture* that targets product lines. The product line has a set of stakeholders holding concerns on the product line. One of this stakeholders is the *architect*, which is responsible for the product line architecture. The architects follows a *product line architecture design process* to design the product line architecture, which is expressed by means of a *product line architecture description*. The architect is responsible for creating and updating the product line architecture description, guided by the design process being followed. A *product line architecture description* is the special case of *architecture description* that are used in the context of product line development. Also, a *product line architecture description* is a *core asset* as it is intended to be reused during the construction of the architecture of products in the product line. As we discussed before in Section §5.1.1 and illustrated in Figure 5.3, a *core asset* has an *attached process* that guides the instantiation of the core asset. We use *product line architecture attached process* as the special kind of *attached process* for *product line architecture description* core assets.

It is important to notice that the contextual model illustrated in Figure 5.4 provides a first approximation to our conceptualization of product line architectures and the application of variability management to them. Later in Section §5.2, when we define our model-based approach to product line architecture description and design, we define a fine-grained correspondence between the internal characterization of architecture descriptions that we discussed in Section §3.1.1 and variability management studied before in Section §5.1.1.

5.1.3 Product Architecture

In the context of software product line development, the *product architecture* is the architecture of a particular product that is a member of the product line. The product architecture is constructed in the Application Design activity of the Application Engineering sub-process. The *product configuration* captured during Application Analysis serves the architect as the specification of the specific characteristics of the product line that are expected to be included in the product being developed. The architect uses the *product line architecture*, and particularly its attached process, as the guide for generating the *product architecture*. The generated product architecture reflects all the decisions regarding the selection of the variation mechanisms to produce the product-specific variants to be used and implemented during the product development. Such decisions are based on the particular product configuration desired. The product architecture has no further variability to be addressed. The architect can then refine the generated architecture so as to take into account product-specific requirements elicited during Product Analysis.

Figure 5.5 illustrates the contextual model for the practice of product architecture description and design. We extend the contextual model we defined before in Section §5.1.2 for

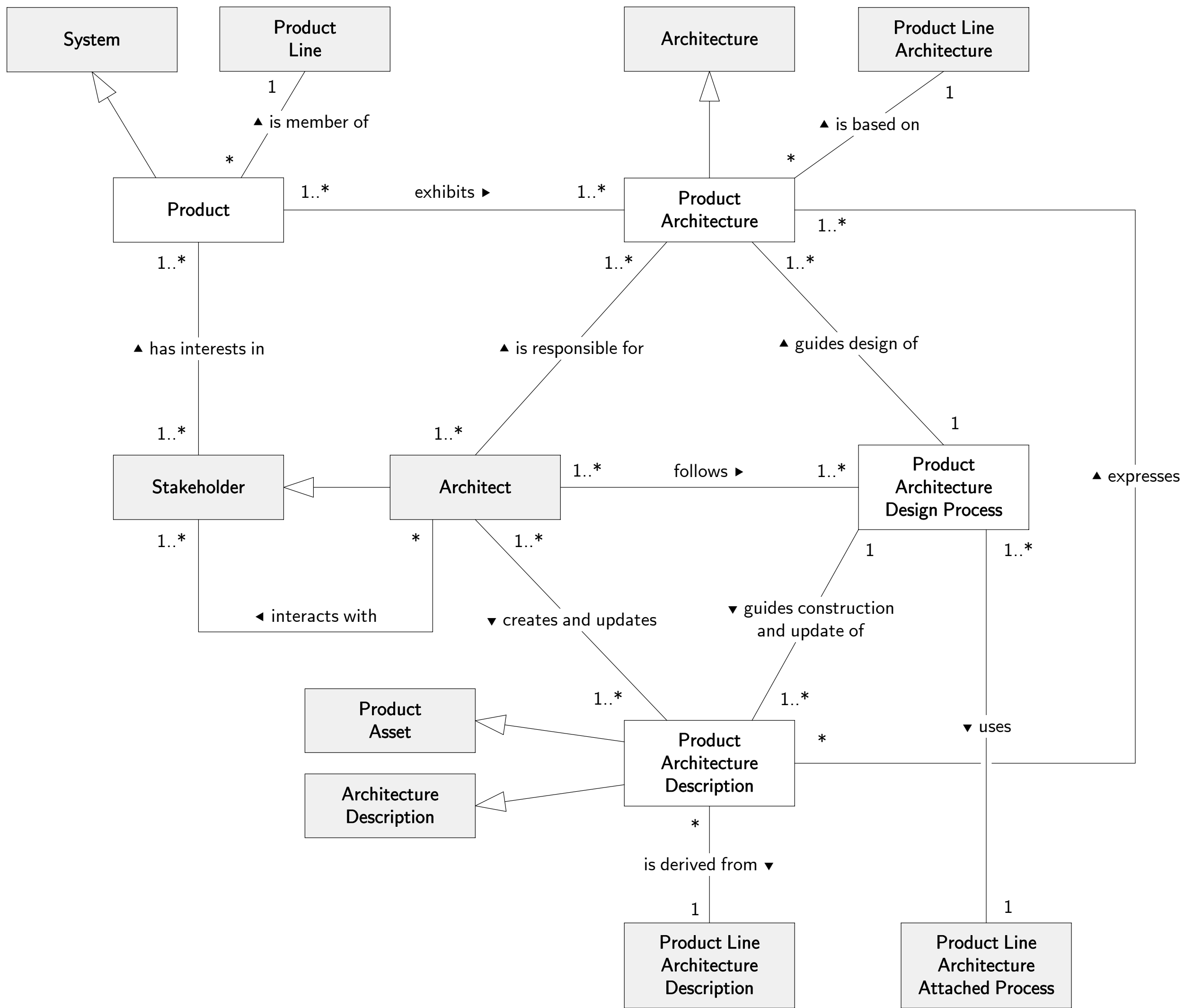


Figure 5.5: *Context of the product architecture practice.*

The figure illustrates the main concepts pertaining to the context of the practice of the product architecture description and design.

the practice of product line architecting. We conceive the *product* itself as the system which architecture is being designed and described. The product is actually a member of a *product line*. The product exhibits a *product architecture*, which is a special case of *architecture* and is based on the *product line architecture* of the product line that the products is a member of. Along the set of stakeholders holding concerns on the product, the *architect* is the one responsible for the product architecture. The architect follows a *product architecture design process* to design the product architecture and to capture the *product architecture description* that expresses that product architecture. The product architecture description is both a special case of *architecture description* and of a *product asset*. The architect creates and updates the product architecture according to the guidance provided by the product architecture design process. Such a process relies on the *product line architecture attached process* to guide that instantiation of the product architecture; such an instantiation is reified by means of the product architecture itself, which is actually the tangible artifact. Thus, the product architecture is derived from the product line architecture.

5.2 Model-Based Architecture in SPLs

Modeling techniques have pervaded the Software Product Line practice. Organizations have started to use them to effectively develop and manage software product lines [Jéz12]. By means of models and modeling techniques, not only both common and variable parts of a software product line design can be captured, but also derived work products and even entire products, members of the product line, can be generated from configuration models. Thus, using modeling techniques allows organizations to plan, specify, process and maintain software product lines on an abstraction level higher than that of implementation work products. While modeling variability allows an organization to capture and select which version of which variant of any particular aspect is wanted in the desired product [BFG+01], modeling techniques leverage modeling variability by enabling automation in the synthesis of work products and products [BFJ+03, BLHM02, CBUE02]. The ability of automation requires models to be no longer informal, but instead, to be captured in well-defined and machine-processable modeling languages. This automation makes the product derivation process cheaper, quicker, and safer, and avoids redoing manually the tedious effort of weaving different aspects of the system or asset into a unified work product.

As we discussed in Section §5.1.1, rigorously modeling variability can be achieved by following either an amalgamated or a separated approach [HMPO+08]. While the former promotes the extension of existing modeling languages to include variability-related constructs, the latter promotes independent modeling languages to capture variability together with a mechanism to establish the interconnection between variable parts and reusable assets. Once variability is modeled using a rigorous modeling language defined by any of these approaches, tools based on modeling techniques can be applied to process those models. J. M. Jézéquel in [Jéz12] classifies those tools in two categories: endomorphic and exomorphic. Endomorphic tools process variability models on their own, either for validation of self consistency or for composition and decomposition. For instance, M. Acher et al. in [ACLF11] use automated techniques to extract and report interesting properties from feature models. Examples of such interesting properties are whether they allow any valid configuration that satisfy all cross-feature constraints, whether there are dead features that do not participate in any valid configuration, among others. Exomorphic tools process variability models to generate other artifacts. As an example, compositional approaches are used at the implementation level where features are implemented in separated implementation units such as classes, packages or plug-ins, which are later processed by an exomorphic tool to compose them in different combinations to generate variants. For instance, M. Völter et al. in [VG07] apply aspect-oriented and model-driven techniques to this end. Another case of exomorphic tools are those providing test generation, such as in [NFTJ03]. A relevant application of exomorphic tools are those aiming product derivation, i.e. the process of constructing products from software product lines, based on the modeling of variability and the choices made in a product configuration [DSB04, GS07]. These tools assemble the derived products using pre-implemented components available as core assets to the product line.

Although variability and variability modeling has been primarily addressed in the context of software product line development, it imposes challenges on software development in general [CABA09]. Particularly, variability is a significant characteristic of the architecture of

software systems [GA11b, Hil10] which explicitly reflects and facilitates variability [BB01b]. As variability is pervasive, affecting the whole architecture and being a relevant concern for many different stakeholders, it is essential for the architect to have suitable methods and tools for handling it [GA11a]. As we reviewed and discussed in Section §5.1.2, several methods have been proposed for the architecture-centric development of software product lines. Most of these methods rely on modeling techniques to capture variability and architecture descriptions. Also, they use modeling techniques to process such models in order to derive additional work products. These methods tend to be structured in terms of the Model-Driven Architecture (MDA) [OMG03] approach, orienting variability management and product derivation to the generation of platform-specific models from platform-independent models built by the development team [GAWM11]. However, these methods restrict the architecting effort forcing a particular application domain, a particular architecture framework consisting of specific kind of architecture models, or a particular modeling language. Such a restriction is necessary to allow customized transformation to generate additional artifacts. As discussed by M. Galster et al. in [GA11b], most approaches focus on Component & Connector models, while variability in other architecture views or models, such as deployment, development and information models, are not addressed sufficiently.

Our model-based approach to architecture design dealing with variability management, mainly in the context of software product line development, has similarities to current approaches. Our approach is aligned to explicitly capturing variations in the architecture, as proposed by F. Bachmann et al. in [BB01b]. Also, it uses features and decisions to organize variations and conceive variability, as noted by M. Galster in [GA11b] as a common practice in the software product line domain. However, as opposed to current approaches, our model-based approach relies on explicitly capturing architecture decisions, the corresponding architecture solutions, and the architecture update scripts reifying these solutions. Thus, we follow the same underlying principle to architecture design that we defined in Section §4.2.1 for single-product architecture design: to explicitly capture how the architecture description is built and why it is shaped in that particular way. We do not rely on an explicit architecture description for the product line architecture. Instead, we rely on the different combinations of decisions – organized by means of features – that can be made to produce each and any product architecture in the software product line. Additionally, we are strongly committed to architecture descriptions adhering to the ISO/IEC/IEEE 42010:2011 standard, as we defined in Section §3.2. Hence, our approach does not restrict the architecture design effort to any particular framework or description language.

In this section, we first define our model-based approach to architecture design in the context of software product lines. Second, we extend our model-based formal semantics that we defined in Sections §3.2.1 and §4.2.2, to cope with architecting in this context, focusing on variability and reuse. To this end, we refine the definitions of our denotational semantic approach and we provide the semantic equations for the new concepts introduced in our conceptualization to architecting in software product lines.

5.2.1 Model-Based Approach

Successful reuse of knowledge, processes and products, in order to get the maximum cost-benefit ratio, depends on both the characteristics of the reused items themselves, and on the environment or context in which they are reused. According to V. Basili et al. in [BBJR87], the three main aspects to consider in the context of reuse are how to create items to make them reusable in the future, how to find appropriate reusable items, and how to integrate them to build a new artifact in a new development project. In their work, the authors also define a reusability framework that characterizes approaches for reuse. This framework consists of five key attributes:

- **Object:** what do we reuse?
- **Goal:** what is the target for the reuse?
- **Mechanism:** how do we accomplish reuse?
- **Timing:** when do we reuse something?
- **Support:** what assistance is there for access, modification, or porting an object?

In terms of this framework, reuse in our model-based approach is characterized as follows:

- **Object:** we reuse architecture decisions, their corresponding architecture solutions, both the selected one and its alternatives, and the architecture update scripts that materialize those solutions in terms of modifications to an architecture description.
- **Goal:** the goal is to automatically derive the architecture of any product in the product line. By reusing decisions, solutions and scripts, the achieved reuse is the particular set of decisions that must be made in any particular product member.
- **Mechanism:** we accomplish reuse by two complementary mechanisms. First, we use a homogeneous means for representing architecture descriptions, conceived in terms of the key concepts identified by the ISO/IEC/IEEE 42010:2011 standard on architecture description practice that we formalized in Section §3.2.2, and for representing the fine-grained architecture design actions, conceived in terms of the model-based architecture update scripting language that we defined in Section §4.2.3. The homogeneous means relies on modeling techniques both for representing architectures and to enact architecture design. As we defined in Section §4.2.1, our approach focus on capturing why + how, i.e. the rationale justifying the decisions that are materialized as architecture updates to the architecture description, rendering the what implicit, i.e. derivable from the execution of the how. Second, we use features to capture the decisions points that imply variability among the different product architectures. Thus, each feature is attached to a sequence of decisions that must be made in order to include in the resulting product architecture, the corresponding variant represented by the feature. Then, any valid configuration of the feature model that captures and organizes these features, conforms the ordered set of decisions that must be made, and, consequently, the architecture update scripts that must be exercised to produce the corresponding product architecture.
- **Timing:** we identify two key stages in an architecture design effort in the context of software product line development where reuse need to be considered. First, the product line architecture is designed in the *Domain Design* stage of the *Domain Engineering* sub-process that we discussed in Section §5.1. In this stage, common and variable parts

are identified and designed, and a feature model is used to organize the corresponding decisions that render the devised product architectures. Second, the product architecture of a particular member of the product line is designed in the *Application Design* stage of the *Application Engineering* sub-process. In this stage, the particular variant for the architecture is declared by means of a valid configuration of the feature model, and the architecture decisions, solutions and scripts attached to the selected features are enacted to produce the initial version of the product architecture. As the particular product envisioned by stakeholders and the architect may require additional functionality that is product-specific and hence that is not directly provided by the product line, the initial product architecture is further refined to meet all stakeholder needs. Then, reuse is planned when defining the product line architecture at the Domain Design stage, and it is achieved when deriving the product architecture for a particular member at the Application Design stage.

- **Support:** our approach is strongly based on modeling techniques for both capturing architecture descriptions and the fine-grained steps involved in architecture design. Modeling techniques, tools and environments, provide the corresponding tool support both to store, explore and visualize modeling artifacts, as well as to exercise the fine-grained steps as they are expressed in terms of operators available in modeling environment. As we discussed in Section §4.2.2, these operators are mainly the manipulation of modeling artifacts in the model repository of the modeling environment, and the execution of model transformations on these modeling artifacts. As we defined in Section §4.2.3, the formal semantics for our model-based interpretation maps the fine-grained steps specifically to these available operators.

Succinctly, in terms of the three main aspects identified by V. Basili et al. in [BBJR87], we create the reusable assets by capturing the architecture decisions and their corresponding solutions and updates scripts, we structure them in terms of features in a feature model that enables to find the appropriate decisions to reuse, and we integrate those decisions to create a product architecture by exercising the fine-grained steps captured by the update scripts of the decisions corresponding to the selected features. We explain our approach in detail next.

Conceptualization of Product Line Architecture Design

In Section §5.1.2 we studied the concept of Product Line Architecture and we extended the contextual model provided by the ISO/IEC/IEEE 42010:2011 standard for the architecture description practice, to define our contextual model for architecture description and design practice in the context of software product lines. We illustrated our contextual model in Figure 5.4. We understood a *Product Line Architecture Description* as the complex work product that expresses a Product Line Architecture, and that is designed by an Architect following a Product Line Architecture Design Process. For us, as illustrated in the figure, a Product Line Architecture Description is a specialization of both an Architecture Description and a Core Asset. Being an *Architecture Description*, in the sense of the ISO 42010 standard that we studied and formalized in Chapter §3, the Product Line Architecture consists of an aggregation of architecture views and architecture models, governed by system-independent architecture viewpoints and model kinds respectively, that captures and enforce inter-view

and inter-model relationships by means of correspondences and correspondence rules, and that captures the architecture decisions that renders such a description. As a consequence, we embrace the consensual and standardized conceptualization of the architecture description practice and hence we allow the full potential of current approaches for architecture description, such as those captured in architecture frameworks and architecture description languages, as well as architecture design mechanisms as architecture styles, patterns and tactics that we studied in Chapter §4. Thus, as opposed to current approaches, ours is not restricted to any particular architecture framework or architecture views such as Components & Connectors. Being a *Core Asset*, in the sense of variability management that we studied in Section §5.1.1, a Product Line Architecture Description is formed by a *Common Part* and a set of *Variable Parts* that are expected to be filled by specific *Variants* when building any particular *Product Asset*. In our case, while our Core Asset is the Product Line Architecture Description, our Product Asset is actually the Product Architecture Description of each and any product in the software product line. We illustrated the conceptual model for variability management in Figure 5.3.

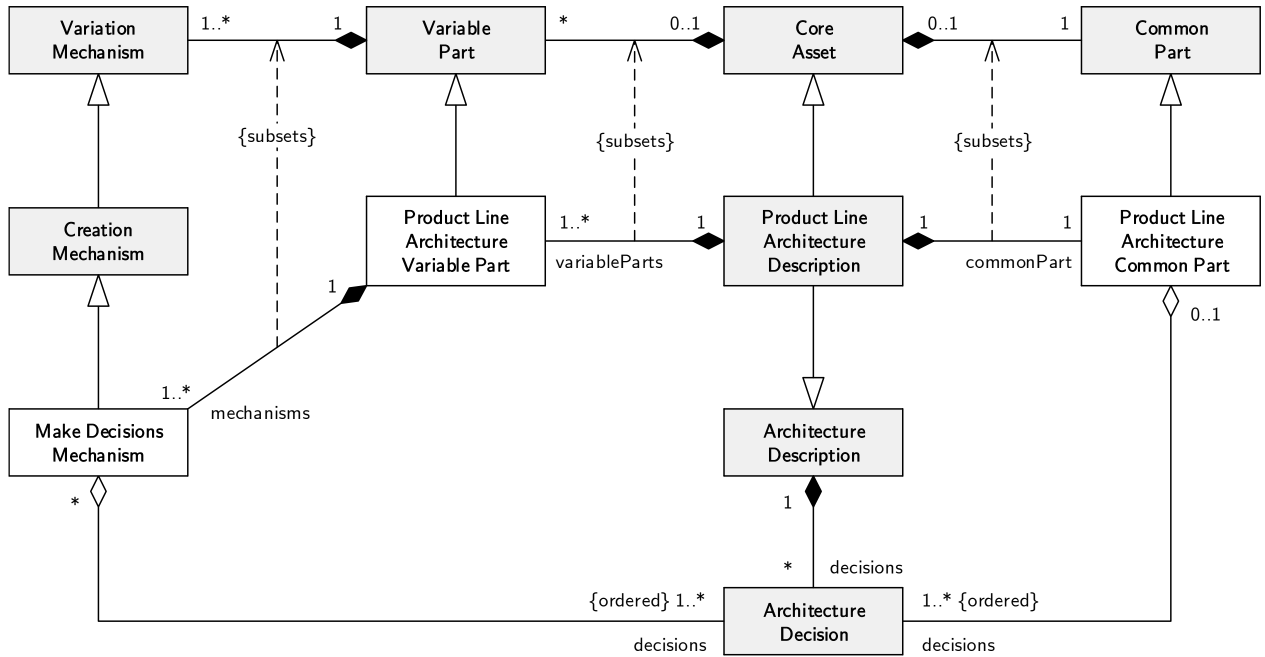
It is important to remark that the common and variable parts of the Product Line Architecture Description conceived as a Core Asset, are actually *parts* of the architecture description, i.e. they are conceived as organized following the same organization practices and rules defined for Architecture Descriptions in the ISO 42010 standard. Thus, the concepts from variability management and those from the architecture description practice overlap. This is reflected in current approaches where architecture models are extended to include variability constructs, like in amalgamated approaches, or that additional models are built to define the variability on architecture models, like in separated approaches. However, our approach relies on a different principle. As we defined in Section §4.2.1, our approach to architecture design focuses on explicitly capturing architecture decisions, their solutions and the reifying scripts, instead of explicitly capturing the architecture views and architecture models. Architecture descriptions are derived by exercising the scripts of the solutions of the decisions made. In the ISO 42010 standard, an Architecture Description contains an aggregation of Architecture Decisions to describe the key decisions made. As we discussed in Chapter §4, decisions are mainly descriptive, and thus, given an architecture description, we can (should, as in practice they are not always or completely captured) have the set of decisions made. Our architecture design approach reverse this relation. For us, architecture decisions are first prescriptive, i.e. they state and capture the architect's intention on how to update the architecture description to address a particular set of concerns or requirements. Then, architecture decisions are applicable as they can be exercised to actually produce these updates in the architecture description being built. As a result of making the decisions and performing the corresponding updates, the resulting architecture description reflects these updates and also captures the decision made. Hence, from the architecture decisions we obtain the resulting architecture description.

We follow the same approach in the context of product line architecture design. Instead of directly capturing the Product Line Architecture Description, the architect captures the set of architecture decisions that render such description. As we defined in Section §4.2.3, in the case of single-system architecture design, the architecture description is rendered by a sequence of decisions. However, in the context of product line architecture design, a sequence of decisions cannot capture the expected variability to produce all products in the product

line. Instead, we organize architecture decisions following the structural organization of core assets. To this end, we understood the *Common Part* of the Product Line Architecture Design as a sequence of architecture decisions, and we understood each *Variant* also to be generated by means of a sequence of architecture decisions. As we illustrated in Figure 5.3, *Variants* are obtained by applying *Variation Mechanism* attached to the *Variable Part* of the variant. Then, we define a purpose-specific mechanism, namely *Make Decisions Mechanism*, that captures the sequence of architecture decisions that must be made to produce a variant.

We illustrate in Figure 5.6 the conceptualization of variability in product line architecture descriptions. As we discussed before, a ProductLineArchitectureDescription is a specialization of ArchitectureDescription, and as such, it is a composition of ArchitectureDecision. These decisions conform the reusable *objects* that are used for deriving any particular ProductArchitectureDescription. Also, a ProductLineArchitectureDescription is a specialization of CoreAsset and as such, it is a composition of a single CommonPart and a set of VariableParts. We refine these concepts by defining purpose-specific specialization of them. Then, a product line architecture description is actually a composition of a ProductLineArchitectureCommonPart that captures a sequence of ArchitectureDecisions. Also, the product line architecture description is a composition of ProductLineArchitectureVariableParts that define the variation points of the architecture, and captures a set of purpose-specific variation mechanisms for producing variants for the ProductArchitectureDescriptions. In particular, we define the MakeDecisionsMechanism as the special case of CreationMechanism that captures the sequence of ArchitectureDecisions that creates or produces a particular variant for the variable part. While the purpose-specific common part captures the architecture decisions that produce the common structures and behaviors for the architecture of all products in the product line, the purpose-specific variation mechanism captures the architecture decisions that produce the corresponding updates to the architecture description to include in it the particular variant selected. In our approach, variants are not explicit. Rather, they are implicitly produced by the corresponding variation mechanism, which, in turn, is specified as an update to the architecture description being derived.

One of the topics analyzed in the 1st International Workshop on Variability in Software Architecture in 2011, reported in [GAWM11], was to conceive variability management as a decision problem. In the working group discussions held in this workshop, participants analyzed how to describe variation points by adapting the notion of decision. They identified similarities between decisions and decision alternatives (for us alternative solutions to choose from), and variation points (variable parts) and variants. Some of the concluding remarks of the discussion were that while every variation point is a decision topic, not every decision topic is a variation point. Also, provided the different levels of decisions, some of them constrain variability for later design phases when they are not managed as variability. As illustrated in Figure 5.6, our conceptualization actually provides an understanding or variable parts (variation points) in terms of decisions. We agree that every variable part implies a decision, either the inclusion or not of an optional variant, or the selection among alternative ones. As a consequence, our conceptual model reflects that a variable part is a decision point where a set of variation mechanisms can be applied. The selection of the corresponding mechanism is the decision to make regarding variability. It is important to remark that this kind of decisions which purpose is the selection of variants is not captured as an ArchitectureDecision is our conceptualization, but rather as the set of purpose-specific



context ArchitectureDescription
inv: self.decisions =
self.commonPart.decisions→union(**self**.variableParts.mechanisms.decisions)

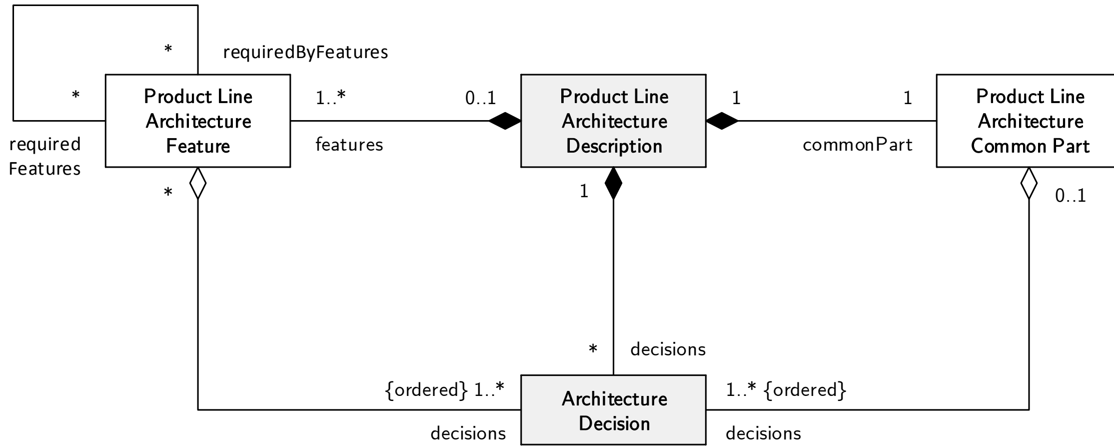
Figure 5.6: *Conceptualization of Variability in Product Line Architecture Descriptions.*
The figure illustrates the Product Line Architecture Description as a specialization of an Architecture Description, as understood by the ISO/IEC/IEEE 42010:2011 standard, and as a Core Asset as understood in the context of variability management. This conceptualization refines the contextual model presented in Figure 5.4 by defining the specialization for those concepts defining core assets that we illustrated in Figure 5.3.

variation mechanism MakeDecisionsMechanism. We also agree that not every architecture decision is actually a variation point. As we defined in our conceptual model in Section §4.1.3 each ArchitectureDecision may present a set of alternative solutions devised by the architect. Each alternative solution is just that, an alternative, that is discarded by the architect when deciding (selecting) a single one of them. In our conceptualization, we do not consider alternative solutions as variants of the product line. Hence, in our approach, when alternative solutions are not representing different ways to address the same problem, but rather different alternatives that can be used in different products of the product line, then such decisions and alternative solutions are actually promoted to the level of variable parts and variants, and captured as such. It is important to remark, also, that our conceptualization considers that each variant is produced by a sequence of architecture decisions, not a single one. Thus, clearly, for us, not every architecture decision is a variation point, as also stated in [GAWM11]. Besides, we agree that there are decisions at different levels that constrain variability when not explicitly managed. This is reflected in our conceptualization by the fact that a ProductLineArchitectureDescription contains a common part capturing the set of architecture decisions that impacts all products in the product line, and that restrict variability as the architecture decisions that we use to reify variants depends on the set of decisions previously made, mainly those captured as the common part. In [GAWM11] it

is also stated that while the understanding of the notion of decision to describe variation points could help when designing product families, it would not be equally helpful during product derivation. We claim that this is actually the case when architecture decisions are descriptive and captured along a manually built architecture description for the product line. However, making architecture decisions prescriptive and applicable by means of architecture solutions and scripts, as in our model-based approach, render the derivation of any product architecture automatic and hence, it benefits product developers as they can count on a product-specific architecture description based on the decisions made for the whole software product line.

The conceptualization for ProductLineArchitectureDescription that we illustrate in Figure 5.6 hides an important aspect of variability management that we have not discussed yet. As we reviewed in Section §5.1.1 and illustrated in the conceptual model in Figure 5.3, not only CoreAssets have variable parts, but also Variant. According to the conceptual model, each variant is defined as a composition of a single common part and a possibly empty set of variable parts, that are conceived in the same way as variable parts for core assets. In our conceptualization of product line architecture descriptions as core assets, variants are implicit as they are automatically generated by exercising the decisions of variation mechanisms. Hence, our conceptualization is still hiding how variable parts of variants are captured, and we cannot use variant themselves as our goal is to keep them implicitly defined, not explicitly. In our approach, a variant is reified by the sequence of updates to be performed on an architecture description. While the variants of the core asset (the product line architecture description) are reified by the sequence of updates to the architecture description produced by the common part of the core asset, variants of variants must be captured as updates to be performed on the architecture description in which the common part of the variant is already reflected. Then, each variant, independently of whether it is a variant of the core asset or a variant of a variable part of other variant, is produced by the sequence of updates to the architecture description being derived. As a consequence, there exists a dependency relationship between variants establishing that for any variant to be produced in the resulting product asset (the produce architecture description), all the variants on which which it depends must be previously produced in the resulting product asset. It is important to notice that this dependency already exists in the conceptualization for variability management illustrated in Figure 5.3. As shown in the figure, a variant is provided by a variation mechanism to fill a variable part in the resulting product asset. Such variable part can correspond to the core asset or to other variant. Then, the composition traversal from Variant to Variant in the conceptual model forces a relation dependency between variants.

In order to capture variants of variants and their interdependency, we refine our conceptual model in Figure 5.6 by introducing the concept of ProductLineArchitectureFeature to represent the distinctive mechanism that captures how to produce and include a variant of a variable part into a product architecture description being built. In terms of K. Kang et al. definition of *feature* in [KCH+90], we define a product line architecture feature as a prominent or distinctive stakeholder-visible aspect or characteristic of the architecture of a product in the product line. These features capture the decisions regarding variability on the product architectures. They are defined and managed by the architect of the software product line, and are selected by the architect of individual products of the product line. A ProductLineArchitectureFeature consists of an aggregation of the architecture decisions that must be made to include the vari-



context ProductLineArchitectureDescription
inv: **self**.decisions = **self**.commonPart.decisions→union(**self**.features.decisions)

context ProductLineArchitectureFeature
def: AllRequiredFeatures : Set(ProductLineArchitectureFeature) =
self.requiredFeatures→closure(f | f.requiredFeatures)

inv: **self**.AllRequiredFeatures→excludes(self)

Figure 5.7: *Conceptual model of Product Line Architecture Description.*
The figure illustrates the conceptual model for product line architecture descriptions using features to capture the mechanism for producing variants for any variable parts of the product line architecture. This conceptual model is a refinement of the conceptual model illustrated in Figure 5.6.

ant in the resulting product architecture. Besides, each feature may require a set of features to be previously including. In Figure 5.7 we illustrate our conceptual model that refines that illustrated in Figure 5.6. conceptual model. An ProductLineArchitectureDescription is composed of a single ProductLineArchitectureCommonPart and a set of ProductLineArchitectureFeature, both of them structurally organized as an aggregation of a the sequence of ArchitectureDecision. As we discuss later in Section §5.2.2, while we use a dependency association in the conceptual model, the semantic denotation for product line architecture features in terms of modeling artifacts is actually captured by a full-fledged feature model conforming to the metamodel that we defined in Section §5.1.1 and illustrated in Figure 5.2. It is important to remark that these features conceived to capture, organize and manage variability in the product line architecture is not necessarily the same set of features of the software product line developed during the *Domain Analysis* stage that we review in Section §5.1. The features in the product line architecture are identified and organized to reflect the architect's intention on how to resolve variability in the product line architecture. Nevertheless, it is expected that those features identified during Domain Analysis for the product line naturally propagates to the features in the product line architecture as the former represents variability offered or expected by external stakeholders, and the latter represents variability on the architecture level that is actually addressing the variability needs in the product line level. We consider both sets of features as separate so as not to force the architect into a particular organization

of architecture decisions that he made when designing the architecture of the product line in the *Domain Design* stage.

Conceptualization of Product Architecture Design

As we discussed in Section §5.1.1 and we illustrated in Figure 5.3, a *core asset* defines an *attached process* to guide the instantiation of *product assets* from the core asset. An attached process is performed using a set of *attached process steps*. Each step is qualified by means of a *condition* that states whether the step is applicable or participates in the production of the product asset. Also, each step indicates the *variation mechanism* that must be used when the step is performed. The order in which steps are applied strongly depends on the purpose and characteristics of the attached process itself. In terms of our conceptualization, as we have defined a *product line architecture description* as a *core asset*, we then need to define the attached process that guides the instantiation of product assets, particularly, the *product architecture description* of any product in the product line.

As we defined in our conceptualization, a *product line architecture description* is defined as a composition of ArchitectureDecisions with their corresponding architecture solutions along with the reifying architecture update scripts capturing the knowledge on how to update the architecture description in order to reflect the decision made. As we illustrated in Figure 5.6, we organize these decisions in terms of a *common part* and the *make decisions variation mechanism* attached to the *variable parts*. We introduced the concept of ProductLineArchitectureFeature to represent a particular *decision on variability*. Each of these features represents one realization of a variable part, either if it produces a variant for a variable part of another variant, or for a variable part of the architecture description as a whole. Technically, by means of these features we structure the reification of the variable parts of the architecture description, which is enacted by exercising the architecture update scripts that reify the architecture solution selected by the architecture decisions to be made to include the represented variant in the architecture description. We introduced an interdependence relationship between these features to enforce precedence in the order of the application of the corresponding scripts. We define the *attached process* to our *product line architecture description* as a sequence of steps, each of them associated to a particular ProductLineArchitectureFeature. Every valid configuration of these features renders a initial product architecture in which the architecture decisions of the selected features were made. Thus, during an architecture design effort of a product architecture, the architect provides the set of ProductLineArchitectureFeature that participate in the resulting product architecture being design. By selecting a feature, the architect is applying the corresponding variation mechanism, and as a consequence, is making the sequence of ArchitectureDecisions required to produce the variant into the resulting product architect description. The *condition* statement of each feature determines whether the feature is selected to participate or not, according to the selection made by the architect.

The order in which the steps are applied is important. While the condition of each step determines whether the feature is included or not, it does not impose any order in the application. Moreover, as we stated before, there is a precedence relation between the features and consequently, they cannot be applied in any order. It is important to recall that the architecture update scripts corresponding to a given decision assume a certain state

```
context ProductLineArchitectureFeature
  def: RequiredFeaturesInPostOrder : Sequence(ProductLineArchitectureFeature) =
      self.requiredFeatures→iterate(f; acc = Sequence{} |
          acc→merge(f.RequiredFeaturesInPostOrder)
      )

context Sequence(ProductLineArchitectureFeature)
  def: RequiredFeaturesInPostOrder() : Sequence(ProductLineArchitectureFeature) =
      self→iterate(f; acc = Sequence{} |
          acc→merge(f.RequiredFeaturesInPostOrder→append(f))
      )

context Sequence(T)
  def: merge(seq : Sequence(T)) : Sequence(T) =
      seq→iterate(e : T; acc = self |
          if acc→includes(e) then acc else acc→append(e) endif
      )
```

Table 5.1: *Conceptualization of the post-order traversal of required features.*

in the architecture description being built. These assumptions on the state are captured by the precedence relationship. For any feature in the product line architecture description, its required features determines the sequence of decisions that must be made before the decisions of the given feature are made. By this means, the architecture description is set to the necessary state in which the decisions of the feature can be exercised. Then, for any selected feature, not only its complete set of required features, transitively, must be also included in the product, but also, their corresponding architecture decisions must be made in the appropriate order so as to ensure the correct execution of the architecture update scripts of the architecture solutions. As we established in Figure 5.7 by means of an invariant, the precedence relation between features conforms a direct acyclic graph. We use a post-order traversal of this graph to determine the sequence of required features that must be included before a given feature is. In terms of our conceptualization, we use the auxiliary OCL declarations to specify the expected behavior of the post-order traversal. Table 5.1 provides these definitions. In the table, we define the RequiredFeaturesInPostOrder property for ProductLineArchitectureFeature that produces the sequence of features visited according to a post-order traversal. Also, we define a similar property on *sequences of features*. By this means, given a sequence of features $\mathsf{fs} \equiv \langle \mathsf{f}_1, \ldots, \mathsf{f}_n \rangle$, fs.RequiredFeaturesInPostOrder() yields the sequence of features required by every f_i in fs, including each f_i itself. Both definitions rely on a merge operation on sequences, that appends to a given sequence the elements of another sequence that are not yet in the result, preserving the order in the sequences. We provide the definition for merge in Table 5.1 as this operation is not part of the OCL Library provided by the OCL specification [OMG12].

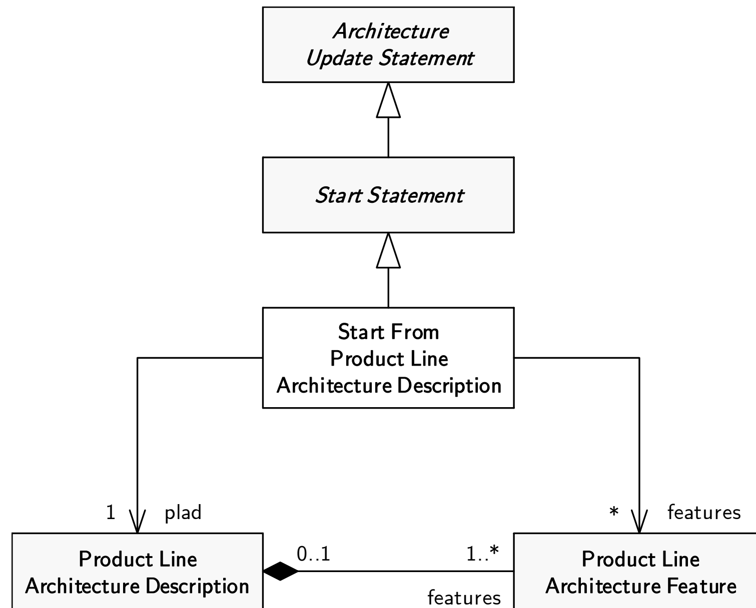
In practice, a product architecture design effort is performed in the *Application Design* stage of the *Application Engineering* sub-process, as we reviewed in Section §5.1.3. The *product configuration* captured during the *Application Analysis* stage serves the architect as the specification of the characteristics expected in the product being developed. The architect uses the product line architecture description, and its attached process, to generate

the initial architecture description for the product to build. In terms of our model-based approach to architecture design, the architect inspect the product configuration and the product line architecture description to decide the set of ProductLineArchitectureFeatures that must be selected and thus included in the resulting *product architecture*. Then, given a set of selected features sfs, the complete sequence of features to include, enumerated in the correct order of application, corresponds to fs ≡ sfs.AsSequence().RequiredFeaturesInPostOrder(). It is important to notice that the order in which the features are selected by the architect is not important; we represented this fact by using a set to express sfs. As we defined in Table 5.1, it is the responsibility of RequiredFeaturesInPostOrder to produce the correct ordering. Then, the initial product architecture is derived by first, making the sequence of architecture decisions corresponding to the common part of the product line architecture description, and second, by making the sequence of architecture decisions corresponding to each feature f_i in the order stated by fs. The resulting product architecture is in fact an initial version as additional design might be required for the product being built. In practice, additional requirements can be placed on products that were not consider as part of the domain of the software product line. Then, in this scenario, the architect proceeds following our model-based approach to architecture design in the context of single-product development, that we defined in Section §4.2. It is important to remark that the derived product architecture is structured as an ArchitectureDescription as we defined in Section §3.1.1, and hence, our approach to architecture design can be applied.

We conceptualize the derivation of the product architecture description from the product line architecture description as a special kind of *start* statement of our architecture design scripting language. As we defined in Section §4.1.2, a StartStatement allows the architect to take the initial step by creating the initial work products aggregated in the architecture description to be built. In this product architecture design, the initial work products are those derived from the product line architecture description by selecting the features that are included in the product being built. Figure 5.8 illustrates the conceptual model for the StartFromProductLineArchitectureDescription statement.

5.2.2 Semantics of Product Line Architecture Design

In Section §3.2 we formally defined our model-based interpretation of the key concepts pertaining the architecture description practice, in terms of modeling constructs available in the Model-Driven Engineering discipline, particularly in the Global Model Management approach to megamodeling. In Section §4.2.2 we extended our formalization to cope with the architecture design practice. First, we covered the interpretation of system-independent architecture design knowledge, such as architecture styles, patterns and tactics, that we conceptualized as composing parts of an architecture design library in Section §4.1.1. Second, we provided formal interpretation to architecture decisions and architecture solutions, as conceptualized in Section §4.1.3. Third, we provided formal meaning to our model-based architecture design scripting language, which provides architects a modeling language to capture fine-grained updates to architecture descriptions by means of statements to be enacted during an architecture design effort as we conceptualized in Section §4.1.2. We defined this formal interpretation by applying the denotational semantics approach [NN92, Sch86], consisting of the definition




```
context StartFromProductLineArchitectureDescription
  inv: self.plad.features→includesAll(self.features)

  inv: self.plad.commonPart.decisions→first()
         .solution.script.statements→first()
             .oclIsKindOf(StartStatement)
```

Figure 5.8: *Conceptual model of the* StartFromProductLineArchitectureDescription *statement.*
The figure illustrates the conceptual model for the special kind of StartStatement of our architecture design scripting language, that captures the starting point of the architecture design of a product architecture from a ProductLineArchitectureDescription and a set of selected ProductLineArchitectureFeature to be included in the product architecture.

of the semantic functions that maps constructs from a syntactic domain – the key concepts and concepts instances in the architecture description and design practice –, to constructs on a semantic domain – the modeling constructs and techniques available in the Model-Driven Engineering practice. We captured the mapping rules from the syntactic to the semantic domain by means of semantic equations which conform the definition of the semantic functions. In this section, we extend the definitions introduced in the previous chapters to provide formal semantics to the practice of product line architecture design.

Semantic Functions

In Section §3.2.1 (i) we defined the syntactic domain $\mathcal{A}$ as the union of the finite set $\mathcal{A}^{\mathsf{C}}$ of all concepts in the architecture description practice, as we conceptualized it in Section §3.1, and the infinite set $\mathcal{A}_{\mathsf{c}}$ of all conceivable concept instances of those concepts. In Section §4.2.2 (i) we extended the definition of $\mathcal{A}$, particularly of $\mathcal{A}^{\mathsf{C}}$ and $\mathcal{A}_{\mathsf{c}}$, to also embrace the key concepts that we identified in the conceptualization of the architecture design practice in Section §4.1.

We distinguished the particular subset $\mathcal{U}^{\mathsf{C}}$ of $\mathcal{A}^{\mathsf{C}}$ that includes only those concepts that, according to our conceptualization, present two aspects simultaneously: to be prescriptive and descriptive, and to be applicable or behavioral. Analogously, we distinguished the subset $\mathcal{U}_{\mathsf{c}}$ of $\mathcal{A}_{\mathsf{c}}$ that only includes the concept instances of those concepts in $\mathcal{U}^{\mathsf{C}}$. Also, we distinguished the subset $\mathcal{S}^{\mathsf{C}}$ of $\mathcal{U}^{\mathsf{C}}$ that includes all concepts corresponding to statements in our architecture design scripting language. The definitions for these sets conforming the syntactic domain are presented in Table 3.1 and Table 4.2. In order to cover architecture design in the context of software product lines, we refine these definitions so as to cover also the key concepts of our conceptualization introduced in Section §5.2.1. Thus, we redefine $\mathcal{S}^{\mathsf{C}}$ to also include the StartFromProductLineArchitectureDescription statement. Then, we also redefine the set $\mathcal{U}^{\mathsf{C}}$ to include this statement, and the set $\mathcal{U}_{\mathsf{c}}$ to include all conceivable instances of this concept. Additionally, we define the set $\mathcal{A}^{\mathsf{C}}$ as before, but also including the ProductLineArchitectureDescription concept. As a consequence, the set $\mathcal{A}_{\mathsf{c}}$ contains every conceivable concept instance, including those of product line architecture descriptions. It is important to remark that the new start statement concept and its instances are also part of $\mathcal{A}^{\mathsf{C}}$ and $\mathcal{A}_{\mathsf{c}}$ respectively, as $\mathcal{U}^{\mathsf{C}}$ and $\mathcal{U}_{\mathsf{c}}$ are subsets of those sets.

In Section §3.2.1 (ii) we defined the semantic domain $\mathcal{R}^*$ as the infinite set of model repositories containing modeling artifacts characterized by means of the Global Model Management approach. We defined $\mathcal{R}^*$ as the power set of $\mathcal{R}$ in Table 3.2, being $\mathcal{R}$ a model repository. We defined assertions to formally state the characterization of the modeling artifacts, their properties and their relations. Also, we defined the semantic operators that are available in the semantic domain, mainly for the manipulation of artifacts and for processing and updating the model elements defined within them. These assertions and semantics operators are thoroughly defined in Section §3.2.1 (ii) and Section §4.2.2 (ii). We use the same semantic domain, assertions and operators, for the formal interpretation of the product line architecture design.

In Section §3.2.1 (iii) we defined the semantic function $\mathcal{M} : \mathcal{A} \longmapsto \mathcal{R}^*$ that maps elements in the syntactic domain $\mathcal{A}$ to modeling artifacts within model repositories in the semantic domain $\mathcal{R}^*$. As stated in Table 3.7, the semantic function $\mathcal{M}$ is defined by case analysis, and relies on two auxiliary semantic functions $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$ that maps concepts in $\mathcal{A}^{\mathsf{C}}$ and concept instances in $\mathcal{A}_{\mathsf{c}}$, respectively. Consequently, when we defined the semantic equations for $\mathcal{M}$, we provided two equations, one corresponding to $\mathcal{M}^{\mathsf{C}}$ and one corresponding to $\mathcal{M}_{\mathsf{c}}$. The semantic function $\mathcal{M}$ serves the purpose of providing the prescriptive and descriptive representation for elements in the syntactic domain. The behavioral and applicable aspect of the concepts in the conceptualization is not addressed by this function. To this end, in Section §4.2.2 we defined the semantic function $\mathcal{D} : \mathcal{U}_{\mathsf{c}} \longmapsto (\mathcal{R}^* \hookrightarrow \mathcal{R}^*)$ that maps concept instances in $\mathcal{U}_{\mathsf{c}}$ to their effect on model repositories. Each concept instance $\mathsf{u} \in \mathcal{U}_{\mathsf{c}}$ is mapped to a partial function $\mathcal{D}[\![\mathsf{u}]\!] : \mathcal{R}^* \hookrightarrow \mathcal{R}^*$ that denotes the effect of u on a given model repository to produce another, in terms of the semantic operators available in the semantic domain. In the context of product line architecture design, we use the same semantic functions to provide formal denotation for the key concepts in our conceptualization. We define the remaining semantic equations next.

1	$[\![\mathsf{ProductLineArchitectureDescription} : \circ]\!]$
2	$\quad \stackrel{\text{def}}{=} [\![\mathsf{ArchitectureDecision} : \circ]\!] \;\cup$
3	$\quad\quad \{\mathsf{MMM}_\rho :_\bullet \mathsf{Metametamodel}, \mathsf{MgMM}_\rho :_\bullet \mathsf{Metamodel},$
4	$\quad\quad\; \mathsf{FMM}_\rho :_\bullet \mathsf{Metamodel}, \mathsf{FDWMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
5	$\quad\quad \mathsf{MgMM}_\rho \lhd \mathsf{MMM}_\rho \;\;\wedge\;\; \mathsf{FMM}_\rho \lhd \mathsf{MMM}_\rho \;\;\wedge\;\; \mathsf{FDWMM}_\rho \lhd \mathsf{MMM}_\rho$
6	$[\![\mathsf{plad} :_\circ \mathsf{ProductLineArchitectureDescription}]\!]$
7	$\quad \stackrel{\text{def}}{=} [\![\mathsf{ProductLineArchitectureDescription} : \circ]\!] \;\cup \bigcup_{\mathsf{d} \in \mathsf{plad.decisions}} [\![\mathsf{d}]\!] \;\cup$
8	$\quad\quad \{\mathsf{pladMgM} :_\bullet \mathsf{Megamodel}, \mathsf{cpMgM} :_\bullet \mathsf{Megamodel},$
9	$\quad\quad\; \mathsf{fM} :_\bullet \mathsf{TerminalModel}, \mathsf{fdWM} :_\bullet \mathsf{WeavingModel}\}$ such that
10	$\quad\quad \mathsf{pladMgM} \lhd \mathsf{MgMM}_\rho \;\;\wedge$
11	$\quad\quad \forall \mathsf{d} \in \mathsf{plad.decisions}.\;\; \mathsf{dMgM}_\nu \lesssim \mathsf{pladMgM} \;\;\wedge$
12	$\quad\quad \mathsf{cpMgM} \lhd \mathsf{MgMM}_\rho \;\;\wedge\;\; \mathsf{cpMgM}_\nu \lesssim \mathsf{pladMgM} \;\;\wedge$
13	$\quad\quad \forall \mathsf{d} \in \mathsf{plad.commonPart.decisions}.\;\; \mathsf{dMgM}_\nu \lesssim \mathsf{cpMgM} \;\;\wedge$
14	$\quad\quad \mathsf{fM} \lhd \mathsf{FMM}_\rho \;\;\wedge\;\; \mathsf{fM}_\nu \lesssim \mathsf{pladMgM} \;\;\wedge$
15	$\quad\quad \forall \mathsf{f} \in \mathsf{plad.features}.\;\; \mathsf{f}_\mu \lesssim \mathsf{fM} \;\;\wedge$
16	$\quad\quad \mathsf{fdWM} \lhd \mathsf{FDWMM}_\rho \;\;\wedge\;\; \mathsf{fdWM} \blacktriangleright \mathsf{fdMW}\langle \mathsf{fM}, \mathsf{pladMgM}\rangle \;\;\wedge\;\; \mathsf{fdWM} \lesssim \mathsf{pladMgM} \;\;\wedge$
17	$\quad\quad \forall \mathsf{f} \in \mathsf{plad.features}.\;\; \forall \mathsf{d} \in \mathsf{f.decisions}.\;\; \langle \mathsf{f}, \mathsf{d}\rangle_\mu \lesssim \mathsf{fdWM}$
18	$\quad\quad \nu(\{\mathsf{MMM}_\rho, \mathsf{MgMM}_\rho, \mathsf{FMM}_\rho, \mathsf{FDWMM}_\rho\}) \lesssim \mathsf{pladMgM}$

Table 5.2: *Semantic equations:* ProductLineArchitectureDescription.

Semantic Equations

In Section §3.2.1 (iv) we introduced the general form for the semantic equations of the semantic function $\mathcal{M}$, and in Section §4.2.2 (iv) we proceed analogously for the semantic equations of the semantic function $\mathcal{D}$. In Sections §3.2.2 and 3.2.3 we defined the semantic equations of $\mathcal{M}$ for the key concepts in the architecture description practice, and in Section §4.2.3 we complemented the semantic equations of $\mathcal{M}$ and we defined the semantic equations for $\mathcal{D}$ to address the key concepts in the architecture design practice. As we extended the syntactic domain to include the key concepts of architecting in product line development, we need to define the supplemental semantic equations that address those added concepts. It is important to remark that the semantic equations already defined are still valid.

Table 5.2 defines the semantic equations for the concept ProductLineArchitectureDescription and its instances, for the semantic function $\mathcal{M}$. As this concept is not behavioral or applicable, there is no corresponding semantic equation for the semantic function $\mathcal{D}$. The denotation of ProductArchitectureDescription : $\circ$ (1) is compositional on the denotation of the ArchitectureDecision concept (2). It is important to remark that as we have not included ProductLineArchitectureCommonPart and ProductLineArchitectureFeature concepts in the syntactic domain, we do not define semantic equations for them and consequently, the denotation of ProductArchitectureDescription not need to be compositional on their definition. However, it must be compositional on ArchitectureDecision : $\circ$, which was defined in Table 4.6–(1). We excluded those intermediate concepts from the syntactic domain so as to define a more compact denotation of ProductArchitectureDescription. The denotation also includes the modeling

artifacts providing the required infrastructure for the definition of the modeling artifacts for the concept instance. It includes the customized metametamodel MMM_ρ and the metamodel for megamodels MgMM_ρ (3). Also, it includes a customized metamodel FMM_ρ (4) that defines the modeling language for feature models. We use the metamodel illustrated in Figure 5.2 as the metamodel FMM_ρ. By this means, we provide the architect with a full-fledged characterization of ProductLineArchitectureFeature and their interdependency. The denotation also includes a customized metamodel FDWMM_ρ (4) for weaving models that captures the relationship between features and architecture decisions.

The denotation of plad $:_\circ$ ProductLineArchitectureDescription (6) is compositional on the denotation of its concept (7). Provided that a product line architecture description is a complex work product, its denotation includes the megamodel pladMgM (8) to capture the representation and characterization of all the modeling artifacts participating in the denotation of plad. The denotation is compositional on the denotation of all architecture decisions made in the construction of plad (7). Then, the representative megamodel dMgM of each architecture decision d is captured in pladMgM (11). By this means we capture the fact that a Product-ArchitectureDescription is a composition of ArchitectureDecision as illustrated in Figure 5.7. As also shown in the figure, architecture decisions are organized in terms of a common part and in terms of the features of the product line architecture description. Then, the denotation of plad includes a megamodel cpMgM (8) that captures the sequence of all decisions made by the common part (13). Architecture decisions associated to features are captured differently. We include in the denotation of plad a feature model fM (9) that conforms to the metamodel FMM_ρ (14) illustrated in Figure 5.2. The feature model includes the customized representation of each feature and their interdependencies (15). By this means, the feature model of the product line architecture description provides the structural organization of the variability of the product line architecture. The association between features and architecture decisions is captured by means of the weaving model fdWM (9) that conforms to the metamodel FDWMM_ρ (16). The weaving model realized the model weaving between the feature model fM and the pladMgM itself. While the former contains elements representing every feature of the product line architecture, the latter contains elements representing the megamodel dMgM of each architecture decision defined in plad. The weaving model then captures the links between a feature and the sequence of the decisions that must be made to include the feature in a derived product architecture (17).

As the StartFromProductLineArchitectureDescription statement is both descriptive and behavioral, its denotation must be defined in terms of both semantic functions $\mathcal{M}$ and $\mathcal{D}$. Table 5.3 defines the semantic equations for this concept and its instances. From the prescriptive and descriptive perspective, the StartFromProductLineArchitectureDescription statement requires the architect to indicate which product line architecture description to use, as well as the set of architecture features selected to be included in the product architecture. Thus, the denotation of s $:_\circ$ StartFromProductLineArchitectureDescription includes the representative megamodel pladMgM of the product line architecture description to use, and an annotation model that indicates which features of plad are selected. The annotation model is a weaving model fcWM (6) that realizes the model weaving on the feature model fM of plad and that contains a link to every selected feature (8). The feature model fM is part of the denotation of the product line architecture description plad being used (7), which is defined in Table 5.2–(9). As the denotation of a particular instance s of this start statement required a weaving

1	$[\![\mathsf{StartFromProductLineArchitectureDescription} : \circ]\!]$
2	$\stackrel{\text{def}}{=}$ $\{\mathsf{FCWMM}_\rho :_\bullet \mathsf{Metamodel}\}$ such that
3	$\mathsf{FCWMM}_\rho \lhd \mathsf{MMM}_\rho$
4	$[\![\mathsf{s} :_\circ \mathsf{StartFromProductLineArchitectureDescription}]\!]$
5	with $\mathsf{plad} \equiv \mathsf{s.plad}, \mathsf{fs} \equiv \mathsf{s.features}$
6	$\stackrel{\text{def}}{=}$ $\{\mathsf{pladMgM}, \mathsf{fcWM} :_\bullet \mathsf{WeavingModel}\}$ such that
7	$\{\mathsf{pladMgM}, \mathsf{fM}\} \subset \mathcal{M}[\![\mathsf{plad}]\!] \;\; \wedge$
8	$\mathsf{fcWM} \lhd \mathsf{FCWMM}_\rho \;\; \wedge \;\; \mathsf{fcWM} \blacktriangleright \mathsf{fcMW}\,\langle\mathsf{fM}\rangle \;\; \wedge \;\; \forall \mathsf{f} \in \mathsf{fs}.\; \langle\mathsf{f}\rangle_\mu \lesssim \mathsf{fcWM}$
9	$[\![\mathsf{s} :_\square \mathsf{StartFromProductLineArchitectureDescription}]\!]\, \varnothing$
10	with $\mathsf{cds} \equiv \mathsf{s.plad.commonPart.decisions},$
11	$\mathsf{fs} \equiv \mathsf{s.features.AsSequence().RequiredFeaturesInPostOrder()},$
12	$\mathsf{dfs} \equiv \mathsf{fs}\rightarrow\mathsf{collect(f \mid f.decisions)}\rightarrow\mathsf{flatten()}$
13	$\stackrel{\text{def}}{=}$ $\mathsf{R_{pa}}$
14	such that $\mathsf{R_{cp}} \equiv \mathcal{D}[\![\mathsf{cds}]\!]\, \varnothing \;\; \wedge$
15	$\mathsf{R_{pa}} \equiv \mathcal{D}[\![\mathsf{dfs}]\!]\, \mathsf{R_{cp}}$

Table 5.3: *Semantic equations:* StartFromProductLineArchitectureDescription *statement.*

model, its customized metamodel is then part of the denotation of its concept (2) that provides the infrastructure modeling artifacts. As we explained in Section §4.2.3, statements are captured by a terminal model and hence their structural denotation is not compositional to the denotation of its concept. Such composition is addressed by the denotation of ArchitectureUpdateScript itself as we defined in Table 4.8–(2).

From the behavioral perspective, the application of any StartStatement can only be done on an empty model repository, and hence it is the case for any instance s of StartFromProductLineArchitectureDescription (9). The resulting model repository conforming the effect of s is constructed in two stages. First, an intermediate model repository $\mathsf{R_{cp}}$ conforms the model repository generated by the common part of the product line architecture description. To this end, the sequence of architecture decisions of the common part of the product line architecture description in use by the statement s (10) are applied on an empty model repository to produce $\mathsf{R_{cp}}$ (14). It is important to remark that while the semantic equation uses an OCL expression to determine the sequence of decisions, the actual modeling artifacts can be obtained from the denotation of the product line architecture description plad in use by s. For instance, the denotation of plad includes a megamodel cpMgM in which all representative megamodels of the associated architecture decisions are recorded, as defined in Table 5.2–(8). Second, the sequence of architecture decisions corresponding to the selected features is applied on $\mathsf{R_{cp}}$ to produce the resulting model repository $\mathsf{R_{pa}}$. The selected features s.features are enumerated in post-order, as we discussed in Section §5.2.1, yielding fs (11). Then, the architecture decisions corresponding to the features in fs are concatenated preserving the order of the features and the order of the decisions for each feature, yielding dfs (12). Thus, the resulting model repository $\mathsf{R_{pa}}$ is obtained by making the decisions dfs on the model repository $\mathsf{R_{cp}}$ (15). As we explained before, even though we use OCL expressions to capture the decisions to apply and to order them, the corresponding modeling artifacts can be obtained from the denotation

of the product line architecture description, and the OCL expressions can be enacted by a model transformation that queries the feature model fM and the weaving model fdWM.

It is important to notice that the first architecture decision made is actually the first decision of the common part of the architecture description. As a consequence, the first statement processed is the first statement of the architecture update script reifying the architecture solution selected by that first architecture decision in the common part. This first statement is actually applied on an empty model repository, as stated in Table 5.3–(14). Then, the first statement must be one of the StartStatement defined in Section §4.1.2 so as it can be applied on the empty repository. This restriction is captured in Figure 5.8 by means of an invariant.

5.3 Contributions & Discussion

In this chapter we addressed the problem of opportunistic reuse of architecture decisions and architecture solution that takes place in the scenario of forward architecture design, as we discussed in Section §4.3. To this end, we analyzed architecting in the context of software product lines, which provides the scenario for *planned reuse* of architecture decisions and solutions.

The original contribution of this chapter is the contextualization and conceptualization of the architecture design practice in software product lines. We extended the contextualization of architecture practice defined by the ISO 42010 standard to position the concept of Product Line Architecture and Product Line Architecture Description. As we considered a Product Line Architecture Description as both an Architecture Description and a Core Asset, we then provide a conceptual model of Product Line Architecture Descriptions, using the variability management conceptual model presented by F. Bachmann et al. in [BC05]. Then, we refined this conceptualization so as to adopt proven practices in software product lines to capture variability, particularly, the use of feature models. We used a feature model to capture the interdependency of the variants for the variable parts of the Product Line Architecture Description, either for the description itself or for other variants. Moreover, we defined our model-based approach to architecture design of software product lines, extending our model-based approach to architecture design of single-products. By this means, we shifted the architect's focus from capturing variability within the architecture description, to capturing variability as the different combinations of decisions that can be made to produce product architectures. We used the feature model to provide structural organization to these architecture decisions and to simplify their selection when designing the architecture of a product of the product line. We originally proposed this approach in [PRB09, RPB09]. In this work we extended those results by providing foundational conceptualization to the approach, and also, formal semantics. In the latter, we extended the formalization the we have developed in Chapters §3 and §4 to cover also the key concepts of the practice of architecting in software product lines. To this end, we extended the syntactic domain to include the ProductLineArchitectureDescription concept and the StartFromProductLineArchitectureDescription statement that captures the knowledge on how to derive an initial product architecture description from the product line architecture description. Noticeably, this initial product architecture description not only is compliant to the ISO 42010 standard,

but also it can be further designed by means of our model-based architecture design scripting language that we defined in Chapter §4.

Model-based product line development. A notorious example of model-based approach to software product line development is defined by G. Perrouin in [Per07]. The author defines an model-driven product line development method, namely FIDJI, that allows developers to build products by transforming a coherent and layered set of product line models. The author defines four layers of models, Requirements, Analysis, Design and Implementation, and he called architecture framework a set of models organized in these layers to represent the core assets of the software product line. A set of model transformations "instantiates" these models for particular products. Constraints on the possible transformations have to be specified in order to determine which products cannot be derived both for functional and technical reasons. In that work, modeling techniques are used in their most classical way: models for representation, and transformations for generation of derived models. In our work, we do not use explicit models for representation. Instead, we use parameter models and models transformations to capture how to create the models for representation. By this means, we have a first class representation for decisions, alternative solutions and how these solutions are applied to build the architecture. Then, in the context of software product line, the selection of the core decisions, in addition to the selection of the decisions for each variation point, generates the product architecture automatically. Moreover, although we allow the use of models with embedded variability, we do not enforce it. Then, the product line architecture is implicit and the architect can generate any or all variants at any time, to make different kind of analysis from different perspectives, without the need to deal with special constructs for optional or alternative elements in models.

Challenges of variability in architecture. M. Galster et al. in [GA11a, GA11b] reported the problems and implications of dealing with variability in software architecture, identifying three main groups of challenges captured by means of a survey. The first group refers to the *complexity* of keeping variability simple and straight-forward, and of identifying what and how to change and vary. Second, the lack of *formality*, preciseness and rigor in the definitions of relations among variations as well as the validation of variability models. Third, challenges concerning *management*, which includes updating architectures, traceability of variability from feature models to architecture models, and consistency management and evolution. These identified challenges confirm the findings in the context of software product line reported by L. Chen et al. in [CAB10]. However, as noted in [GA11b], in contrast to the product line domain, the software architecture community has no common understanding of the nature of variability yet. Survey participants identified formal semantics for modeling and analysis, patterns, different architecture views, step-wise refinement, separation of concerns, and reference architectures, as potential counter measures to these challenges. The future direction suggested by the authors is to conceive a purpose-specific architecture viewpoint and view to capture and analyze variability-related aspects of the architecture. In our work, we take a *constructive* approach to architecture design. As current methods to variability in architecture design, mainly in the context of software product lines, we use features and decisions to capture variability. However, for us, they do not play a simple descriptive role in the architecture design process. Capturing features and decisions, along with the correspond-

ing architecture solutions and their implementing architecture update scripts, allows for the automatic derivation of any product architecture. Besides, as we discussed in Section §4.2.4, traceability information can be extracted for the resulting architecture descriptions, linking features to decisions, to architecture description elements. By this means, further analysis and comparison of the resulting architectures can be provided by additional tools – which can be implemented by means of modeling techniques – either for each individual product or for a set of products in the product line. We do not claim that our approach solves every challenge identified by the authors. However, our approach strongly relies on modeling for the representation of every key concept pertaining architecture descriptions, and promotes the usage of multiple architecture views and architecture models, all relying of well-defined metamodels captured by system-independent architecture viewpoints and model kinds, as we defined in Chapter §3. Also, our approach focuses on explicitly capturing architecture decisions and architecture solutions that are enacted (executed) to produce architecture descriptions. These decisions and solutions can be sequentially applied as we defined in Chapter §4 for single-system architecture design, or structured in terms of a feature model capturing those decision points that imply variability as we defined in this chapter for architecture design in the context of software product lines. In both cases, architecture design relies on step-wise refinement of the architecture description, which is actually built incrementally guided by the decisions applied. As a consequence, our approach promotes most of those techniques identified as counter measures by the survey participants. Besides, provided that our approach explicitly captures *how* the architect proceeds to build the architecture or product architectures, modeling techniques might be further used to analyze and evaluate properties in the design process itself, not only the resulting architecture descriptions.

Chapter 6

Applying the Model-Based Approach

The definition and construction of our homogeneous interpretation of architecture knowledge on architecture description and design in terms of Model-Driven Engineering (MDE) constructs, was performed in two steps. The first step consisted of defining a conceptual model of the practice, characterizing the principal constructs and their interrelationships. The second step consisted of defining a model-based approach for the practice, providing a formalization of its semantics in terms of the Global Model Management (GMM) constructs, the conceptual framework for medium- and large-scale applications of MDE that we discussed in Section §2.3. We followed these two steps incrementally. First, in Chapter §3 we defined our interpretation for *architecture description*. In Section §3.1 we reviewed and discussed the conceptual ISO/IEC/IEEE 42010:2011 standard, and we identified the adaptations that we considered necessary to capture the actual concepts pertaining architecture descriptions. In Section §3.2 we defined our formal semantics approach that provides a denotation for each construct in terms of GMM constructs, and we applied it to the conceptualization of architecture description. Second, in Chapter §4 we defined our interpretation for *architecture design*. Unlike the architecture description practice, there is no standard yet characterizing the context of the practice and providing its conceptualization. As a consequence, we defined in Section §4.1 our conceptualization of architecture design, based on the available literature, and we provided an architecture design scripting language to capture the fine-grained updates to architecture descriptions that reify the architecture solutions devised and applied by the architect during architecture design. In Section §4.2 we defined our model-based approach to architecture design and we formally specified its semantics in terms of GMM constructs and MDE operators. Then, in Chapter §5 we extended our conceptualization and formalization to cover architecting in Software Product Lines in order to provide planned reuse of architecture decisions and solutions.

In this chapter we explain how to apply our model-based conceptualization and formalization of the architecture description and design practice, and by this means, we provide validation and exemplification. To this end, we define systematic step-wise procedures to guide practitioners on how to capture system-independent architecture knowledge on the practice. First, in the context of architecture description, the procedure establishes the required steps to capture existing architecture knowledge in terms of architecture frameworks or architecture description languages. As we discussed in Section §3.1.2, these concepts represent system-

Chapter Contents

6.1 Capturing Architecture Knowledge **297**
6.1.1 Architecture Description Knowledge 298
Step I: Align architecture knowledge to the standard 298
Step II: Produce the infrastructure modeling artifacts 301
Step III: Produce the knowledge-specific modeling artifacts . . . 303
6.1.2 Architecture Design Knowledge 308
Step I: Identify the body of knowledge to operate on 309
Step II: Align architecture knowledge to the conceptualization . 310
Step III: Produce the infrastructure modeling artifacts 310
Step IV: Produce the knowledge-specific modeling artifacts . . . 311
6.1.3 Architecture Design Methods 312
Step I: Align artifacts and tasks to the conceptualization. 313
Step II: Produce the infrastructure modeling artifacts. 315
Step III: Refine the enactment of the decision making step. . . . 316
6.2 Meshing Tool Software Product Line **317**
6.2.1 Mesh & Meshing Tools 318
The Mesh Domain 318
Meshing Tools 320
6.2.2 Designing the Product Line Architecture 321
Capturing Variability 322
Designing the Common Part 324
Designing the Variable Parts 325
6.2.3 Deriving Product Architecture Descriptions 327

independent architecture knowledge. We exercise the procedure to illustrate how to capture part of the Views & Beyond approach to architecture documentation [CBB+10] defined by the Software Engineering Institute (SEI) of the Carnegie Mellon University. Second, in the context of architecture design, the procedure establishes how to capture system-independent architecture knowledge on architecture design by means of an architecture design library. We introduced this concept in Section §4.1.1 as the aggregation of architecture design mechanisms such as architecture styles, architecture patterns, and architecture tactics. This procedure can also be applied to capture system-dependent knowledge on how to address system-specific concerns or requirements by means of architecture design mechanisms. We exercise the procedure to illustrate how to capture a specialization of a SEI's Views & Beyond style, and an architecture pattern used to populate an architecture model with elements organized by means of the specialized style. Also in the context of architecture design, we define another procedure to guide practitioners on how to align our model-based approach to architecture design, to assist the decision-making activity of existing architecture design methods. As we

discussed in Section §4.2.1, our approach is not a full-fledged architecture design method, but rather, it is focused on the decision-making activity in which the architecture description is actually built. In this activity, our approach relies on explicitly capturing decisions and their corresponding solutions reified by scripts stating how to update the architecture description to reflect the decided solutions. We exercise the procedure to illustrate how to align our approach to SEI's Attribute-Driven Design method [SEI14a]. A second level of validation of our work is also illustrated in this chapter. We use the conceptualization and formalization that we defined in previous chapters, along with the particular instantiation of the architecture knowledge captured in this chapter, namely SEI's techniques, to design the product line architecture of the Meshing Tool Software Product Line. Also, we exemplify how our model-based approach, as defined in Section §5.2.1, can be used to automatically derive the product architecture of some particular products in this software product line.

This chapter is structured as follows. In Section §6.1 we define the systematic step-wise procedures to capture system-independent architecture knowledge on architecture description and design practice. Also, we exercise these procedures by capturing SEI's techniques for the practice. In Section §6.2 we discuss the architecture design effort of the Meshing Tool Software Product Line to illustrate, examine and validate the application of our model-based approach to architecture design in this context of software product lines. By this means, we also exemplify and validate the application of captured system-independent architecture knowledge to build a real-world software product line.

6.1 Capturing Architecture Knowledge

In order to make architecture knowledge explicit and homogeneously captured, rendering it shareable and reusable in multiple practitioner communities such as research teams, organizations and architect teams, we formally defined a mapping from the key concepts in architecture description and design to the set of modeling constructs and techniques available in the research field of Model-Driven Engineering. Particularly, this formal mapping, specified by means of the semantic function $\mathcal{M}$ that we defined in Chapters §3 and §4, rigorously states which are the set of modeling artifacts required to completely and correctly represent those key concepts. Then, the question arises of how to apply this formalization to capture actual knowledge.

In this section we define step-wise procedures to guide practitioners on how to accomplish such a goal. Naturally, we define these procedures by strongly relying on the semantic equations for the semantic function $\mathcal{M}$. First, in Section §6.1.1 we examine how to capture system-independent architecture knowledge in architecture description. We use SEI's Views & Beyond approach [CBB+10] as a running example. Second, in Section §6.1.2 we examine how to capture architecture knowledge on architecture design, particularly, on how to capture recurrent representation and problem-solution design mechanisms, such as styles, patterns and tactics. We use an application of SEI's Views & Beyond approach as a running example. Third, in Section §6.1.3 we examine how to align our model-based approach to architecture design, to assist architecture design methods. In this case, we rely on the semantic function $\mathcal{D}$ that we defined in Chapter §4 that maps behavioral constructs on the architecture design

practice to the effect of these constructs on the set of modeling artifacts conforming the representation of the architecture description being built. We use SEI's Attribute-Driven Design Method as a running example.

6.1.1 Architecture Description Knowledge

According to the ISO/IEC/IEEE 42010:2011 standard [ISO11], as we reviewed and discussed in Section §3.1, while an *architecture description* captures the architecture knowledge that is specific to the system which architecture is designed, *architecture frameworks* and *architecture description languages* capture system-independent architecture knowledge on how to structurally organize and express architecture descriptions. As shown in the conceptual models that we illustrated in Figures 3.4 and 3.5, both architecture frameworks and description languages conform an organization of *correspondence rules*, *model kinds*, and *architecture viewpoints*, characterized in terms of the *concerns* they addressed and the *stakeholders* that present an interest on them. Consequently, in order to capture a particular body of knowledge by means of our formalization, it needs to be aligned and conceptualized in terms of these concepts. Once this alignment is achieved, we can apply the corresponding semantic equations from our formalization, both for the concepts themselves, and for the instances of these concepts. In what follows we define the steps conforming the procedure that guides practitioners on how to capture architecture knowledge on software architecture description. We use SEI's Views & Beyond (V&B) approach as a running example.

Step I: Align architecture knowledge to the standard

The ISO 42010 standard defines how to organize architecture descriptions and architecture knowledge on the architecture description practice, particularly in terms of architecture frameworks and architecture description languages. As noted by the standard, examples of complaint architecture frameworks are The Open Group's Architecture Framework (TOGAF) [OG11], P. Kruchten's 4+1 view model [Kru95], Siemens' 4 views method [HNS99], Reference Model for Open Distributed Processing (RM-ODP) [ISO98], and Generalized Enterprise Reference Architecture (GERA) [ISO00]. In these cases, the practitioner should skip to **Step II** as no alignment to the standard is required. However, not every body of knowledge available in the literature is actually complaint to the standard. For instance, V&B defines a different set of concepts referring to the organization of architecture documentations. Particularly, V&B identifies three fundamental kinds of structures which are used to characterize and categorize the styles, and uses views governed by these styles to capture the architecture documentation of a system of interest. There is no straightforward correspondence of these constructs to those of the standard, as we review later. Then, in the cases where the constructs in the body of knowledge are not one-to-one aligned to those in the standard, this **Step I** is required to decide such an alignment. To this end, practitioners need to identify the key concepts in the body of knowledge being captured, and decide the mapping between these concepts and those from the standard, specifically to those defined in our conceptualization defined in Section §3.1. It is important to notice that, as more than one mapping can be conceived, practitioners need to decide the mapping the best suits their intention, goal, or working context.

The V&B approach [CBB+10] to architecture documentation does not use the same terminology as that of the standard. For V&B, an *architecture documentation* is a complex work product organized in a set of *views* and an additional section that captures the *documentation beyond views*. Each *view* represents a set of interrelated system elements, and can be structured and presented by means of *view packets*. The elements within the views follow one or more *styles* that define the type of elements that can be used, their properties, their interrelations, as well as constraints ruling their organization. V&B defines three fundamental structures, or *style categories*, and uses them to characterize and categorize several specialized *styles*. Style categories are Module, Component & Connector, and Allocation, for which V&B provides a definition of the main types of elements, their properties and relationships, at a higher level of abstraction that those of the styles. Then, each *style* provides a refinement of these constructs to address a particular design or documentation purpose. Even though V&B lacks a conceptualization model that would enable a rigorous mappings from V&B concepts to the standard's concepts, the authors review in [CBB+10, Section E.1] how V&B can be used to produce architecture descriptions that are compliant to the standard. By means of a table, the authors discuss the correspondences for concepts, mainly focusing on viewpoints, views, inconsistencies and rationale, but omitting the notion of model kind and architecture models that, according to the standard, are an integral part in the organization of architecture viewpoints and architecture views. Nevertheless, inspired in the reported correspondence, we decide the following mapping of the concepts. As we mentioned earlier, not only there is not a single mapping, but also the best mapping cannot be determined in isolation but rather in terms of a specific goal. Then, while we define one particular mapping, we also discuss some of the alternatives.

We consider that V&B *architecture documentation*, *views* and *view packets* are system-dependent work products used to document the different aspects of the architecture of a system. Particularly, we map *architecture documentation* to the standard's *architecture description*, and *views* to *architecture views*. In V&B, a *view packet* is "the smallest bundle of view documentation you [the architect] would show an individual stakeholder, such as a developer assigned to implement a small portion of the system or a customer interested in an overview." Then, in V&B a *view* is organized in terms of *view packets*, which are targeted to individual stakeholders. We do not consider a one-to-one relation between *view packets* and the standard's *architecture models* used to organize *architecture views*. An architecture model contains elements that capture a particular set of aspects of a system, and while different stakeholders may have an interest on an architecture model, they may not be interested in the whole architecture model, but rather on a fragment of it (e.g. a developer implementing only some modules) or a projection of it (e.g. a customer interested in an overview containing the top-level decomposition only). Then, for us, an architecture model can and should be processed before being presented to stakeholders. As we discussed in Section §3.2.4 when we analyzed the three-forces approach to architecture description, we consider the goal of the architecture model to be capturing the complete architecture design for certain aspects, and projections of architecture models to be used as targeted reports to stakeholders. As a consequence, we do not map *view packet* to any particular concept in the standard. We allow architecture models to fully capture a given set of aspects of the system, that then, by means of post-processing, can be used to provide stakeholder related documentation, for instance, the view packets that might be planned and targeted for individual stakeholders.

These previous concepts are system-specific, and hence, they are used during an architecture design effort to build the architecture of a particular system. However, V&B also provides system-independent techniques to be used to produce those system-specific ones. We consider *style categories* and their *styles* to be system-independent. By definition, these constructs are used to guide the construction of *views*, and the elements populating those views. This relationship between *style categories* (and *styles*) and *views* in V&B, is analogous to the *governing* relationship between *architecture viewpoints* and *architecture views*, and between *model kinds* and *architecture models* in the standard. As stated by both the V&B authors and the standard, *style categories* conform an implicit definition of an *architecture viewpoint*. Our mapping is aligned to this statement. However, the mapping of particular *styles* is not that straightforward. One may claim that each *style* renders a *model kind* that rules how to produce *architecture models* according to the particular style. By this means, we would have an *architecture model* that provides only one single organization of the system, and which elements are not the whole set of elements of a given type. For instance, consider the Component & Connector style category and the Client/Server style in this category. If we define a model kind providing constructs for the Client/Server style only, the architecture models for this model kind can only refer to server components and client components, that rarely conform the whole set of components within an architecture description. Hence, these architecture models do not capture all system's components, regardless of the different styles that are used to organize them. This is not good or bad in itself, it is just a consequence of mapping styles to model kinds, and it can be useful when the architect's intention is to produce small architecture models that actually resemble view packets. Notice that the number of *correspondences* between these models is large as each single component will probably populate multiple architecture models. However, if the architect's intention is to produce a single architecture model capturing all components in the system, then, instead of one model kind per style, a single model kind for the style category is required. In this case, the modeling language of this model kind is complex as it needs to allow architecture models to express not only the component and connectors, but also the component types and connector types. Additionally, constraints are require to validate the well-formedness of component instances with respect to component types, as both are defined within the same architecture model.

In addition to the structural aspects of the system, it is important to capture its behavior. V&B discusses in [CBB+10, Chapter 8] which aspects of behavior should be documented and how such information can be useful. Also, the authors review alternative notations to capture behavior. Most of the proposed alternative notations recommended rely on the Unified Modeling Language (UML), such as use cases, object interactions and state machines. We decide to capture behavior by means of a separate model kind in the architecture viewpoint that defines the kind of elements interacting in the behavior to specify. For instance, we decide to use a behavior model kind relying on state machines to specify the behavior of components. Then, we aggregate this model kind in the architecture viewpoint for the Component & Connector style category.

Finally, we consider the V&B approach itself as an *architecture framework* as it provides the conventions and common practices for capturing and representing architecture descriptions. We prefer an architecture framework to an architecture description language as the former has a broader scope than the latter, and V&B can be used to capture whole architec-

ture descriptions of systems in different application domains. Nevertheless, it is important to recall that in our formalization, we devoted a special effort to isolate the definitions of finer constructs from coarser ones. Hence, the mapping in terms of architecture viewpoints, and most specially in terms of model kinds, can be reused in other contexts, independently if they are captured altogether in an architecture framework for V&B or not.

Step II: Produce the infrastructure modeling artifacts

The semantic function $\mathcal{M} : \mathcal{A} \longmapsto \mathcal{R}^*$ maps concepts and concept instances of the conceptualization to modeling artifacts characterized in terms of the GMM approach to megamodeling. As we defined in Section §3.2.1, $\mathcal{M}$ is defined in terms of two semantics functions, namely $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$, mapping concepts and concept instances, respectively. The role of $\mathcal{M}^{\mathsf{C}}$ is to determine the set of modeling artifacts that provides the required infrastructure for the modeling artifacts produced by $\mathcal{M}_{\mathsf{c}}$. Infrastructure modeling artifacts are mainly those metamodels that are required for denoting instances of a particular concept, but that are independent of any specific concept instance being denoted. The semantic function $\mathcal{M}_{\mathsf{c}}$ is compositional on $\mathcal{M}^{\mathsf{C}}$ to ensure that the infrastructure modeling artifacts are available in the denotation of each instance. Then, we separate the application of $\mathcal{M}$ in two different steps, guided by its definition in terms of two separate semantic functions. By this means, practitioners can first focus on building the infrastructure, prior to deal with the particularities of each concept instance in the body of knowledge to capture. Thus, in **Step II** practitioners apply the semantic function $\mathcal{M}^{\mathsf{C}}$ to the top-most concept participating in the body of knowledge to capture. According to the compositional traversal that guides the definition of the semantic equations, $\mathcal{M}^{\mathsf{C}}$ is then recursively applied to all those concepts that are related to the top-most one. As a result, practitioners obtain the set of modeling artifacts providing the infrastructure for those modeling artifacts to be created afterwards in **Step III**. It is important to recall that the semantic equations for $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$ use customization functions to allow practitioners to determine how the modeling artifacts are actually defined. As we defined in Section §3.2.1, while the semantic equations establish what modeling artifacts are required, practitioners decide how they are internally shaped. As a consequence, **Step II** involves to make these decisions as well. The artifacts defined in this step can be reused when capturing another body of knowledge on architecture description. These artifacts do not depend on any particular characteristic of the body of knowledge being captured, but rather on the nature of what the practitioner is capturing. For example, in this step the practitioner decides first the underlying technology to use, and second, the constructs – by means of metamodels – to capture concerns, stakeholders, and documentation information for the main concepts of the conceptualization. The actual concerns, stakeholders, documentation information and modeling artifact particular to the body of knowledge are captured in **Step III**.

In the case of our running example, in **Step I** we decided to use an *architecture framework* to capture the body of knowledge of the V&B approach. Then, to proceed with **Step II**, we apply $\mathcal{M}[\![\mathsf{ArchitectureFramework} : \circ]\!]$ defined in Table 3.18–(1), to produce the set of infrastructure modeling artifacts required to capture architecture frameworks. Due to the compositional application of $\mathcal{M}$, we also have to apply $\mathcal{M}$ to Concern, Stakeholder, ArchitectureViewpoint and recursively on ModelKind, and finally to CorrespondenceRule. The

semantic equations for these concepts are defined in Tables 3.8, 3.9, 3.12, 3.10 and 3.15, respectively.

Many of the infrastructure modeling artifacts, which provide the foundation for the construction of the modeling artifacts of the concept instances, are technology-related. While our formalization is based on our technology-agnostic conceptualization of the GMM approach to megamodeling that we defined in Section §2.3, its application needs to be reified by means of specific technology. In Section §2.3.2 we discussed the available and current tool support for GMM. We use those technologies here. The customized meta-metamodel MMM_ρ is present in the semantic denotation of all the concepts to be mapped, and determines the modeling language for expressing metamodels. Any metametamodel might be used, and the decision is strongly related to the selection of the tool support for the modeling environment. We decide to use the ECore metametamodel so as we can rely and use the complete tool support of the Eclipse Modeling Framework, on which the AM3 realization of GMM is implemented. Also, we use AM3 metamodel for the customized metamodel for megamodels MgMM_ρ that is part of the denotation of the concepts to be mapped. Another technology-related infrastructure modeling artifact is the metamodel for transformation models TMM_ρ. Any model transformation language available in the modeling environment can be used. We use ATL as the model transformation language and hence, we decide TMM_ρ to be the ATL metamodel for transformation models. It is important to notice that our formalization does not force all transformation models to be captured using the same model transformation language, as we discussed in Section §3.2.2, and consequently, other metamodels, such as that for operational QVT, might be used in some cases. The denotation of the concepts to be mapped yields three metamodels for weaving weaving models, particularly, metamodels for weaving models linking stakeholders and concerns SCWMM_ρ, architecture viewpoints to concerns $\mathsf{AVPCWMM}_\rho$, and model kinds to concerns $\mathsf{ModelKindCWMM}_\rho$. We use the AMW tool for creating weaving models. As we discussed in Section §2.3.2, AMW defines a generic metamodel that can be extended to attach purpose-specific information to links between elements in different models. The three metamodels can be defined as a specialization of AMW's metamodel, or they can be defined as AMW's metamodel itself if we require no additional information to be captured. We decide to use the latter alternative to keep weaving models simple. As a consequence, we use an AMW weaving model as the weaving model SCWM that is also part of the denotation of the mapped concepts to capture links between concerns and stakeholders.

The denotation of the concepts also yields a metamodel CMM_ρ and a terminal model CM to capture the concerns of the architecture framework. The customized metamodel CMM_ρ can be either simple or complex, depending of the degree of detail that is expected to be captured for concerns. As concerns are usually conceived as a list, a simple metamodel with a single metaclass Concern would be enough to model concerns. However, we prefer to use a hierarchical representation of concerns, that can also have interdependencies. Then, our CMM_ρ metamodel contains a composition association for the hierarchical organization, and a simple association for the dependency relationship. It is important to notice that concerns must have an identifier to be used when merging concerns models. Also, the more complex the metamodel, the more complex the transformation CMerge_ρ that practitioners need to implement. We proceed analogously for stakeholders. We define SMM_ρ as a simple metamodel with a single metaclass Stakeholder, with an identifier, making it simple to implement the transformation model SMerge_ρ. As this transformation merges both stakeholders and

concerns, we implement it by delegating to CMerge_ρ the merging of concerns. The denotation also yields a customized metamodel $\mathsf{ProblemMM}_\rho$ that defines the modeling language to capture problems detected by constraints implemented as model transformations. Again, a simple metamodel containing a single metaclass Problem with identifier, description, details and severity attributes suffices.

The denotation of every complex concept includes a documentation metamodel that is used to capture the documentation information for concept instances. A metamodel $\mathsf{ArchitectureFrameworkDocMM}_\rho$ must be defined, as stated in Table 3.18–(5), to determine the modeling language for expressing the documentation of architecture framework instances. Analogously, we need to define documentation metamodels for architecture viewpoints, model kinds and correspondence rules. The standard establishes which information should be captured in the documentation of instances of these concepts. As we discussed in Section §3.2.2, this can be achieved with documentation metamodels that provide a precise domain-specific structure for the information to be documented, or by metamodels that provide a simple general structure mainly based on *text* attributes for the different sections of the documentation. The decision of which metamodel to choose is up to the practitioner community capturing the body of knowledge.

Step III: Produce the knowledge-specific modeling artifacts

The auxiliary semantic function $\mathcal{M}_\mathsf{c}$ is responsible for mapping instances of concepts in the conceptualization of the architecture description practice. As we explained in Section §3.2.1, the denotation of concept instances is compositional on the denotation of the concepts. The latter provides the infrastructure modeling artifacts to be used by the former. In **Step II**, we decided and built these infrastructure modeling artifacts. In **Step III** we define the modeling artifacts that are specific to the body of knowledge being captured. In this step we proceed similarly to the previous one. We apply the semantic function $\mathcal{M}_\mathsf{c}$ to the single instance of the top-most concept participating of the body of knowledge. According to the compositional traversal, the semantic function $\mathcal{M}_\mathsf{c}$ is also applied to the concept instances associated to the top-most one.

Let $\mathsf{vb} :_\circ \mathsf{ArchitectureFramework}$ be the architecture framework for the V&B approach according to the mapping that we decided in **Step I**. In this step we apply $\mathcal{M}[\![\mathsf{vb}]\!]$ as defined in Table 3.18–(9) to produce the set of modeling artifacts required to completely capture vb. According to the semantic equation, the denotation includes three groups of modeling artifacts: (a) the infrastructure modeling artifacts produced by the denotation of the concept, (b) the artifacts resulting from the denotation of the concept instances associated to vb, particularly the concerns, stakeholders, architecture viewpoints and correspondence rules, and (c) the artifacts that are particular to the denotation of architecture frameworks. The set of modeling artifacts (a) was developed in **Step II**. The set of modeling artifacts (c) includes a terminal model vbDocM for the documentation of the architecture framework, a representative megamodel vbMgM capturing all the artifacts participating in the denotation of vb, and a customized metamodel vbMM_ρ that defines the abstract set of constructs to be extended by every architecture viewpoint in vb. Then, practitioners need to fill the documentation model vbDocM, using the constructs defined by the customized metamodel

$\mathsf{ArchitectureFrameworkDocMM}_\rho$ defined in **Step II**, with the corresponding information extracted from the source publication of the body of knowledge, in this case [CBB+10]. This information is also used to populate the documentation models corresponding to the architecture viewpoints and model kinds defined in the denotation of vb. The customized metamodel vbMM_ρ can be created using the *style guide* defined in [CBB+10, Section I.2]. The style guide states that each *style category* and each particular *style* must be defined in terms of elements, relations and properties. Each style category specializes these constructs, and in turn, each style specializes the constructs defined by the style category. Then, we define the metamodel vbMM_ρ having three metaclasses, namely Element, Property, and Relation, where both Element and Relation have a composition of Property, and Relation has an nary association of Element. The style guide also indicates additional information to be captured for style categories and styles. Finally, the megamodel vbMgM is populated according to the assertions in the semantic equation in Table 3.18–(9).

The set of modeling artifacts (b) is produced by applying $\mathcal{M}_\mathsf{c}$ to the concerns, stakeholders, correspondence rules, and architecture viewpoints, defined by vb. Particularly, concerns and stakeholders do not produce additional modeling artifacts, as we defined in Table 3.8–(9) and Table 3.9–(15). Instead, they populate the terminal models CM and SM, and the weaving model SCWM, that we built in **Step II**. In the V&B approach, in addition to *views*, the architecture documentation contains one section to capture documentation beyond views. This documentation capture different aspects of the architecture itself, such as a roadmap, information on how views are documented, rationale, directory, and mapping between views. In the context of the standard, part of this information is captured by architecture viewpoints, as stated also in the correspondence provided by the authors of V&B in [CBB+10, Section E.1]. Other part of this information is captured in the documentation models for the architecture descriptions, for architecture views and for architecture models. It is the mapping between views that may render correspondences and correspondence rules in terms of the standard. We do not capture any specific correspondence rule to be included in our vb framework, although several of them might be required during an architecture design effort to capture specific relationships between the architecture models governed by the model kinds inspired by the *styles*. Then, the denotation of vb does not yield any artifact from the compositional application of $\mathcal{M}_\mathsf{c}$ to correspondence rules. Nevertheless, the architect can add custom correspondences and correspondence rules during the architecture design effort as appropriate, mainly using the corresponding update statements of our architecture design scripting language.

As we decided in **Step I**, we define three architecture viewpoints, one per *style category*. The resulting artifacts for the architecture viewpoints are produced according to the semantic equation in Table 3.12–(10). For each architecture viewpoint, we produce a documentation model using the information in the source publication, we produce a weaving model to identify the concerns framing the architecture viewpoint and we populate it according to the *what is for* section of the corresponding style guide, and we produce a representative megamodel and we populate it according to the assertions in the semantic equation. For each architecture viewpoint we develop a metamodel that defines the set of constructs that serve as the base for all constructs defined by the model kinds aggregated in the architecture viewpoint, as stated in Table 3.12–(15). Also, we develop the set of modeling artifacts corresponding to the denotation of the aggregated model kinds which were identified in the mapping defined in

Step I. These modeling artifacts are created by applying the semantic function $\mathcal{M}_{\mathsf{c}}$ according to the semantic equations for model kinds defined in Table 3.10–(1). We discuss Module and Component & Connector *style categories* as an example for **Step III**.

In **Step I** we decided that each style category is mapped to an architecture viewpoint. Then, in the case of the Module style category we have the moduleAVP architecture viewpoint defined in vb, which metamodel $\mathsf{moduleAVPMM}_{\rho}$ is defined according to the style guide for the style category. In particular, the metamodel defines a single metaclass Module and three associations to capture the relations is-part-of, depends-on, and is-a, defined in the style guide. However, as we stated in **Step I**, *styles* in a style category are not necessarily mapped one-to-one to model kinds in the architecture viewpoint. In the case of the Module style category, there are several styles that, while varying in the relation, use Module as the single type of element. This is the case for the Decomposition, Uses and Generalization styles. In this case, instead of defining one model kind for each of these styles, we can define a single model kind that provides one type of construct, namely Module, but that allows to capture all the relations between modules. This combined model kind allows architects to have a single and unified architecture model, avoiding the complexity of managing multiple architecture models with correspondences between them to match various representations of the same module in different models. In addition, in the examples of combined views provided in [CBB+10, Section 6.6.4], combining these styles is noted as a common practice. We decide to also combine them with the Layer style as layered architectures are a common mechanism to structure and organize the set of modules of a system. Then, we define the moduleMK in the architecture viewpoint moduleAVP and we apply the semantic function $\mathcal{M}_{\mathsf{c}}$ to produce the corresponding modeling artifacts, according to the definition of the semantic equation in Table 3.10–(14). The application of $\mathcal{M}_{\mathsf{c}}$ in the case of model kinds proceeds analogously as the application of $\mathcal{M}_{\mathsf{c}}$ to other complex concepts such as architecture viewpoint. One of the significant modeling artifacts in the denotation of moduleMK is its metamodel $\mathsf{moduleMKMM}_{\rho}$ that we define by a combination of styles. We illustrate the metamodel in Figure 6.1. The constraints established by the styles are captured by means of invariants in the metamodel. However, as we explained in Section §3.2.2, there are some constraints that must be selectable so as the architect can decide whether they apply or not in the particular architecture model being built. For instance, the fact that the allowed-to-use relation between Layers must be acyclical is captured as an invariant, as no architecture can have a circular dependency between Layers. However, whether this dependency is strict or relaxed actually depends on the architecture being designed. A strict organization forces the layers to be a sequence of layers, where one layer can only use the one immediately below. A relaxed organization allows any layer to use any of the layers below, not just the one immediately below. These two selectable constraints are captured by means of transformation models from $\mathsf{moduleMKMM}_{\rho}$ to $\mathsf{ProblemMM}_{\rho}$, as defined in Table 3.10–(19). The transformation encapsulates the knowledge on how to evaluate the terminal models conforming to $\mathsf{moduleMKMM}_{\rho}$ to check whether the constraint holds or not. By applying this transformation, the resulting problem model contains the elements represented the problems detected. Thus, the resulting problem model conforms the record of inconsistencies found in the architecture model.

In the case of the Component & Connector style category, we also have its corresponding architecture viewpoint defined in vb. The resulting modeling artifacts of applying $\mathcal{M}_{\mathsf{c}}$ to this architecture viewpoint are analogous to those of the Module architecture viewpoint. There

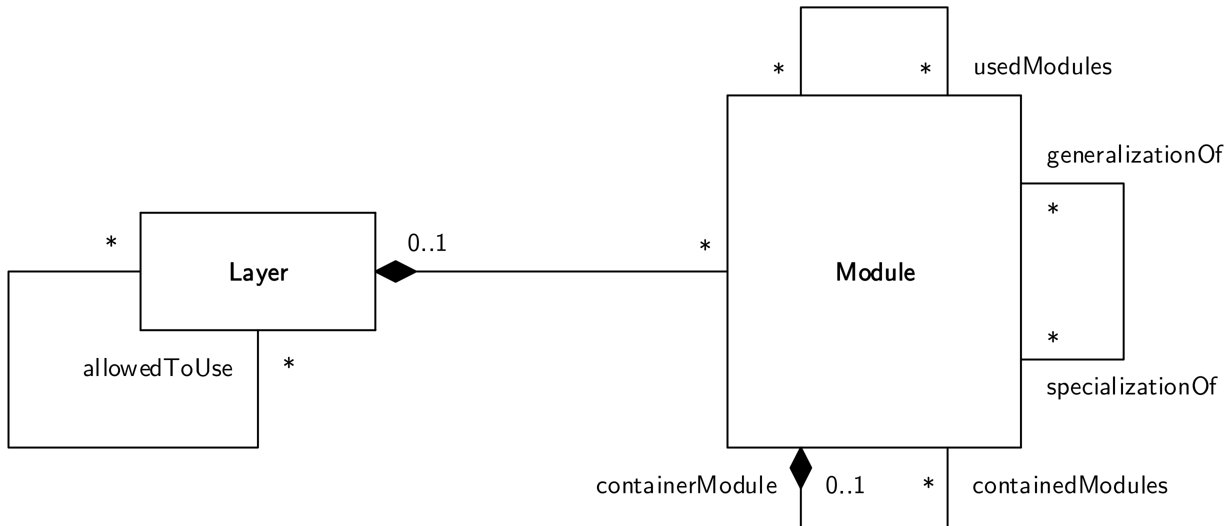


Figure 6.1: *Metamodel of the Module Architecture Viewpoint.*
The figure illustrates the metamodel of the Module Architecture Viewpoint that captures the key constructs defined by the Module category style of the Views & Beyond approach [CBB+10], unified with the specialized constructs of the Decomposition, Uses, Generalization and Layered styles of this category.

are two different approaches to define the metamodel for this architecture viewpoint. As opposed to the case of the Module style category, the distinction between component types and component instances, as well as connector types and connector instances, is very important as we can design the structural organization in terms of type of components interconnected, in terms or instances of components interconnected, or both. We identify two mechanisms to deal with this distinction. One mechanism uses model kinds to define the component types and connector types, and then we use architecture models to capture component instances and connector instances. In this scenario, the metamodel for the architecture viewpoint can be defined as illustrated in Figure 6.2. In the figure, the metamodel states that architecture models have components and connectors, and that they are attached by means of ports and roles. A model kind for each style in this category produces a specialization of this constructs. For instance, let consider the Client/Server style. As we map this style to a model kind, then, we have to define its metamodel by extending the constructs in Figure 6.2. We illustrate the resulting metamodel in Figure 6.3. In the figure, we introduce specialized constructs for components, connectors, ports and roles. Then, in the scenario of one model kind per style, during an architecture design effort, the architect produces a system-specific style, by extending those styles that provide the basic constructs. For instance, the architect can define a system-specific style defining a type of component that is both Client (from the Client/Server style) and Filter (from the Pipes & Filters style). Then, the metamodel of the new model kind defines this new component type than can then be used in architecture models governed by the model kind. In summary, while model kinds are used to capture component types and connector types, architecture models are used to capture component instances and connector instances. The other mechanism introduces in the model kind constructs to capture both types and instances. For example, we would have a model kind which metamodel defines the metaclass ComponentType and the metaclass ComponentInstance, and an association between

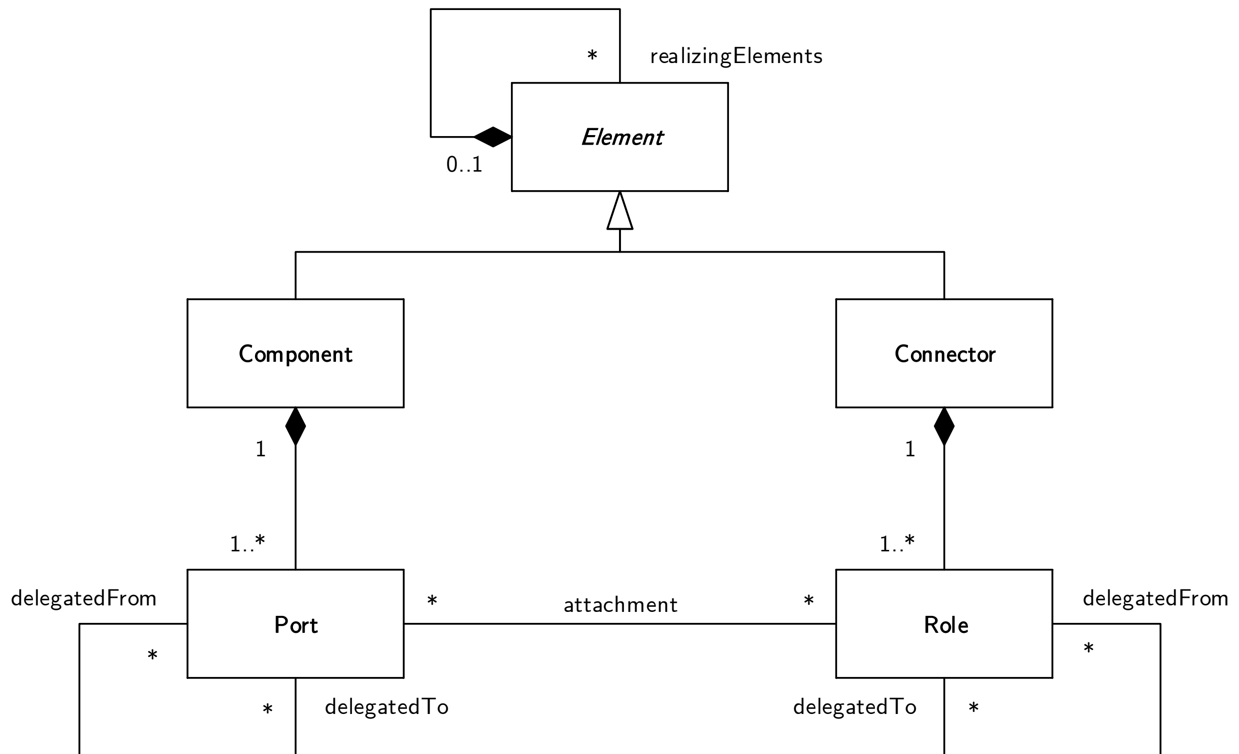


Figure 6.2: *Metamodel of the Component & Connector Architecture Viewpoint.*
The figure illustrates the metamodel for the Component & Connector Architecture Viewpoint that captures the key constructs defined by the Component & Connector category style of the Views & Beyond approach [CBB+10].

them to capture the typing relationship. In this case, a single model kind suffices to capture every style in the style category. Then, styles are captured by populating an architecture model with the corresponding elements of the metaclass ComponentType. This approach was followed by J. Ivers et al. in [ICG+04] when they discuss how to apply the Unified Modeling Language to capture Component & Connector views. We discussed these alternatives when we defined the ApplyStatement of our architecture design scripting language in Section §4.1.2. A lightweight practice is to avoid this distinction and directly rely on a Component & Connector metamodel, such as that illustrated in Figure 6.2. In this case, we have a single model kind that uses the same metamodel as the architecture viewpoint. By this means, we avoid creating several system-specific metamodels and keeps the single architecture model simple. However, the resulting architecture description is less descriptive as important information is omitted or delegated to textual explanations.

Finally, the denotation of the model kinds regarding to the behavioral aspects of the system yields the same modeling artifacts as any other model kind. In this case, we develop the metamodel for the model kind by extracting the corresponding segment of the Unified Modeling Language (UML) specification.

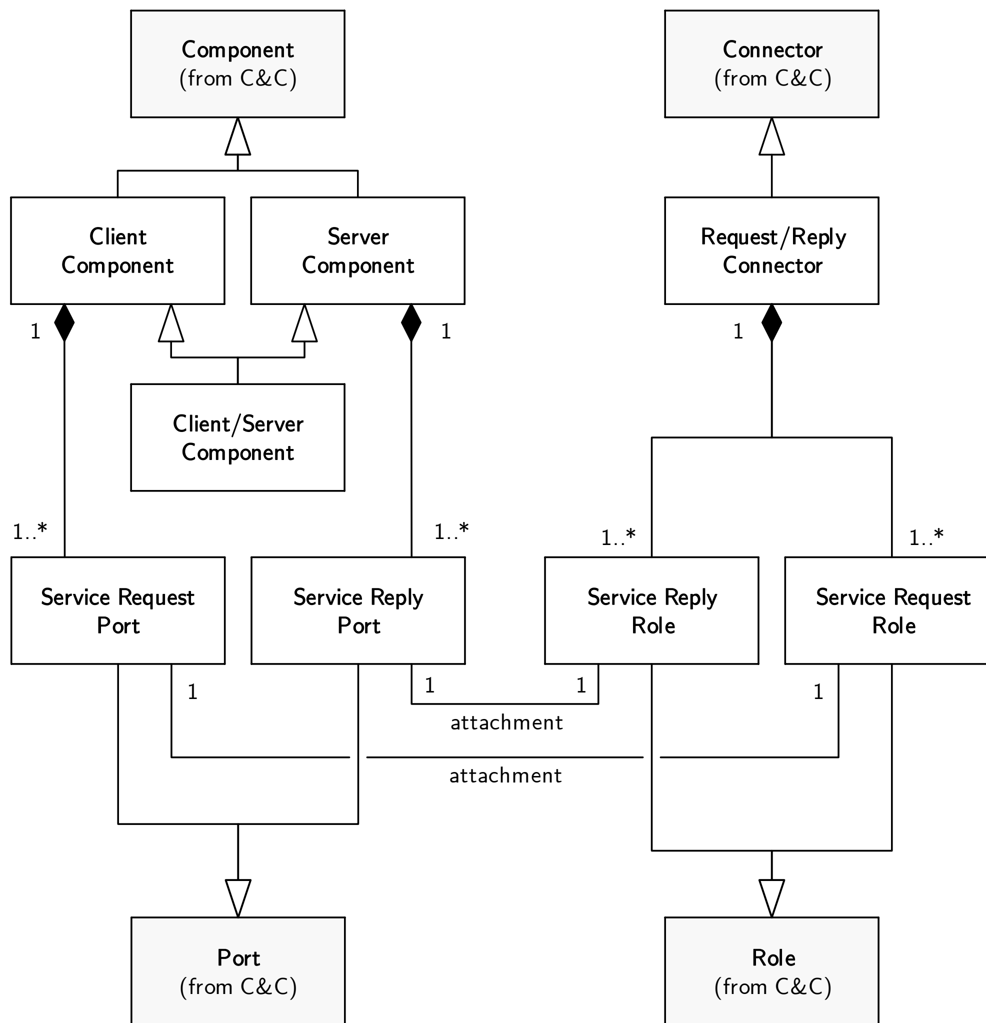


Figure 6.3: *Metamodel of the Client/Server Model Kind.*
The figure illustrates the metamodel for the Client/Server model kind as an extension of the metamodel for the Component & Connector architecture viewpoint illustrated in Figure 6.2.

6.1.2 Architecture Design Knowledge

In the context of architecture design, we capture system-independent architecture knowledge by means of an architecture design library, that we defined in Section §4.1.1 as an aggregation of architecture design mechanisms such as architecture styles, architecture patterns and architecture tactics. Architecture styles are defined by providing and extending model kinds. Architecture patterns and tactics are defined in terms of the model kinds and correspondence kinds they operate on, and on the correspondence rules they enforce. Also, patterns and tactics are parameterized so as to isolate their definition from any particular element populating architecture models. The system-specific elements are providing when applying the pattern or tactic during architecture design. Then, we classify this kind of knowledge as system-independent as they need not to depend on any specifics of the architecture being built and thus they can be reused in different architecture design efforts. As we discussed in Section §4.2.1, we use the same mechanisms to capture both generic and recurrent problem-solution patterns, and system-scoped problem-solution ones. However, even in

this case, patterns and tactics are defined in terms of model kinds and are parameterized. It is at the application of patterns and tactics during an architecture design effort, by means of the ApplyStatement of our architecture design scripting language, that the system-dependent knowledge in provided. In what follows, we define the steps of the procedure to guides practitioners on how to capture reusable system-independent architecture design knowledge, both recurrent and system-scoped, by using our conceptualization and formalization of the practice. To this end, we use the extension of the $\mathcal{M}$ semantic function that we defined in Section §4.2.3.

As a running example, we set our goal to create a specialized style of the Client/Server style in the Component & Connector style category defined by V&B, and to define a pattern that populates an architecture model using the constructs provided by this new style.

Step I: Identify the body of knowledge to operate on

An architecture design mechanism provides the capability of extending the set of architectural constructs that are available to the architect to build architecture models, and of manipulating the architectural elements in architecture models and correspondences to achieve a particular design purpose. In our conceptualization, architecture design mechanisms are defined in terms of the concepts available in the architecture description practice. As we illustrated in Figure 4.6, the architecture design mechanisms conforming an architecture design library *operate on* model kinds and correspondence kinds to determine what to extend or what to manipulate. As we studied in Section §6.1.1, a particular body of architecture description knowledge may not be aligned to our conceptualization, and it may have been captured in different ways as it is up to the practitioner to decide the most appropriate way to customize the application of our formalization. As a consequence, to capture system-independent architecture design knowledge, practitioners need to first identify the body of architecture description knowledge that is involved or impacted by the architecture design knowledge to capture. Second, they need to identify how such architecture description knowledge was actually captured in terms of customized modeling artifacts. The conceptualization of the particular architecture design knowledge, and the result of applying the formalization to it, strongly depend on the structure and internals of the architecture description knowledge they operate on.

In the running example, we want to create a specialized *style* of the Client/Server *style* in the Component & Connector *style category*. Then, in this step we need to identify how this style category and its corresponding styles were conceptualized when the body of knowledge defined by V&B was captured. As we defined in **Step II** in Section §6.1.1 we mapped the Component & Connector style category to an architecture viewpoint and we discussed different alternatives to capture the *styles* in the style category. For the purpose of this example, let consider that we followed the approach of mapping styles in the style category to model kinds in the corresponding architecture viewpoint. Now, as the result of this step with respect to the new style we want to create, we identify that the impacted architecture description knowledge is the Client/Server style which is mapped to a model kind. With respect to the new pattern, we identify that there is no impacted architecture description knowledge as we want our pattern to operate on models governed by the new style to create.

Step II: Align architecture knowledge to the conceptualization

The body of architecture design knowledge we want to capture may not be aligned to the conceptualization illustrated in Figure 4.6, or may not be directly stated in terms of the constructs identified in **Step I**. In this case, practitioners need to decide a mapping from the constructs in the body of knowledge to capture, to the constructs of our conceptualization of architecture design mechanisms. Altogether, they need to decide how the body of knowledge to capture can be understood in terms of the particular way that the impacted architecture description knowledge, identified in **Step I**, was conceptualized. In the case that the constructs are aligned, practitioners should skip to **Step III**.

In the case of our running example, the alignment of the knowledge to capture and the conceptualization is straightforward. The style to capture is mapped to an *architecture style* of the conceptualization, as defined in Figure 4.6, which provides a model kind defining the style-specific constructs for models governed by the style. The pattern we want to capture is mapped to an *architecture pattern* of the conceptualization, which operates on the model kind provided by the *architecture style* of the style to capture. Additionally, there is no misalignment between what we want to capture, and the way that the impacted architecture description knowledge was conceptualized. In particular, the architecture style extends the the model kind corresponding to the Client/Server style from V&B. Also, the pattern to capture operates on the particular model kind provided by the architecture style.

A conceptual misalignment would have presented in the case that the Component & Connector style category was defined by means of a single model kind that includes constructs to define component types and component instances in the same architecture model. This was one of the alternatives that we discussed in **Step III** in Section §6.1.1. In this case, we cannot map the style to an *architecture style*. Rather, styles are captured by populating the corresponding architecture model of the single model kind aggregated in the architecture viewpoint, creating the corresponding elements of the metaclass ComponentType in the architecture model. Then, the style must be mapped to an *architecture pattern*, as it involves the manipulation of elements within an architecture model governed by a particular model kind. Then, the result of this step in this case would have been to conceptualize the body of knowledge to capture as two architecture patters, one for populating the component types in the corresponding architecture model, and other for populating the component instances in the very same model. Thus, both architecture patterns would have been defined to operate on the same model kind.

Step III: Produce the infrastructure modeling artifacts

This step is analogous to **Step II** in Section §6.1.1. Provided that the semantic function $\mathcal{M}$ is defined in terms of two auxiliary semantic functions $\mathcal{M}^{\mathsf{C}}$ and $\mathcal{M}_{\mathsf{c}}$, we use two separate steps for the application of each of them on the body of knowledge to capture. In this step we apply $\mathcal{M}^{\mathsf{C}}$ to capture the infrastructure modeling artifacts. Later, **Step IV** is responsible for the application of $\mathcal{M}_{\mathsf{c}}$. The semantic equations for the denotation of system-independent architecture design knowledge were defined in Section §4.2.3, and particularly, Table 4.5 defined the semantic equations for ArchitecturePattern. Then, in this step, practitioners apply

the semantic function $\mathcal{M}^{\mathsf{C}}$ to the top-most concept when capturing an architecture design library, or to the set of concepts when capturing only the architecture design mechanisms.

If our goal was to define an architecture design library containing the two mechanisms we are capturing in our running example, in this step we would have to apply the semantic function $\mathcal{M}[\![\mathsf{ArchitectureDesignLibrary} : \circ]\!]$ which compositionally applies $\mathcal{M}$ to all concepts representing architecture design mechanisms, and particularly on architecture style and architecture pattern. As we are not defining an architecture design library, but simply capturing two specific architecture design mechanisms, in this step we apply $\mathcal{M}[\![\mathsf{ArchitectureStyle} : \circ]\!]$ and $\mathcal{M}[\![\mathsf{ArchitecturePattern} : \circ]\!]$ to determine the set of infrastructure modeling artifacts that we need to build. The resulting set of artifacts is similar to that created in **Step II** in Section §6.1.1. Then, we make the same decisions as before to determine the metametamodel, the metamodel for transformation and weaving models, and the metamodel for documentation and concerns. As we noted before, all the infrastructure modeling artifacts are highly reusable.

Step IV: Produce the knowledge-specific modeling artifacts

This step is analogous to **Step III** in Section §6.1.1. We use the auxiliary semantic function $\mathcal{M}_{\mathsf{c}}$ to determine the required modeling artifacts to capture the body of knowledge conceptualized as decided in the **Step II** of this procedure. As $\mathcal{M}_{\mathsf{c}}$ on a concept instance is compositional on $\mathcal{M}^{\mathsf{C}}$ on the corresponding concept, the infrastructure modeling artifacts defined in **Step III** are used here. Then, in this step we apply $\mathcal{M}_{\mathsf{c}}$ to the instance of the top-most concept, if defined, or to all the instances of the architecture design mechanisms to capture.

Continuing with our running example, let $\mathsf{newAS} :_{\circ} \mathsf{ArchitectureStyle}$ be the architecture style conceptualizing the style to capture, as we decided in **Step II**. Let $\mathsf{newMK} :_{\circ} \mathsf{ModelKind}$ be the model kind provided by newAS and let $\mathsf{csMK} :_{\circ} \mathsf{ModelKind}$ be the model kind corresponding to the Client/Server style of V&B. As we defined in Section §4.2.3, the denotation of an architecture style mainly consists of the denotation of its model kind, which must be defined as an extension of the model kinds extended by the architecture style. Then, the significant resulting artifacts for $[\![\mathsf{newAS}]\!]$ are those provided by $[\![\mathsf{newMK}]\!]$. The denotation of a model kind is defined by the semantic equation Table 3.10–(14). We proceed analogously as we did before in **Step III** to capture model kinds of an architecture framework. In addition to the documentation model, the weaving model on the concern model, and the representative megamodel, one of the most significant modeling artifacts is the metamodel of the model kind defining the modeling language for architecture models governed by the model kind. This metamodel is defined by extending the constructs provided by the metamodel of the Client/Server model kind csMK that we illustrated in Figure 6.3. The resulting metamodel can define specialized types of client components, server components, request/reply connectors, and also specialized types of the ports and roles defined by the metamodel in the figure. For the purpose of example, let us consider the Meshing Tool that we study later in Section §6.2. Particularly, we want to define two components, a Master Processor and a Slave Processor as specialized kind of client and server components respectively. The Master Processor delegates computation to one or more Slave Processors, exchanging infor-

mation through a purpose-specific connector to exchange data. The resulting metamodel is illustrated in Figure 6.4.

Let newAP :$_{\circ}$ ArchitecturePattern be the architecture pattern conceptualizing the pattern to capture, as we decided in **Step II**. newAP operates on the model kind newMK defined above. The semantic equations in Table 4.5–(10) defined the set of modeling constructs required to completely denote an architecture pattern. We proceeded as before by defining the modeling artifacts for the related constructs – particularly those for the model kind newMK are already defined, – and the modeling artifacts for documentation, the weaving models, and the representative megamodel. The most significant artifacts are the parameter metamodel and the transformation model. The parameter model, $\mathsf{newAPParamMM}_{\rho}$ according to Table 4.5–(16), defines the constructs for providing parameters to the transformation. The transformation model provides the implementation of the transformation that performs the corresponding updates to terminal models conforming to the metamodel defined by mk. Continuing with the example above, let consider that the goal of the pattern is to create an instance of Master Processor, and several instances of Slave Processor according to the information provided as parameter. Then, the parameter metamodel need to define a simple metaclass that contains the name of the master processor instance and the names of the slave processor instances. Then, the transformation model creates the corresponding elements of the corresponding types – component instances, connector instances, and the corresponding ports and role instances, – using the names provided by an input parameter model. Notice that in the resulting model, there will be a single instance of Master Processor but one or more instances of Slave Processor, according to the number of names provided. It is important to notice that this transformation is just an example. Once defined the metamodel for the model kind to operate on, multiple patterns can be defined, from fine-grained patterns that perform CRUD operations in terms of the constructs of the metamodel, to coarse-grained patterns that perform several operations at once. While the former are inspired in the syntactical constructs, the latter are inspired in their semantics.

6.1.3 Architecture Design Methods

The model-based approach to architecture design that we defined in Section §4.2.1 is not a full-fledged architecture design process, but rather, it is focused on the decision-making activity of such processes. However, our approach, along with its formalization in terms of the semantic function $\mathcal{D}$ that we defined in Section §4.2.3, can provide support for the enactment of architecture design processes, mainly for those process's tasks in which solutions are devised, decisions are made, and architecture descriptions are created and updated. Thus, in order to use our model-based approach to assist a particular architecture design processes, practitioners need to determine how the proposed conceptualization and techniques overlap the artifacts produced and consumed and the tasks performed according to the definition of the process. For those overlapping points, practitioners need to decide how to apply our techniques to improve the representation of the artifacts and the enactment of the tasks. In what follows, we define the steps of the procedure that guides practitioners on how to align our model-based approach to assist architecture design processes or methods available in the community. We use SEI's Attribute-Driven Design Method [SEI14a] as a running example.

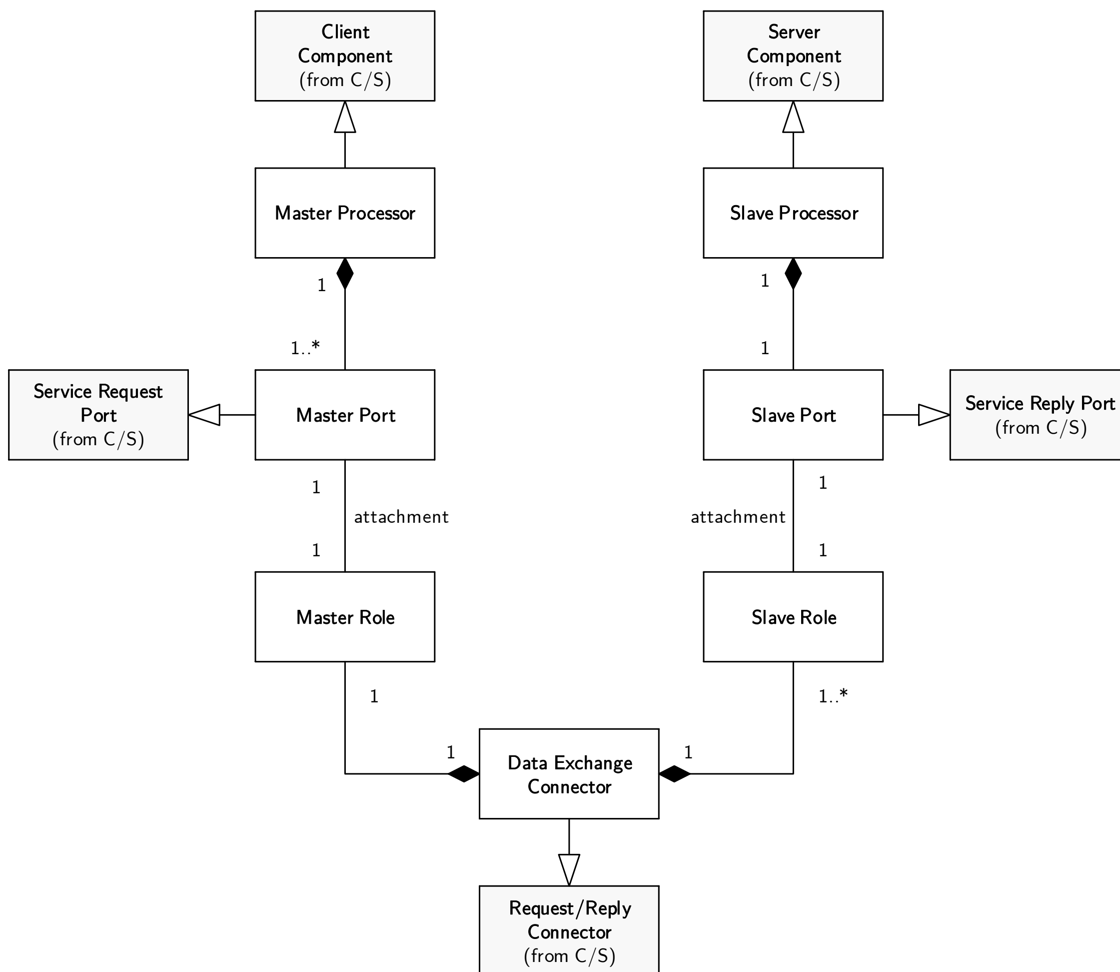


Figure 6.4: *Metamodel of a style specializing the Client/Server style.*
The figure illustrates the metamodel of a model kind of an architecture style that specializes the Client/Server style. The metamodel is defined as an extension of the metamodel illustrated in Figure 6.3, and defines specialized types for client components, server components, request/reply connectors, and specialized ports and roles.

Step I: Align artifacts and tasks to the conceptualization.

Although architecture design methods follows a general structure of three stages, as we discussed in Section §2.1.2, methods usually provide a finer definition of the tasks that must be performed and the artifacts that must be produced and consumed. In order to determine how our model-based approach to architecture design can assist a particular method, practitioners need to identify the overlapping points of both approaches. In this step, practitioners first identify the input, intermediate and output artifacts, and map them to the conceptualization. Provided the difference in scope between our approach and design methods, it is expected that some of these artifacts are out of the scope of our conceptualization and consequently, they cannot be mapped. Then, these artifacts with no mapping are treated as external entities in our formalization. Recall that in the context of the Global Model Management approach that we reviewed and refined in Section §2.3, artifacts are represented by the concept Entity, and there are two main kind of entities, namely Model and ExternalEntity. Thus, while artifacts with a mapping can be denoted in terms of modeling artifacts according to our model-

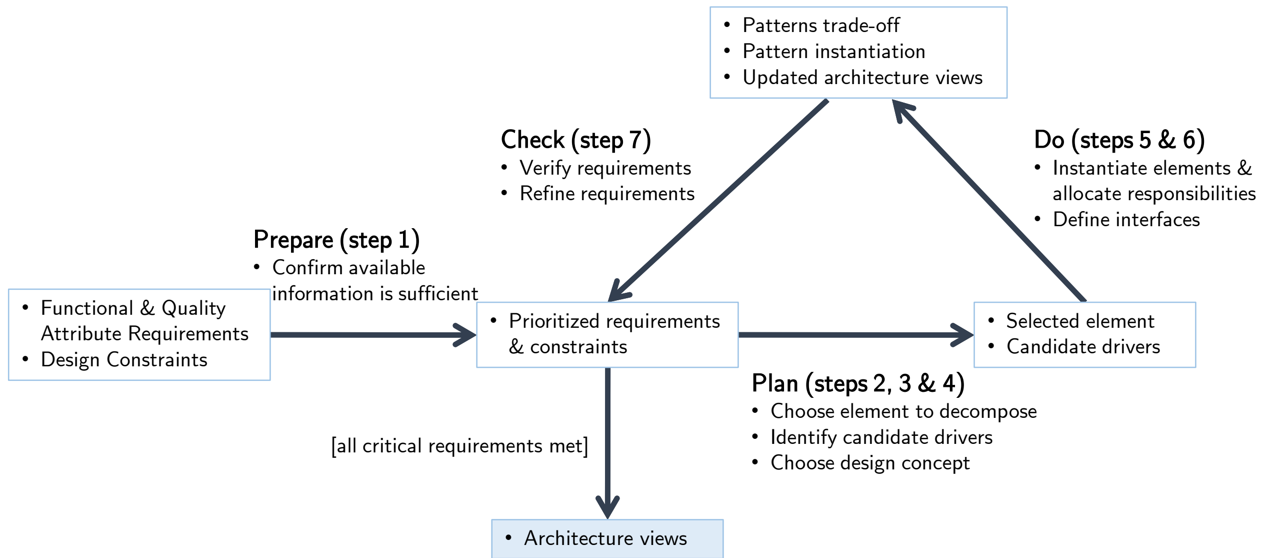


Figure 6.5: *Steps of the Attribute-Driven Design Method.*

The figure illustrates the main artifacts and steps of the Attribute-Driven Design Method, as defined in [WBB+06].

based interpretation, the artifacts with no mapping are denoted as ExternalEntity. Second, practitioners identify the tasks that conforms the architecture design methods in use, mainly those regarding the creation and update of the artifacts that were mapped to concepts in our conceptualization.

The Attribute-Driven Design (ADD) method is a systematic step-by-step method for designing the software architecture of a software-intensive system [SEI14a]. It follows a recursive decomposition process where, at each stage in the decomposition, tactics and architecture patterns are chosen to satisfy a set of quality attribute scenarios. ADD defines four main activities, three of them taking place iteratively. Figure 6.5 illustrates the process' activities, steps and main artifacts. The inputs of ADD consists of the set of functional and quality attribute requirements on the system of interest, along with the design constraints imposed on the architecture to be built. This input is processed and refined in the *Prepare* activity until the architect confirms that the available information is sufficient to start with the architecture design effort. The output of ADD is an architecture description captured in terms of views according to the V&B approach. In the *Plan* activity requirements and constraints are prioritized, design mechanisms are devised and weighted, and the most promising mechanism is selected. This is achieved by means of the *Candidate Patterns Evaluation Matrix* that crosses key architectural drivers and patterns. In the *Do* activity the decided pattern is instantiated, creating and updating architecture description elements and assigning them responsibilities. In the *Check* activity requirements are verified and refined. The iteration resumes with a new Plan activity when critical requirements remain to be addressed. The iteration ends when all critical requirements are met.

The input artifact is processed in the Prepare activity and is then prioritized at the beginning of the Plan activity. There are several alternatives for mapping requirements and constraints, from managing them outside the architecture domain, to use specialized views

for this purpose. In our running example, we decide to map the prioritized requirements and constraints to the *concerns* of our conceptualization. As we decide in **Step II** in Section §6.1.1, we consider concerns to be hierarchically organized from coarse- to fine-grained ones, being the bottom-most concerns the key requirements and constraints that must be met by the architecture. The output artifact is conformed by a set of views organized according to the V&B approach. This output is complete and detailed enough to address all critical concerns and the information needs of all stakeholders. However, it require further processing to produce human-readable documents from the views. In this sense, the ADD output is understood as an initial architecture documentation, as it needs to be subsequently refined to be communicated. For us, this distinction is favorable. The focus of the design method is to produce a representation of the architecture, delegating its communication to a separate effort. We discussed this issue before in Section §3.2.4. Moreover, ADD's output is organized in terms of views as defined in the V&B approach. As we decided in **Step I** in Section §6.1.1, we mapped V&B's architecture documentation, which is organized by a set of views and additional information beyond views, is mapped to *architecture description* in our conceptualization. Then, we map ADD's output to an *architecture description* and follows the mapping for system-dependent knowledge on architecture description that we discussed in **Step I** in Section §6.1.1. ADD's patterns are actually a combination of well-known architecture patterns, tactics, and architect's own inventions [WBB+06], that provide a particular architecture solution to address the selected concerns. We map ADD's patterns to our *architecture solutions* and they identifies the benefits and liabilities, along with the architect's intention on how to impact the architecture description accordingly. We map ADD's Candidate Patterns Evaluation Matrix as an *architecture decision* as conceptualized in Figure 4.20. For us, an architecture decision involves a set of alternative architecture solutions, selects the most appropriate and justifies the selection by means of rationale. Thus, ADD's drivers maps to our concerns, ADD's patterns to our solutions, and the cells for pros and cons maps to the documentation of the solutions and their involved concerns.

In the mapping we defined, almost every artifact has a corresponding concept in our conceptualization, and hence, they can be captured in terms of modeling artifacts by applying the semantic function $\mathcal{M}$. The application of this semantic function practically reifies all tasks in ADD as most activities and steps in ADD are focused on building the evaluation matrix, that is reflected on capturing architecture decisions and solutions according to our mapping. Also, we map the task of pattern instantiation to our reification of architecture solutions by means of *architecture update scripts*. Then, we map the task of updating the architecture views to exercising the script of the decided solution according to the semantic function $\mathcal{D}$ that states the effect of architecture update statements on architecture descriptions.

Step II: Produce the infrastructure modeling artifacts.

This step is analogous to that of previous procedures. The practitioners apply the auxiliary semantic function $\mathcal{M}^{\mathsf{C}}$ to capture the infrastructure modeling artifacts for all concepts involved in the architecture design method, as identified in **Step I**. This step is crucial as, in addition to technology-related artifacts, practitioners decide the constructs in the documentation models, concerns models and stakeholder models, that best fit the artifacts used by the design process being assisted.

In **Step I** we decided to map the output of ADD to *architecture description*. Then, in this step we apply $\mathcal{M}[\![\mathsf{ArchitectureDescription} : \circ]\!]$, which compositionally applies $\mathcal{M}$ to all other concepts pertaining the architecture description practice. As we decided that ADD's evaluation matrix is mapped to *architecture decisions*, in this step we also apply $\mathcal{M}[\![\mathsf{ArchitectureDecision} : \circ]\!]$ to determine the set of infrastructure modeling artifacts. Then, we first decide technology-related infrastructure modeling artifacts in the same way that we proceeded in **Step II** in Section §6.1.1. Second, we decide the metamodels to documentation models and weaving models to capture the necessary information for ADD to be enacted. For instance, the documentation model of *architecture solutions* must explicitly capture pros and cons, as they are used in the evaluation matrix.

Step III: Refine the enactment of the decision making step.

In **Step I** we identify in which part of the architecture design process takes place the decision-making activity. As the original specification of the method is not aware of our model-based approach, in this step practitioners refine this particular task of the design process to include the application of our techniques.

In ADD, architecture decisions are made during the *Plan* activity and the architecture description is updated during the *Do* activity. In terms of our formalization, the *Plan* activity produces the decision and the alternative solutions, marks the selected solution and defines the architecture update script for that solution. Let consider $\mathsf{d} :_{\square} \mathsf{ArchitectureDecision}$ to be one particular decision made, then, the *Do* activity uses $[\![\mathsf{d}]\!]_{\square}$ to update the architecture description according to the script attached to the selected solution by d.

The enactment of the decision-making activity by means of our model-based approach, conceptualization and formalization, can be summarized as follows. Architecture design begins by setting up a working environment with support for model-driven techniques, such as the Eclipse Integrated Development Environment with the corresponding plug-ins from the Eclipse Modeling Project [Ecl14b] and from the AM3 Project [Ecl14a] that we reviewed in Section §2.3.2. The first decision the architect makes is the starting point for the architecture design endeavor. The architect uses one of the *start* statements to create the initial architecture description, either from scratch, from a reference architecture, or from a predefined architecture description. Second, the architect decides the architecture frameworks and ADLs to adhere to, using the *structural statements* defined by our scripting language. Third, the architect follows the architecture design process to incrementally and iteratively design the architecture of the system of interest. To this end, new model kinds can be defined so as to capture system-specific styles, and new architecture models can be added to capture additional aspects of the system. The decision making step consists of defining the system-independent reusable architecture design mechanism in terms of an architecture pattern or tactic. Given a devised or required $\mathsf{p} :_{\circ} \mathsf{ArchitecturePattern}$, the architect uses $\mathcal{M}[\![\mathsf{p}]\!]$ to produce the required artifacts to represent it. Then, when the architect decides to use p as part of a *solution*, the architecture script uses the $\mathsf{ap} :_{\square} \mathsf{ApplyPattern}$ statement to define how to apply the pattern p on the architecture models aggregated in the architecture description. Then, the semantic function $\mathcal{D}$ is applied to the decision made on the current model repository $\mathsf{R_i}$, yielding a new version $\mathsf{R_{i+1}}$ of the model repository that reflects the updates

prescribed by the decision, its selected solution, and its corresponding script. The process ends when all critical concerns are met, and, as a result, the model repository contains all the modeling artifacts representing the architecture description, the architecture decisions made, the alternative solutions weighted, and the script stating how to produce the architecture description from scratch.

6.2 Meshing Tool Software Product Line

The Model and Transformation Engineering (MaTE) Lab of the Computer Science Department at University of Chile has been intensively working on the application of Model-Driven Engineering techniques to different areas of the Software Engineering discipline, such as software processes, software architecture, and software product lines. Since its conformation in 2006, master and PhD theses have been developed covering different aspects of these disciplines. Our work is part of this larger research effort. In the context of the MaTE research group, *meshing tools* have been addressed from various perspectives. From the implementation perspective, mesh generation and mesh quality improvement algorithms were developed. From the architecture perspective, an already existing implementation presenting poor maintainability was refactorized, its architecture was migrated to a blackboard style, improving its maintainability without loosing performance. From the software development perspective, the poor capability for integration and reuse of the developed tools lead to the emergence of a software product line for meshing tools. In this context, domain engineering activities have been performed, rendering a product line architecture that governs the architecture design and implementation of meshing tool products [RBHK+14]. Our work has been directly involved in the domain design activities of this research effort. We reported some of these results in [PRB09, RPB09].

The meshing tools software product line, studied and developed at MaTE, poses a great opportunity for the successful application of our model-based approach to architecture description and design. First, the domain analysis effort is already performed, and it is both available and thoroughly studied and understood by the research group. Second, previous development efforts have yielded well-identified architectural and implementation risks, that can be understood in quality attributes that are expected for the meshing tool products. Third, meshing tools present complex requirements, are inherently sophisticated, are used by people from diverse knowledge areas, and require high programming skills. Then, the architecture is a key artifact for dealing with this complexity and for guiding the diverse stakeholders involved. Thus, meshing tools suit our needs and expectations for practical experimentation and validation of our theoretical contributions.

In this section, we examine how our work can be successfully applied to address the architecture description and design of the meshing tool software product line. To this end, we characterize this validation as follows:

Goal: To provide the product line architecture for the meshing tools software product line that favors planned reuse at the architectural level.

Problem: Cope with the commonality and variability of meshing tools at the architecture level, facilitating the reuse of architecture decisions and architecture solutions in the architecture of particular meshing tool architectures. Variability is presented both at the functional level and at the quality attribute level.

Solution: Apply our model-based approach to architecture description and architecture design to the definition of the product line architecture of the meshing tools software product line. By this means, the variability model of the product line, expressed in terms of a feature model, is associated with the set of architecture decisions and solutions that are required to realize and satisfy each feature in the architecture. Thus, the product architecture of any particular meshing tool in the product line can be automatically derived from the core product line architecture and the set of selected features conforming a valid configuration of the feature model.

This section is structured as follows. In Section §6.2.1 we define the main concepts from the mesh domain and meshing tools, and we describe the critical functional and quality requirements along with the expected variability. In Section §6.2.2 we apply our model-based approach to architecture design in the context of software product lines, that we defined in Section §5.2.1, to design the product line architecture of the meshing tool product line. For the sake of clarity and brevity, in this section we focus on some aspects of the product line only, and we discuss those relevant aspects of the application of our approach. Section §6.2.3 concludes this chapter by illustrating the derivation of the product architecture description of two different products of the product line, from the product line architecture description and the selection of a specific set of features. Our focus here is to illustrate how we dealt with variability in the scalability quality attribute.

6.2.1 Mesh & Meshing Tools

The Mesh Domain

Modeling and simulating complex natural phenomena, scientific and engineering problems, and medical applications, among others, usually require a proper spatial discretization of the object to be modeled. Depending on the requirements of the mathematical model used to simulate the behavior of the object being modeled, triangles or quadrilaterals or both for two dimensions, and tetrahedra or hexahedra or both for three dimensions, are used as basic elements of the discretization. Thus, a *mesh* is defined as a discretization of a certain domain or object geometry, which is generally composed of a set of many small adjacent elements. Meshes are used for numerical modeling, and for visualizing and/or simulating objects or phenomena. Meshes are categorized in two kinds: structured and unstructured. The difference between them lies in the size of the elements: in structured meshes the elements are squares of equal size; in unstructured meshes there is no constraint on the size of the elements. Both kinds of meshes have advantages and disadvantages. Structured meshes are easy to generate and manipulate, use simple data structures reducing programming complexity and require less computer memory at runtime. However, structured meshes are limited to simple domains, so the use of unstructured meshes is inevitable in

the solution of complex problems with a more sophisticated domain geometry or shape. Unstructured meshes offer more convenient mesh adaptability and a better fit to complicated domains or objects with complicated topologies. However, some aspects of the theory behind unstructured meshes are not as well understood as those for structured meshes.

Mesh generation is the discipline that studies the discretization of domains, the process to generating such discretizations, and their application in other fields of engineering. For the International Society of Grid Generation [GG14], mesh generation "is that discipline of applied science and engineering which is devoted to the discretization of fields associated with computational analysis of scientific and engineering problems encountered in fluid mechanics, heat and mass transfer, aerospace and mechanical engineering, biomedical engineering, geophysics, electromagnetics, semiconductor devices, atmospheric and ocean science, hydrodynamics, solid mechanics, civil engineering related transport phenomena, and other physical field problems involving sets of partial differential equations formulated in a continuum." Such definition illustrates major fields of application. Currently, mesh generation plays an important role in finite element methods, biomechanics, virtual surgery and geometric modeling, computer graphics, geometric modeling, geographical information systems, terrain modeling and real time rendering.

A *mesh generation process* is a process that guides and defines how to produce or generate a mesh. The main steps of any mesh generation process are:

- read the geometry and physical values of the object to be modeled specified in some input format;
- generate the first mesh that fits the object geometry;
- refine the mesh to allow to increase the number of points and elements in some regions of the object, by generating smaller elements through the partition of the original ones in the object regions where the refinement is required;
- improve the element quality, usually required when the mesh elements do not satisfy some quality criteria such as minimum angle or aspect ratio;
- optimize the current mesh by moving points so the quality of the mesh is improved;
- derefine the current mesh in order to eliminate unnecessary elements or to coarse a mesh in certain regions of the domain;
- generate the final mesh in an appropriate output format.

The steps of a mesh generation process are reified by *algorithms* which have been implemented in different technologies and programming languages. Thus, there are algorithms for mesh generation, for refinement, for improvement, for optimization and for derefinement. Then, there is an intrinsic variability in mesh generation processes, recognized in the different types of input and output formats, in the different types of representations of meshes, and on the variety of algorithms for the different kinds of manipulations of meshes.

Meshing Tools

A *meshing tool* is a software product for generating and managing meshes, enabling the improvement of the quality of meshes by means of the application of algorithms. Thus, a meshing tool reifies and implements a *mesh generation process*. A meshing tool can be general-purpose and provide a variety of algorithms for mesh generation and quality improvement. Meshing tools can be also domain-specific and provide only a particular set of algorithms that are useful in the particular domain the meshing tool is targeted. While some of these useful algorithms might be the general-purpose ones, some others might be specifically designed for the domain. Besides, given that meshing tools can be used in a variety of application domains, different tools tend to require different functionalities.

Meshing tools are inherently sophisticated due to the complexity of the application requirements they have to fulfill. The development of these tools requires specialists from diverse knowledge areas and skills. Besides, the complexity of the algorithms requires from developers high programming skills to develop efficient and robust code, and enough mathematical background to be able to understand the particular requirements of the algorithms. As a consequence, the cost of commercial meshing tools tends to be high. Non-commercial meshing tools and algorithms have been usually developed with ad hoc methodologies, without taking reuse or variability as a goal. In practice, every new tool to target a particular domain needs to be developed from scratch, even in cases that already implemented algorithms and already design data structures.

Thus, meshing tools poses a good opportunity for planned reuse and variability management, making Software Product Line a promising approach for their development. Research effort in this direction has been intensively devoted in our research group.

Business goal. The business goal for defining a software product line for meshing tools is to count on an integrated approach to manage the core assets developed and to be developed, preserving the variability of the domain and enabling the automatic generation of particular products with low effort. Our goal is to illustrate how our work empowers the business goal by making the product line architecture description the core for variability management, for explicitly recording the architecture decisions that render the different product architectures, and for enabling the automatic generation of particular product architectures, as an intermediate step towards the automatic generation or particular products. We are not particularly concerned with the actual implementation of mesh representation data structures, mesh generation, or mesh quality improvement algorithms. These responsibilities are delegated to technical specialists on the corresponding areas. We are concerned with the definition of the critical decisions that rule how the component implementations are integrated in the large number of combinations of particular products of the product line.

Requirements & variability. The most significant functional requirement for the Meshing Tool Software Product Lines is to provide support to all the steps of a *mesh generation process*. Each step renders a functional area of meshing tools that might be fulfilled by the plethora of formats, structures and algorithms. Thus, variability must embrace all func-

tional areas and must provide the alternative mechanisms within the scope of the product line. While the different types of algorithms are part of the variability of the product line, variability on actual implementations of those algorithms may not be necessary.

Requirements on quality attributes are mostly related to the fact that algorithms are resource consuming, both of memory and CPU time. Then, variability in performance and scalability is expected. In terms of performance, faster algorithms are not always preferable as they may be faster but produce less accurate results, or be more expensive in cost, time to market, or time to adoption. In terms of performance on memory consumption, or particularly in terms of scalability on the number of elements or points of the mesh that can be handled by algorithms and by the meshing tool itself. While some products in the product line are targeted to process small meshes, other products are targeted to large meshes that cannot be processed on a single workstation.

Figure 6.6 illustrates the feature model that captures the variability. Notice that the feature model in the figure is not the feature model of the product line that is built during Domain Analysis. The figure illustrates the product line architecture features identified by the architect to organize architecture decisions, as we discuss in the next section. Particularly, notice that scalability is not present as a feature, instead, two alternative solutions are captured: distributed and non-distributed meshing tools. Nevertheless, the feature model provides a grasp on requirements on variability as it is built inspired in the feature model of the product line.

It is important to remark that even though we can think of the particular product that includes all the variants in the product line, i.e. supports all input and output formats, all mesh representations and provides a large set of algorithms for mesh manipulation, such product is rarely desirable in practice. Users tend to require succinct products containing only the set of functionalities that target their particular domain of application.

6.2.2 Designing the Product Line Architecture

In Section §5.2.1 we defined our model-based decision-centric approach to architecture design in the context of software product lines. The foundational principle of our approach is to conceive the product line architecture description as both, an *architecture description* in the sense of our conceptualization of the architecture design practice we defined in Section §3.1, and a *core asset* to be reused in the design of several product architectures in a planned fashion. Being an *architecture description* not only implies that it is structured in terms of the standard, but also that it is conceived by explicitly capturing architecture decisions and solutions, reified by architecture update scripts whose enactment renders the architecture description being built. This conceptualization of architecture description and architecture design was introduced in Section §4.2.1. However, in the context of software architecture product lines, there is not a single architecture description to build, but rather, a set of product architectures that share some commonalities but are distinguished by the potential variability of the product line. Then, our model-based approach to architecture design, in this context, organizes architecture decisions in a way that they can produce the whole set of different product architectures. This is the goal of considering the architecture description

as a *core asset*. As we reviewed in Section §5.1.1 and applied in Section §5.2.2 to the case of product line architecture description, such a description is structured in a common part and a set of variable parts that can be filled-in by specific variants when building particular product assets. In order to capture variability and the potential variants, we introduced the concept of ProductLineArchitectureFeature that represents the distinctive mechanism that captures how to produce and include a variant of a variable part into a product architecture description. Thus, these features allows the architect to organize architecture decisions and solutions in terms of features representing the potential inclusion of a variant in the resulting product architecture.

Then, the architect follows three steps to produce the product line architecture description. First, the architect captures variability in terms of features, identifying the variable parts and the variants required in the product line. Second, the architect makes the decisions regarding the common part of the architecture description, that is shared among all products in the product line. Third, the architect makes the decisions corresponding to each variant captured in the first step. These decisions state how to include each particular feature in the resulting product architecture. In practice, the architecture decisions of a given feature are reified by means of updates to an ongoing architecture description being built, which is set by the decisions of the common part and the decisions of all features required by the given feature. Also, the architect proceeds iteratively through these steps, refining and improving the variability model and the attached decisions incrementally. By this means, the architect can present and discuss the ongoing result with stakeholders, improve visibility of the product line architecture being developed, and can adapt the architecture to changes due to new or missing information uncovered that has an impact on the architecture.

In what follows, we address these three steps in the context of the meshing tool software product line. We do not capture and report the whole set of architecture decisions here. As our focus is to exemplify and validate the application of the approach, we discuss the most relevant aspects and we illustrate the most representative decisions and solutions.

Capturing Variability

The architect starts by capturing the variability of the product line architecture by means of a feature model, as we defined in Section §5.2.1. This feature model is originally inspired in the feature model resulting from the Domain Analysis activity, as it captures the variability expectations on the product line. However, the feature model prepared by the architect may diverge from that as the goal of this structural organization is to assist the architect in managing variability in the product line architecture. In Figure 6.6 we use a feature diagram to illustrate the features that capture the variability requirements and expectations in the Meshing Tool Software Product Line.

Figure 6.6 illustrates in terms of a feature diagram the product line architecture features that capture the variability in the product line architecture as a core asset. Variability is organized in six functional areas. The *User Interface* represents all possible user interfaces for a meshing tool, that range from command language, menu selection, direct manipulation and form fill-in. *Geometry* captures the different mechanisms for the representation of the

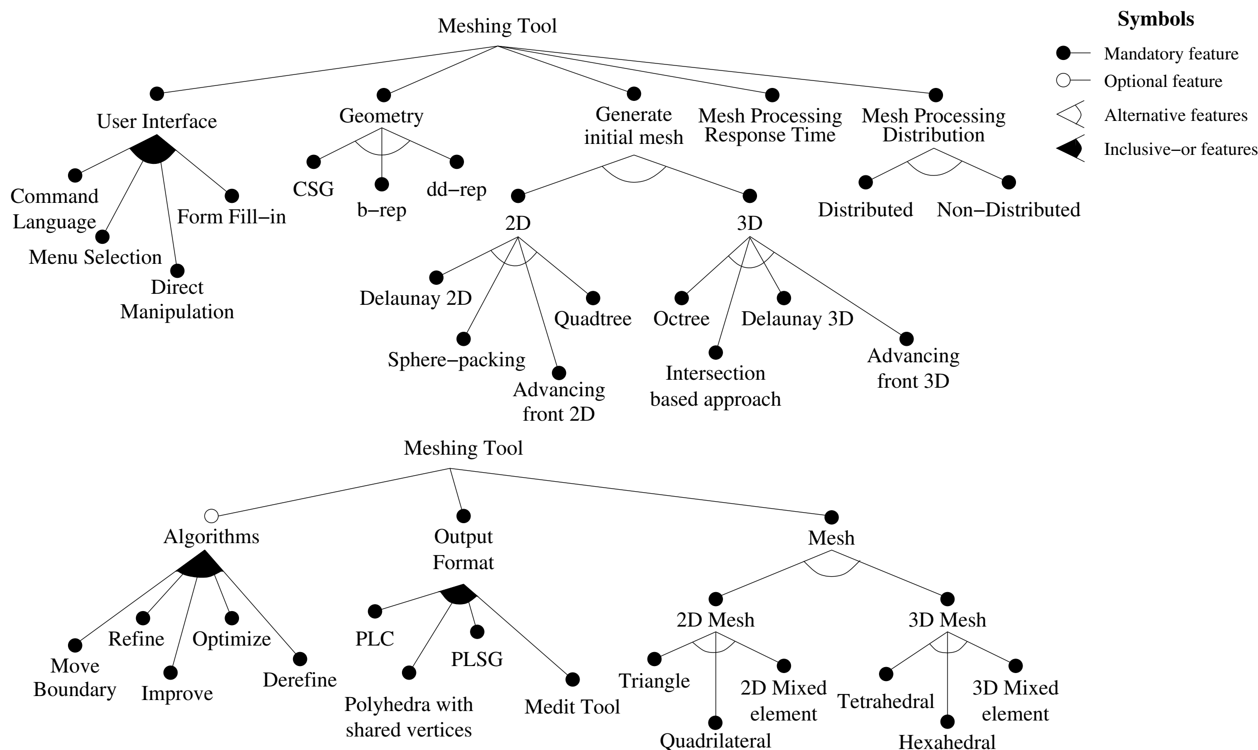


Figure 6.6: *Feature Model for Meshing Tools.*

The figure illustrates the feature models that captures the variability on the Meshing Tool Software Product Line, both on functional and quality attribute requirements.

object to be modeled, that implies different ways of loading information into the meshing tool by processing different input formats. *Generate initial mesh* captures the large number of algorithms for transforming an input geometry – loaded by means of the tools grouped as *Geometry* – to a *Mesh*. These algorithms generate either two dimension or three dimension meshes. The mesh under development can be modified by means of different *Algorithms*. While the diagram in Figure 6.6 captures the major categories of algorithms, as grouped in a mesh generation process, the actual feature model contains a further categorization of these features. Finally, the mesh can be stored in different *Output Formats*. Variability is also captured in two quality areas. *Mesh Processing Response Time* captures the variability in the expectations on the response time. Although not shown in Figure 6.6, different alternatives act as constraints on other features, mainly on algorithms, to enable or disable their selection according to the required or acceptable response times in particular products. *Mesh Processing Distribution* captures variability on the distribution of the processing of the meshing tool. Provided that algorithms are intensive in memory consumption, their application to a large mesh can be unfeasible in a single workstation. We stated this as a scalability requirement in Section §6.2.1. Here, we state this as the two solutions that address that particular requirement: non-distributed products and distributed products. We examine the decision later when we discuss the design of the variable parts.

Designing the Common Part

In this step, the architect *decides* the common part that is shared among the product architecture descriptions of every product in the product line. The goal of these decisions is threefold. First, the architect determines the starting point of the architecture descriptions, adheres to a particular set of good practices on architecture description, and determines its structural organization. Second, the architect addresses the most significant concerns that are shared among all products and that are not conflicting to other concerns or requirements that can vary from product to product. By this means, the architect build the top-level structural organization of the elements populating the architecture description. Third, the architect decides how to set the architecture description to a state in which the realization of the variable parts is possible. The set of decisions made are captured as we conceptualized in Section §4.1.3. As a consequence, the decision selects a solution that, in turn, is reified by an architecture update script that capture how to produce the necessary modifications to the architecture description being constructed.

In the context of the meshing tool, we first (a) decide to start from an empty architecture description. Rigorously, we develop an architecture update script us that contains a single statement s :$_\circ$ StartFromScratch. The justification of such decision is that we neither count with a pre-built architecture description, nor with a reference architecture to start with. Notice that as we explained in Section §5.2.2, the first statement of the script of the first solution decided is actually an StartStatement. When we derive a product architecture description, this is the first statement to be executed. Next, we need to decide which good practice on architecture description to adhere. To this end, we use an already captured system-independent body of knowledge on architecture description, or we applied the step-wise procedure defined in Section §6.1.1 to capture one. We decide to use SEI's Views & Beyond approach to architecture documentation. Particularly, we use the alternative of a single model kind per style category that captures the whole set of elements of the same kind in the same architecture model. For instance, we adhere to V&B understood as an architecture framework, and we have one architecture viewpoint per style category and one model kind per style category that captures the general elements and relation types. Then, Figure 6.1 illustrates the metamodel for architecture models based on Modules. Also, we use the metamodel illustrated in Figure 6.2 for architecture models based on Component & Connectors. The consequences of this decision is that we require less architecture models and less correspondences between them. However, we are also more limited in expressiveness as we have no means to distinguish between component types and component instances as the metamodel in Figure 6.2 does not provide constructs for this purpose.

Second, (b) we need to decide how to address the significant concerns that are common to all products. We proceed iteratively and incrementally, practically applying our model-based approach to architecture design in the context of single-product development that we defined in Section §4.2.1. By this means, we obtain the sequence of decisions that lead to the common architecture description. Then, we repeatedly apply the step-wise procedure defined in Section §6.1.2 to capture the system-independent and system-specific architecture patterns that we need to build the common architecture description. According to the functional requirements and the variability model captured before, a meshing tool has six functional areas where *User Interface* is responsible for user interaction, and *Mesh* is responsible for

data representation and basic manipulation. The functional areas *Generate Initial Mesh*, *Algorithms* and *Output Format* provide specific functionality and operates on the data structures represented by the *Mesh*. The remaining functional area *Geometry* uses the *Generate Initial Mesh* capability to create an initial mesh from the input loaded by *Geometry*. Then, we decide to apply a Blackboard style where a Mesh component acts as the blackboard, and the four areas the processes the mesh acts as knowledge sources. Also, we decide to apply the Call/Return style between a User Interface component responsible of user interaction, and those components providing processing capability that can be requested by the user. Similarly, we apply the Call/Return style between Geometry and Generate Initial Mesh. It is important to remark that even though we are applying architecture styles as defined in V&B, they are actually reifies as *architecture patterns* in our conceptualization as we do not want to extend the base model kind for Component & Connectors, but rather, we aim to populate the corresponding architecture model. Also, these decisions are reified by means of solutions and the corresponding update scripts, particularly the use of the ApplyPattern statement providing the corresponding parameter information. We conceptualized this statement in Section §4.1.2. The sequence of decisions made so far yields to an architecture description with an architecture model populated with the model elements similar to that illustrated in Figure 6.8. However, the main difference is regarding the actual ports (interfaces) of components which are actually appended to the architecture model by the decisions of the variable parts. Notice that the intermediate architecture models rendered by the decisions of the common part can be obtained by applying the semantic function $\mathcal{D}$ to the decisions made and hence exercising the corresponding scripts. In our approach, it is not necessary to count with a complete product configuration to derive the produced artifacts. Intermediate artifacts can be derived by applying any sequence of decisions that respect the precedence relationship between features.

Third, (c) we need to decide how to adjust the current common part to facilitate the decisions for the variable parts. In our case, we decided the main components to be aligned with the major functional areas and hence, variants can be produced by decomposing these components. Also, by means of this common part, variability on quality requirements can be defined by updates to it. We discuss this next.

Designing the Variable Parts

In this step, the architect decides how to include each feature, identified and captured in the variability model built in step one, into the architecture description being built. As we studied in Section §5.2.2, there is a dependency relationship between features, and the updates decided for one feature must be performed under the assumption that the decisions of the required features were already made. Then, the architects visit the features in the feature model in the order that they are applied when deriving product architectures, that is, according to the post-order traversal of the dependency graph. Many of the features in the feature model require corresponding architecture decisions, mainly those features at the leaf of the feature hierarchy. Internal features that are decomposed in other features tend to not require features. However, our method does not restrict that, and any feature can actually have attached decisions.

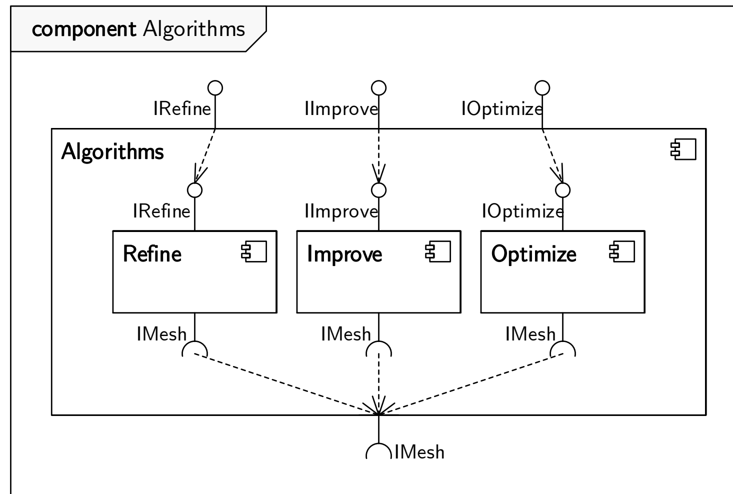


Figure 6.7: *Product Architecture Description (fragment) Concerning Algorithms.*
The figure illustrates the decomposition of the Algorithms component in terms of the particular set of components reifying specific algorithms.

In the context of the meshing tool, we identified in Figure 6.6 one top feature for each functional area. For example, the *Algorithms* feature represents the algorithms functional area, and is decomposed in terms of the different kind of algorithms that can be present in the products in the product line. Let us consider the *Refine* kind of algorithm defined in the feature model. In order to include the corresponding variant in a product architecture, we need to decompose the Algorithms component created by the decisions of the common part. To this end, we define an architecture pattern that creates a new component within the Algorithms component, defines its own port or interface – namely IRefine, – add this interface to the new component and to the outer component, adds the delegation from one port to the other, and proceeds symmetrically for the IMesh interface. If this architecture pattern is sufficiently parameterized in terms of the component to decompose, and the name of the interface to create and delegate, then the same pattern can be applied repeatedly to address variability by the same design mechanism. Figure 6.7 shows the application of the application of this pattern to include three particular kinds of algorithms. However, this decision is not the only one to make to include a particular algorithm into the resulting product architecture. The new defined interface offered by the outer component, and delegated to the new created component, must be used by some other component in the architecture model. In our case, the User Interface is that component. Then, a separate decision updates the User Interface component to use the new defined interface. It is important to remark that this has the corresponding impact in architecture models representing behavior.

A significant variability requirement is that of scalability in some products of the product line. To address this concern, we used distribution and defined two different solutions, one for meshing tools working locally, and other for meshing tools working distributed in several nodes of computation. In a non-distributed meshing tool, we decide that no additional changes are required to the architecture description. In a distributed meshing tool, we apply the Master/Slave pattern. By this means, we create two new components, namely Master Processor and Slave Processor. The slave processor component operates on its own mesh by applying the same algorithms that are available in the meshing tool. However,

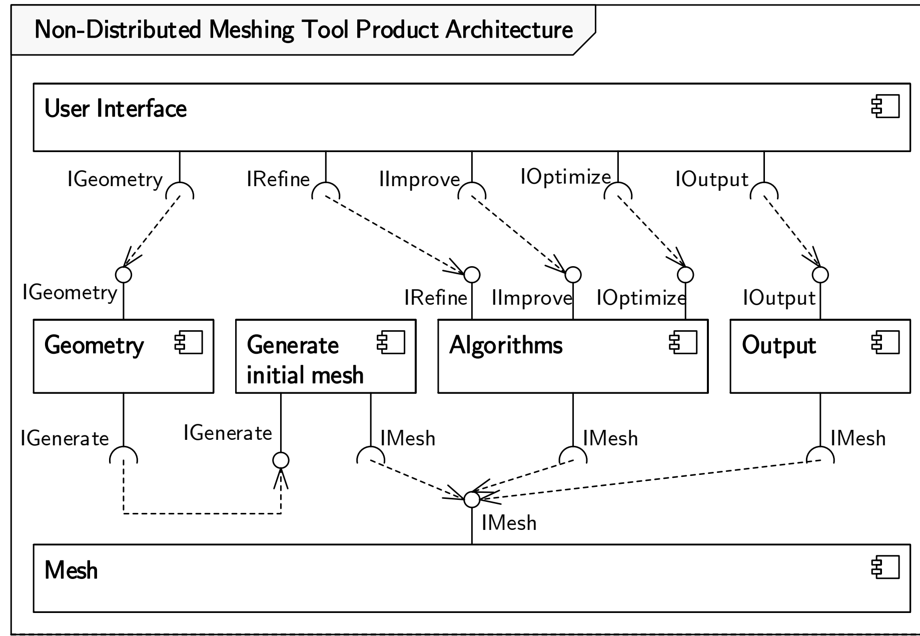


Figure 6.8: *Product Architecture Description of a Non-Distributed Mesh.*
The figure illustrates the architecture model of the product architecture description of a non-distributed meshing tool. The architecture model is governed by a Component & Connector model of the Component & Connector architecture viewpoint defined in Section §6.1.1.

this slave processor is not directly accessible for the user through the user interface. It is the responsibility of the master processor to coordinate user requests and delegate the application of the requested algorithms to different slave processors which operate on the region of the mesh indicated by the master processor. We apply also the Client/Server style to define the purpose-specific data exchange interface to enable the transference of sections of the mesh between the master and the slave processors. We discussed the application of this pattern before in Section §6.1.2, which we illustrated in Figure 6.4, but in the context of V&B styles represented by means of model kinds. The resulting modification of the inclusion of the *Distributed* feature is illustrated in Figure 6.9 which conforms the architecture model for a product of the product line that uses distribution.

6.2.3 Deriving Product Architecture Descriptions

As we conceptualized in Section §5.2.1 our model-based approach to architecting in software product lines enables the automatic derivation of product architecture descriptions from the product line architecture description and the selection of features to be including in the target architecture. We defined the StartFromProductLineArchitectureDescription statement as the special kind of StartStatement that allows architects to start the architecture design of a product in the product line reusing the architecture decisions and solutions captured in the product line architecture description. As we formally specified in Section §5.2.2, the application of the semantic function $\mathcal{D}$ to a particular instance of this start statement defines how the architect should proceed to create the initial version of the product architecture description. The enactment of $\mathcal{D}$ proceeds by making the sequence of decisions for the

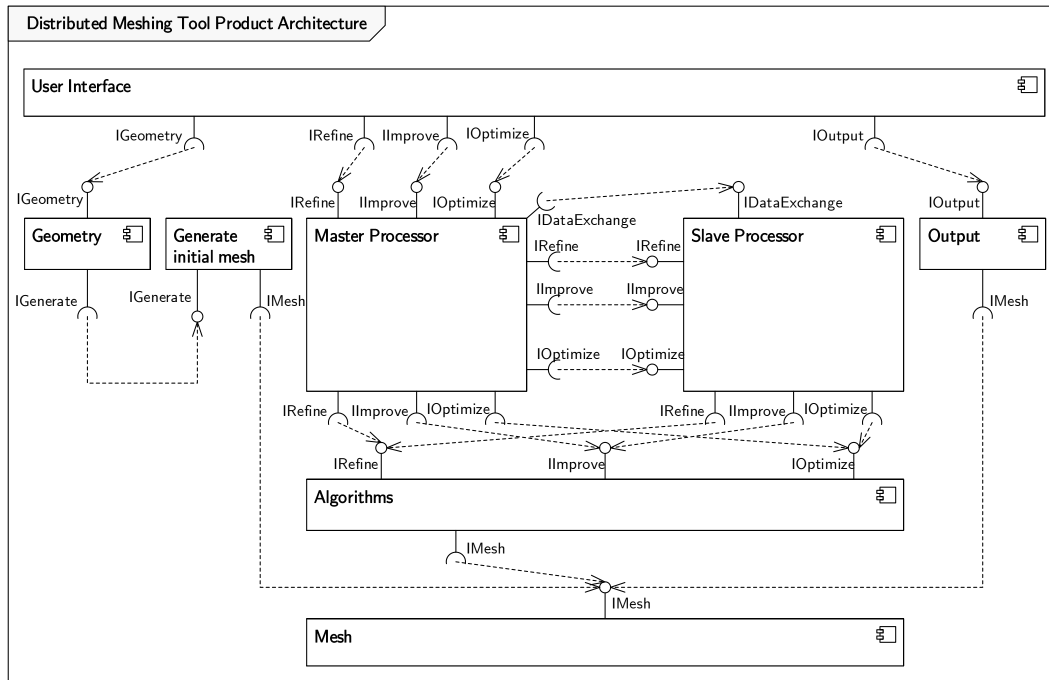


Figure 6.9: *Product Architecture Description of a Distributed Mesh.*

The figure illustrates the architecture model of the product architecture description of a distributed meshing tool. The architecture model is governed by a Component & Connector model of the Component & Connector architecture viewpoint defined in Section §6.1.1.

common part of the product line architecture description, and then making the sequence of decisions for every feature selected by the start statement, in the post-order traversal of the feature model. The enactment of $\mathcal{D}$ is assisted by the tool support available in the modeling environment. The resulting product architecture is an initial one as the particular product may involve the resolution of additional requirements which are out of the scope of the product line, and hence, are not part of the variability management of the product line architecture description.

In order to exemplify this derivation, we select the set of features for two products that provide the same functionality, but vary in the application of distribution. Figure 6.8 illustrates the architecture model resulting of the derivation of the product architecture description for the non-distributed product, and Figure 6.9 illustrates the same model in the case of a distributed product. It is important to notice that the main difference between two models is actually the effect of the sequence of decisions attached to the *Distributed* features, which was selected in one of the products but not in the other.

Chapter 7

Conclusions and Further Work

Software architecture design requires a special mixture of abilities from practitioners. Architects must be creative and visionary, while pragmatic. They must have social skills to productively work with external stakeholders, company or organization managers, product portfolio managers and marketing and sales staff, project managers and all the members of a development team. Architects must have intensive and extensive knowledge and experience on application and technological domains, as well as on problem-resolution techniques, languages and notations. During architecture design, these abilities should lead to high-quality products developed under specific constraints on resources, budget and time. The cumulative body of knowledge produced by two decades of intensive research on the field of Software Architecture, and more that ten years of application in industry, provides architects with methods, techniques, patterns, platforms, and tools, that, once applied, promote informed decisions by predicting potential benefits and liabilities of alternative solutions, reduce the risks of failure, and improve product quality and customer satisfaction. However, software development projects keep failing to deal with the complexity of development and the complexity of software systems, leading to over-budget projects, late or no deliveries, unacceptable levels of quality, low customer satisfaction and return of investment. Software Architecture plays a crucial role in overcoming this reality, being the application of architecture knowledge a key success factor. However, we claim that the intrinsic diverse nature of architecture knowledge, and mainly the heterogeneity of how it is captured and divulged, requires from architects an excessive effort to successfully apply such knowledge during architecture design. Architects need to substantially adapt and adjust current knowledge when applying it to any particular scenario, considering the scarce specialized tool support available, and mainly relying on text- and diagram-based editors. As a result, architects produce architectures that do not achieve their full potential providing the size and coverage of current architecture knowledge.

Our position is that a unified and homogeneous means of capturing architecture knowledge, making it not only shareable and reusable in different development scenarios, but also directly applicable in practice, centers the architects' focus on what is actually important – to address the significant concerns and requirements of the system under development, – reducing the extra effort of adapting and adjusting such knowledge to the particular scenario, of capturing information regarding traceability, and of checking consistency along the different parts of

architecture descriptions. Hence, the main focus of our research effort and contributions has been to develop such a unified and homogeneous means to capture architecture knowledge, and once achieved, to define how the decision-making activity of an architecture design effort can be understood in terms of the direct application of such knowledge.

This chapter is structured as follows. In Section §7.1 we revisit the research hypothesis we postulated and we make our research thesis statement. In Section §7.2 we review the main contributions of our work, we situate them in the context of the current practice in the research and industry community, and we draw conclusions of our research and results, and discuss their benefits, implications and limitations. We also comment on lessons learned during the process of our research effort. Finally, Section §7.3 closes with an outline of perspectives and further work.

7.1 Research Thesis

The research hypothesis we postulated in Section §1.3 for our work states that:

- Current Model-Driven Engineering constructs, techniques and tools, can provide a homogeneous and unified means for capturing and communicating architecture knowledge on architecture description and design, making such knowledge directly applicable by architects, favoring reusability, automating traceability, and enabling tool-support.

The Model-Driven Engineering discipline has evolved significantly in the last decade, both in its conceptualization and in its tool support. The distinction between modeling-in-the-small and modeling-in-the-large has provided the research community the necessary contextualization for research efforts, tools, and their application. While the former embraces techniques and technology for working on single models and model transformations, the latter is aimed toward scenarios involving a large number of modeling artifacts. The megamodeling approach emerges to deal with the complexity of this scenario, promoting the use of modeling techniques to manage modeling artifacts, their properties and interrelations. Thus, the megamodeling approach provides the foundation to the characterization of modeling constructs and their related tools. In Section §2.3 we studied the Global Model Management approach to megamodeling, and we refined it so as to provide a technology-independent definition. It is important for us to define modeling constructs without relying on any particular technological aspect, so as to keep our work current, independently of the active evolution of techniques and tools for modeling-in-the-small.

Then, we position those defined modeling constructs as the unified means to homogeneously capture architecture knowledge, and we thoroughly developed the interpretation of the concepts in the architecture practice to the modeling constructs. To this end, we devise and apply the strategy of (a) conceptualizing the body of knowledge to capture, (b) mapping each concept and any concept instance in term of modeling constructs, and (c) formally specifying the mapping in terms of a denotational semantics approach. In Section §3.1 we refined the ISO/IEC/IEEE 42010:2011 standard to define the conceptualization of the architecture description practice, including system-independent knowledge captured by architecture

framework and description languages. In Section §3.2 we defined the mapping for these concepts and we formalized them using a denotational semantics approach. In Section §4.1 we defined a conceptualization of architecture design mechanisms that capture system-dependent and system-independent knowledge on how to process architecture descriptions to enrich its expressiveness (styles) and to include the resolution of significant concerns or requirements (patterns and tactics). To this end, and based on the current literature, we extended the conceptualization on architecture description to include these concepts. In Section §4.2.2 we extended our interpretation in terms of modeling constructs, and our formal specification in terms of the denotational semantics approach. By this means, we make architecture knowledge homogeneously captured in terms of modeling constructs, and we enable tool-support by relying on the technological support for modeling techniques currently available. Also, we favor reuse as the modeling artifacts resulting from the model-based interpretation can be directly used in an architecture description and design effort.

However, architecture knowledge on architecture design goes beyond capturing architecture descriptions and system-independent design mechanisms. It also includes architecture design processes whose main goal is to design the architecture of a system of interest that meets stakeholders' expectations and whose main output is an architecture description. Our work did not capture this kind of knowledge as a whole, but rather, it is centered on the decision-making activity where the architecture is actually devised and decided, and the architecture description is actually constructed. To this end, we defined an architecture design scripting language in Section §4.1.2 that conceptualizes the fine-grained updates that an architect performs to incrementally and iteratively produce and build and architecture description by applying architecture knowledge. These fine-grained updates are defined in terms of our conceptualization, and thus, they represent the mechanism on *how* to directly apply architecture knowledge during architecture design. Additionally, we extended our specification to formally define the effect of these updates on an architecture description. Such a formal specification provides the foundation to fully automation of the execution of scripts.

We proceeded a step forward and we defined a model-based decision-centric approach to architecture design in Section §4.2.1 that promotes to shift the architect's focus from explicitly capturing architecture descriptions along with the description of the main decisions made, to explicitly capturing the architecture decisions and their effect of an implicit architecture description using our architecture design modeling language. Thus, the architect explicitly captures only *how* he proceeds and *why* he proceeds in this particular way, rendering the derivation of the architecture description derivable from exercising the fine-grained updates. The conceptualization of architecture decisions and solutions was defined in Section §4.1.3 and its formal specification in Section §4.2.3. By this means, we also favor reusability of the actual design steps on building architecture descriptions, and we automate traceability by delegating it to modeling tools that generate trace information on each model transformation reifying the updates and that produce fine comparison of different versions of models, as we defined in Section §4.2.4.

Consequently, we have proved our hypothesis as we used modeling constructs – particularly those defined by the megamodeling approach – to capture architecture knowledge making it shareable, reusable, directly applicable, automating traceability and making it tool-friendly. We cover both architecture description and architecture design, but, in the latter, we focused

only on its decision making activity whose goal is the actual construction of the architecture description work product.

As a corollary, we postulated in Section §1.3 that:

- Planned reuse of model-based architecture decisions and solutions can automate product architecture design in architecture-centric Software Product Line development.

Architecture-centric Software Product Line development relies on a product line architecture not only to meet the critical functionality, the expectations on quality attributes, the constraints and the potential risks of failure, that are common to all the products in the product line, but also to explicitly capture the variation mechanisms to support the diversity among the products. Each product in the product line has a product architecture that is built using the product line architecture and where all variability has been resolved. Provided that the product line architecture is a core asset that focuses on capturing commonality and variability, and that product architectures are product assets derived from that core asset, we applied our conceptualization, formalization and model-based approach to provide automation to the derivation of product architectures. To this end, in Section §5.2.1 we first conceptualized a product line architecture description as both an architecture description (in the sense of Chapter §3) and a core asset. Second, we used our conceptualization on architecture decisions and solutions as the particular variation mechanism to determine how to create variants in the product architectures. By this means, a product line architecture is a sequence of architecture decisions common to all products, and several sequences of architecture decisions to be made to produce a variant for each variable part in the architecture. We used features to structurally organize these decisions and to capture their interdependencies. Then, we conceptualized product architecture design as the selection of those features that are required in the product architecture. Thus, by making the common architecture decisions and the decisions for the variants in the correct order, and consequently, by exercising the architecture update scripts attached to the solutions selected by the decisions, the product architecture description is automatically derived.

Consequently, we have proven our corollary as we used a planned-reuse approach by conceiving and organizing architecture decisions and solutions, mainly in terms of features that represent the variability supported by the product line architecture, to automate the derivation of the product architectures. Automation relies on exercising our formalization for the effect of the architecture update scripts. Our approach automates product architecture design when the required product exactly fits in the scope of the product line. In this case, the product architecture is derived from the product line architecture. However, in practice, stakeholders usually impose additional requirements to particular products that were not considered in the scope of the product line. In this case, our approach yields the initial product architecture and enables the architect to use our model-based approach to architecture design to continue on refining this derived architecture description.

7.2 Contributions & Lessons Learned

Megamodeling in Software Architecture

The widespread usage of text- and diagram-based editors to build the architecture description of a system-of-interest leads practitioners to consider the architecture description as a single work product. This work product is captured by a single document, which is internally organized according to some architecture description method or technique. For instance, the Rational Unified Process calls this artifact the Software Architecture Document (SAD), which is internally structured in sections following the 4+1 viewmodel approach. Those architects following the standard also use sections to capture views and the related information, and those following the Views & Beyond approach use additional sections to document information beyond views. However, the standard is not committed to a particular medium, and mentions document-based descriptions as one of the most notorious, along with repository-based and model-based. The Views & Beyond approach, for instance, also discusses the possibility of capturing documentation packages by means of wikis, i.e. an interconnected repository of text- and diagram-based parts of the architecture description. A model-based approach relies on models and modeling techniques. For us, the model-based approach is the unique one that allows for the separation of capturing the design of the architecture and communicating the design of the architecture. In the other approaches, these two concerns are tangled. The benefits of separating design from design communication is twofold. First, it allows the architect to capture the architecture description that, while satisfying the information needs of stakeholders, it reduces the need of repeating the same information at different levels of abstraction and rigor. Also, it allows the architect to structurally organize an architecture description according to his own best practices, delegating to a second stage the generation of the documentation expected by stakeholders. Second, the generation of such a documentation can be automated by means of modeling techniques. Moreover, once developed, those techniques can be reused in other development efforts to produce similar documents from the same kind of architecture models but populated with different information. We studied this aspect in Section §3.2.4.

Our position is that the use of modeling techniques in software architecture design must be considered as an application of modeling-in-the-large. No single metamodel can actually embrace and completely cover all the aspects and perspectives that are involved in architecture and captured in architecture descriptions. A set of modeling artifacts is required, at least different terminal models to capture architecture models, along with the corresponding metamodels defining the modeling languages. Our approach goes even further by conceiving every bit of information to be captured in architecture descriptions to be represented by means of models. The main goal of our model-based interpretation is to define which are those exact models needed. However, the notion of megamodeling is somehow still alien to the architecture practice. For instance, megamodeling is not mentioned in the ISO 42010 standard, and the research effort in this direction is handful. For instance, J. M. Favre in [Fav04a] used megamodeling to align IEEE 1471 standard's concepts to model-driven techniques. We introduced in [PBR09] our model-driven approach to architecture design using megamodels for the representation of architecture descriptions, and model transformations to explicitly capture the effect of architecture design decisions. R. Hilliard et al. in [HMMP10] presented

MEGAF, a model-based infrastructure for realizing architecture frameworks. The authors define an extension to GMM's conceptual framework, introducing key concepts in the context of architecture frameworks as sub-metaclasses of GMM's metamodel for megamodels. In our work, we refined and improved our previous work [PBR09] by using GMM as the megamodeling approach to capture architecture knowledge. For us, megamodeling, and particularly GMM, is a general-purpose technique, independent of any specific application domain, whose extensions are intended to cope with different modeling-in-the-small techniques like metamodeling languages such as ECore, model transformation languages such as ATL, and model weaving such as AMW, among others. These extensions are defined by GMM documentation and implemented in the AM3 tool set. Then, we followed a different approach than [HMMP10]. Instead of defining an extension, we defined and formalized a mapping from the domain-specific concepts in software architecture to the general-purpose modeling constructs. Extending GMM can be actually a means to reify our mapping.

Nevertheless, we foresee growth in the application of megamodeling to architecture in the years to come. Modeling techniques are pervading all major areas of Software Engineering, and developers are starting to use models and modeling techniques in their practice. Thus, the role and use of models in architecture will necessary grow as architecture is a centerpiece in development efforts. Consequently, megamodeling will be required to deal with the complexity of the large number of artifacts involved in an architecture design effort.

Application & Adoption

Our research effort was performed in three consecutive stages, Fundamentals & Feasibility, Development, and Refinement & Formalization, as we presented in Section §1.6. Along these stages, we applied our work to address different kinds of systems. In the first stage we used the Point-of-Sale system as the case study. We used it in our application of modeling-in-the-small techniques to the extraction of behavior from contract specification in [VPB08], and during the conception and proof-of-concept of our model-based approach to architecture design in [PBR09]. During the second stage, we decided to shift to the Meshing Tool Software Product Line, as it poses a better opportunity to the validation of our approach: its domain analysis effort had been developed by our research group, the functional, quality and variability requirements had been well-identified, and it consisted on a larger and more complex system that our previous case study. We used this case study in our initial approach to architecture design in the context of software product lines, that we reported in [PRB09, RPB09]. We improved this initial work in this thesis, and we used in Chapter §6 the same case study to illustrate the application of our model-based approach for planned reuse of architecture decisions and solutions. As a parallel project, we are pursuing the transference to industry of our model-based approach. We worked as the leader architect in the development of a software product line of an Enterprise Document Management system, which the company is still developing, improving, maintaining and evolving. While this project is not part of the goals of our work, there are some interesting partial results that we can comment on. In this experience, we followed an incremental approach to products lines, starting from one product and evolving to the product line. In two years, the development team built three products of the Enterprise Document Management product line, in 11, 6

and 4 months respectively, even with the financial pressure, the limitations of tool support to manage the product line, the repeated change in the members of the small development team, and the change on the major version of the underlying supporting platform of the production environment. The key factors to reduce the time-to-market relies on the benefits that introduces the adoption of a product line approach. After the first product, the team gained expertise in the application domain and experience on stakeholders' expectations and final user satisfaction. The key features of the product line were identified and used as part of the sale speech. Also, the approach of associating architecture decisions to features was well received although only partially and informally captured. Pressure on time and budget restrictions made the architecture description not to be captured, ending up embedded in source code documentation and implicit in the team members. By applying our model-based approach to a full extent approach, the company would not suffer from knowledge loss due to changes in the development team. Also, actual product architectures would be possible to share with stakeholders, and developers would count on traceability information to improve and refine the new versions under development. However, the lack of pre-captured architecture knowledge, and the up-front cost of capturing it, rendered a full application of our approach unfeasible in the context of this small company.

This experience raises the question of the feasibility of the adoption of our model-based approach. Our position is that in a short timespan, a company will not be able to adopt our approach because it implies the responsibility and effort of both, capturing the foundational system-independent architecture effort currently available in the software architecture community, and also the system-dependent knowledge that is particular to the architecture design effort at hand. For us, this problem is not intrinsic to our model-based approach, but to architecture knowledge as a whole. As discussed in some of the panels held at the joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA) in 2012 [WIC12], practitioners remarked a gap between research and practice priorities, and questioned the proliferation of techniques with little cooperation and reuse, and their scarce transference to industry. These remarks are valid independently of the medium in which such knowledge is published or represented. However, we claim that, if captured by means of our model-based interpretation and formalization, such knowledge can provide the additional benefit of being applicable, reducing any additional effort of adjusting and adapting the large and heterogeneous body of architecture knowledge.

The main contribution of our work is the homogeneous means for capturing architecture knowledge that we defined in terms of modeling constructs and techniques. This contribution is not primarily targeted to small companies designing architectures for the systems they develop. It is targeted to the practitioner and research community that is actually proposing and improving the body of knowledge of the software architecture discipline. We cannot expect small or medium companies to make the effort of mapping knowledge to actual usable and working artifacts, these artifacts must be provided to them. In the current state of the art, the community is divulging such knowledge using diverse representation techniques, different levels of abstraction and rigor, and is not committed to any medium to capture and to provide support and applicability to the captured knowledge. Our work is a step forward this direction, a formal mapping to a unified medium that provides both, to capture knowledge and to render it tool-friendly and applicable. Practitioners may also use our mapping to pro-

duce in-company modeling artifacts for their organization-specific architecture frameworks, description languages and techniques. Our position is that this effort is worthwhile, but it takes time and resources to be done. As we defined in Chapter §4, we provide step-wise procedures on how to capture a body of knowledge according to our formalization. We encourage practitioners and the research community to start applying them to produce shareable and reusable artifacts that can be successfully applied in practice. The homogeneous means we propose enhances and facilitates cooperation as it focuses in the knowledge to capture, and not in the peculiarities of the so many alternatives on how to capture it. We do not claim that the formal interpretation we defined is neither the unique possible one nor the best one. We do claim that it conforms a starting point towards better collaboration and higher quality in knowledge transference. We foresee that the actual application of our interpretation will lead to its refinement and improvement. We actually hope so as we, as a practitioner and research community, will be counting on a homogeneous means to share our knowledge.

7.3 Perspectives & Further Work

Our model-based systematization of Software Architecture Design that we developed in this work enables build supporting tools on a formal and sound basis.

The most important direction for further work is to achieve fully automation of the derivation of architecture descriptions from architecture decisions and solutions. Current tool support provides automation of each kind of statement in our architecture design scripting language, as their effect is defined in terms of modeling techniques available in modeling environments. However, automation of scripts is not available yet. It requires an interpreter tool, i.e. a tool that can process scripts and produce the corresponding effect. The formal specification that we defined in terms of the semantic functions $\mathcal{M}$ and $\mathcal{D}$, embodies the formal specification of this interpreter tool as it rigorously defined both the artifacts to manipulate and the effect of the statements in terms of modeling techniques. In the same direction, adding the ability to process feature configurations and feature models, we can fully automate also the derivation of product architecture descriptions from product line architecture descriptions and feature configurations.

Our model-based approach to architecture design is not an architecture design method by itself. It is committed to a model-based representation of architecture descriptions, it relies on the principle of explicit architecture decisions and application architecture solutions, and it can assist the decision-making activity of an architecture design process. Thus, another line of further research consists of applying modeling techniques to capture knowledge on architecture design that is embedded in architecture design processes. To this end, the application of process modeling languages such as Software & Systems Process Engineering Metamodel (SPEM) [OMG08] not only promotes the rigorous definition of processes, but also, enables the application of tools to analyze and evaluate the processes.

Finally, we encourage the community to align to our unified and homogeneous means for capturing architecture knowledge. Any step towards this direction will allow the improvement of our interpretation, will provide shareability, reusability and applicability of architecture

knowledge between research groups and the practitioner community, and will start to generate the asset base to enable its full application in practice, obtaining the benefits of reusability and variability management, automatic traceability, and improved software quality by facilitating the application of architecture knowledge.

Model-Based Systematization of Software Architecture Design

Model-Based Interpretation of Software Architecture Knowledge

Software architecture design is the creative and dynamic endeavor performed by the architect to iteratively and incrementally design and build the architecture of a system of interest that best addresses stakeholders' functional and quality expectations, constrained by the available team, budget and time. The successful application of architecture knowledge is decisive in achieving an architecture that provides the best possible benefits. However, the current status on how architecture knowledge is captured, communicated, shared and reused, is an obstacle to practitioners as is requires architects to devote a great effort to accommodate such knowledge to make it applicable. Our model-based interpretation of architecture knowledge overcomes this issue by providing a single technical means — namely modeling techniques — to homogeneously represent, divulge and use such knowledge. In turn, this model-based interpretation enables the definition of a concrete model-based decision-centered architecture design approach and a supporting scripting language, that conforms the systematization of the decision-making activity of architecture design. Given that architecture design is intrinsically a human activity where creativity on problem-resolution is crucial, a fully automated approach seems unfeasible today. However, this fact should not discourage any form of systematization and tool assistance. For us this is pivotal for success. Our model-based approach and formalization is a step forward in the direction of allowing architects to focus their effort on what they are best at — to address and resolve significant expectations of a system, — while systematizing and assisting the effort of applying architecture knowledge to achieve their goal.

Bibliography

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. In David Notkin, editor, *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'1993), 7-10 December 1993, California, USA*, pages 9–20. ACM, 1993.

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based Product Line Development: The KobrA Approach. In Patrick Donohoe, editor, *Proceedings of the 1st International Conference on Software Product Lines (SPLC'2000) – Experiences and Research Directions, 28-31 August 2000, Denver, Colorado, USA*, pages 289–310. Kluwer, 2000.

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Slicing Feature Models. In Perry Alexander, Corina Pasareanu, and John Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE'2011), 6-10 November 2011, Lawrence, KS, USA*, pages 424–427. IEEE, 2011.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, July 1997.

[AG01] Michalis Anastasopoules and Critina Gacek. Implementing Product Line Variabilities. In SSR'2001 [SSR01], pages 109–117.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press, 1977.

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, September 2003.

[AMC+06] Eduardo Almeida, Jorge Mascena, Ana Cavalcanti, Alexandre Alvaro, Vinicius Garcia, Silvio Meira, and Daniel Lucrédio. The Domain Analysis Concept Revisited: A Practical Approach. In Maurizio Morisio, editor, *Proceedings of the Reuse of Off-the-Shelf Components, 9th International Conference on Software Reuse (ICSR'2006), 12-15 June 2006, Turin, Italy*, volume 4039 of *Lecture Notes in Computer Science*, pages 43–57. Springer, 2006.

[ASM03] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In Frank van der Linden, editor, *Revised papers of the 5th International Workshop on Software Product-Family Engineering (PFE'2003), 4-6 November 2003, Siena, Italy*, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer, 2003.

[Atl09] AtlanMod. AtlanMod MegaModel Management (AM3) Flyer-Poster, 2009. http://docatlanmod.emn.fr/AM3/Documentation/AM3_Flyer-Poster_v1-0.pdf (accessed on August 2014).

[Atl14a] The AtlanMod research group, 2014. http://www.emn.fr/z-info/atlanmod (accessed on August 2014).

[Atl14b] AtlanMod. How Install New AM3 From SVN, 2014. http://wiki.eclipse.org/AM3/How_Install_New_AM3_From_SVN (accessed on August 2014).

[AZ05] Paris Avgeriou and Uwe Zdun. Architectural Patterns Revisited – A Pattern Language. In Andy Longshaw and Uwe Zdun, editors, *Proceedings of the 1'th European Conference on Pattern Languages of Programs (EuroPLoP'2005), 6-10 July 2005, Irsee, Germany*, pages 431–470. UVK - Universitaetsverlag Konstanz, 2005.

[Bal07] Ricardo Balduino. Introduction to Eclipse Process Framework: EPF Composer and OpenUP/Basic. In *Proceedings of the Eclipse Conference 2007 (ECLIPSECON'2007), 6 March 2007, Santa Clara, California, USA*, 2007.

[BB01a] Felix Bachmann and Leonard Bass. Introduction to the Attribute Driven Design Method. In Hausi Müller, Mary Harrold, and Wilhelm Schäfer, editors, *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), 12-19 May 2001, Toronto, Ontario, Canada*, pages 745–746. IEEE Computer Society, 2001.

[BB01b] Felix Bachmann and Leonard Bass. Managing Variability in Software Architectures. In SSR'2001 [SSR01], pages 126–132.

[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering, Languages and Systems (MoDELS'2006), 1-6 October 2006, Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.

[BBJR87] Victor Basili, John Bailey, Bok Joo, and Dieter Rombach. Software Reuse: A Framework for Research. In *Proceedings of the 10th Minnowbrook Workshop on Software Performance Evaluation, July 1987, New York, USA*, July 1987.

[BBK02] Felix Bachmann, Leonard Bass, and Mark Klein. Illuminating the Fundamental Contributors to Software Architecture. Technical Report CMU/SEI-2002-TR-

025, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, August 2002. http://www.sei.cmu.edu/reports/02tr025.pdf (accessed on August 2014).

[BBK03] Felix Bachmann, Leonard Bass, and Mark Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report CMU/SEI-2003-TR-04, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, March 2003. http://www.sei.cmu.edu/reports/03tr004.pdf (accessed on August 2014).

[BBKS04] Felix Bachmann, Leonard Bass, Mark Klein, and Charles Shelton. Experience Using an Expert System to Assist an Architect in Designing for Modifiability. In WICSA'2004 [WIC04], pages 281–284.

[BBKS05] Felix Bachmann, Leonard Bass, Mark Klein, and Charles Shelton. Designing software architectures to achieve quality attribute requirements. *IEE Proceedings - Software*, 152(4):153–165, August 2005.

[BBN07] Felix Bachmann, Leonard Bass, and Robert Nord. Modifiability Tactics. Technical Report CMU/SEI-2007-TR-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, September 2007. http://www.sei.cmu.edu/reports/07tr002.pdf (accessed on August 2014).

[BBSK05] Rabih Bashroush, John Brown, Ivor Spence, and Peter Kilpatrick. ADLARS: An Architecture Description Language for Software Product Lines. In *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW'2005), 6-7 April 2005, Greenbelt, Maryland, USA*, pages 163–173. IEEE Computer Society, 2005.

[BC05] Felix Bachmann and Paul Clements. Variability in Software Product Lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, September 2005. http://www.sei.cmu.edu/reports/05tr012.pdf (accessed on August 2014).

[BCK98] Leonard Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Professional, 1998.

[BCK03] Leonard Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Professional, 2nd edition, April 2003.

[BEL+03] Mario Barbacci, Robert Ellison, Anthony Lattanze, Judith Stafford, Charles Weinstock, and William Wood. Quality Attribute Workshops (QAWs), third edition. Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, October 2003. http://www.sei.cmu.edu/reports/03tr016.pdf (accessed on August 2014).

[Béz04] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE – The European Journal for the Informatics Professional*, 5(2):21–

24, April 2004.

[Béz05a] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Revised papers of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005), 4-8 July 2005, Braga, Portugal*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2005.

[Béz05b] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[BFG+01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In Frank van der Linden, editor, *Revised papers of the 4th International Workshop on Software Product-Family Engineering (PFE'2001), 3-5 October 2001, Bilbao, Spain*, volume 2290 of *Lecture Notes in Computer Science*, pages 13–21. Springer, 2001.

[BFJ+03] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective Model Driven Engineering. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proceedings of the 6th International Conference on the Unified Modeling Language: Modeling Languages and Applications (UML'2003), 20-24 October 2003, San Francisco, CA, USA*, volume 2863 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2003.

[BH04] Jean Bézivin and Reiko Heckel, editors. *Proceedings of the Language Engineering for Model-Driven Software Development Seminar, 29 February - 5 March 2004, Germany*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[BH06] Nelis Boucké and Tom Holvoet. Relating Architectural Views with Architectural Concerns. In *Proceedings of the Workshop on Early Aspects (WEA'2006) at the 28th International Conference on Software Engineering (ICSE'2006), 20-28 May 2006, Shanghai, China*, pages 11–18, New York, NY, USA, 2006. ACM Press.

[BJB08] Mikaël Barbero, Frédéric Jouault, and Jean Bézivin. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In ECBS'2008 [ECB08], pages 277–286.

[BJRV04] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2004.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patick Valduriez. On the Need for Megamodels. In *Best Practices for Model-Driven Software Development Workshop, at the Third International Conference on Generative Programming and Component Engineerings (GPCE'2004), co-located with the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2004), 24-28 October 2004, Vancouver, Canada*, 2004.

[BLHM02] Don Batory, Roberto Lopez-Herrejon, and Jean-Philippe Martin. Generating Product-Lines of Product-Families. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 81–92. IEEE Computer Society, 2002.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*, volume 1. John Wiley & Sons, Inc., New York, NY, USA, August 1996.

[Boo14] Grady Booch. Handbook of Software Architecture, 2014. `http://www.handbookofsoftwarearchitecture.com/` (accessed on August 2014).

[Bos00] Jan Bosch. *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley Professional, May 2000.

[Bos02] Jan Bosch. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In SPLC'2002 [SPL02], pages 257–271.

[Bos04] Jan Bosch. Software Architecture: The Next Step. In Flávio Oquendo, Brian Warboys, and Ronald Morrison, editors, *Proceedings of the 1st European Workshop on Software Architecture (EWSA'2004), 21-22 May 2004, St. Andrews, UK*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2004.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley object technology series. Pearson Education, 1999.

[BS92] Barry Boehm and William Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference 1992, 28-30 April 1992, Los Angeles, CA, USA*, pages 63–82. Meridien Corp., 1992.

[BSK+08] Rabih Bashroush, Ivor Spence, Peter Kilpatrick, John Brown, Wasif Gilani, and Mathias Fritzsche. ALI: An Extensible Architecture Description Language for Industrial Applications. In ECBS'2008 [ECB08], pages 297–304.

[CAB10] Lianping Chen and Muhammad Ali Babar. Variability Management in Software Product Lines: An Investigation of Contemporary Industrial Challenges. In Jan Bosch and Jaejoon Lee, editors, *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'2010), 13-17 September 2010, Jeju Island, South Korea*, volume 6287 of *Lecture Notes in Computer Science*,

pages 166–180. Springer, 2010.

[CAB11] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information & Software Technology*, 53(4):344–362, 2011.

[CABA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability Management in Software Product Lines: A Systematic Review. In Dirk Muthig and John McGregor, editors, *Proceedings of the 13th International Conference on Software Product Lines (SPLC'2009), 24-28 August 2009, San Francisco, California, USA*, volume 446 of *ACM International Conference Proceeding Series*, pages 81–90. ACM, 2009.

[CBB+10] Paul Clements, Felix Bachmann, Leonard Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'2002), 6-8 October 2002, Pittsburgh, PA, USA*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2002.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, 2000.

[CGB+02] Paul Clements, David Garlan, Leonard Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Addison-Wesley Professional, September 2002.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In Ulrich Eisenecker, Sven Apel, and Stefania Gnesi, editors, *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems, 25-27 January 2012, Leipzig, Germany*, pages 173–182. ACM, 2012.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems*, 45(3):621–645, July 2006.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In SPLC'2004 [SPL04], pages 266–283.

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In José Fiadeiro and Paola

Inverardi, editors, *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'2008) at the Joint European Conferences on Theory and Practice of Software (ETAPS'2008), 29 March - 6 April 2008, Budapest, Hungary*, volume 4961 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6):37–45, 1998.

[Cle96] Paul Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD'1996), 22-23 March 1996, Paderborn, Germany*, pages 16–25, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[CM02] Gary Chastek and John McGregor. Guidelines for Developing a Product Line Production Plan. Technical Report CMU/SEI-2002-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, June 2002. `http://www.sei.cmu.edu/reports/02tr006.pdf` (accessed on August 2014).

[CMV+10] Robert Cloutier, Gerrit Muller, Dinesh Verma, Roshanak Nilchiani, Eirik Hole, and Mary Bone. The Concept of Reference Architectures. *Systems Engineering*, 13(1):14–27, February 2010.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. The SEI Series in Software Engineering. Addison Wesley Professional, 2002.

[CNYM99] Lawrence Chung, Brian Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5 of *The Kluver International Series in Software Engineering*. Springer, October 1999.

[CZZ+11] Rafael Capilla, Olaf Zimmermann, Uwe Zdun, Paris Avgeriou, and Jochen Küster. An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 303–318. Springer, 2011.

[DAH+07] Eric Dashofy, Hazeline Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. In *Companion Volume of the 29th International Conference on Software Engineering (ICSE'2007), 20-26 May 2007, Minneapolis, USA*, pages 67–68. IEEE Computer Society, 2007.

[DC05] Juan Dueñas and Rafael Capilla. The Decision View of Software Architecture. In Ronald Morrison and Flávio Oquendo, editors, *Proceedings of the 2nd European Workshop on Software Architecture (EWSA'2005), 13-14 June 2005, Pisa, Italy*, volume 3527 of *Lecture Notes in Computer Science*, pages 222–230. Springer, 2005.

[DHT05] Eric Dashofy, André van der Hoek, and Richard Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, April 2005.

[Dij74] Edgser Dijkstra. *On the Role of Scientific Thought*, chapter EWD447 of *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, New York, NY, USA, August 1974.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. *ACM SIGPLAN Notices*, 10(6):114–121, April 1975.

[DM12] Zinovy Diskin and Tom Maibaum. Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond. In Ulrike Golas and Thomas Soboll, editors, *Proceedings of the 7th Workshop on Applied and Computational Category Theory (ACCAT'2012) at the European Joint Conferences on Theory and Practice of Software (ETAPS'2012), 24 March - 1 April 2012, Tallinn, Estonia*, volume 93 of *EPTCS*, pages 1–21, 2012.

[DN02] Liliana Dobrica and Eila Niemelä. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, July 2002.

[DoD07] US Department of Defense. DODAF 1.5 – DoD Architecture Framework Version 1.5, April 2007.

[DoD09] US Department of Defense. DODAF 2.0 – DoD Architecture Framework Version 2.0, May 2009.

[DS99] Jean-Marc DeBaud and Klaus Schmid. A Systematic Approach to Derive the Scope of Software Product Lines. In Barry Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering (ICSE'1999), 16-22 May 1999, California, USA*, pages 34–43. ACM, 1999.

[DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in Software Product Families: Problems and Issues During Product Derivation. In SPLC'2004 [SPL04], pages 165–182.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[ECB08] *Proceedings of the 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS'2008), 31 March - 4 April 2008, Belfast, Northern Ireland*, Washington, DC, USA, 2008. IEEE Computer Society.

[Ecl14a] Eclipse. The AM3 project of the Eclipse Community, 2014. `http://wiki.`

eclipse.org/AM3 (accessed on August 2014).

[Ecl14b] Eclipse. The Modeling project of the Eclipse Community, 2014. http://projects.eclipse.org/projects/modeling (accessed on August 2014).

[Ecl14c] Eclipse. The MoDisco project of the Eclipse Community, 2014. http://www.eclipse.org/MoDisco/ (accessed on August 2014).

[Ede05] Amnon Eden. Strategic Versus Tactical Design. In *Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS'2005), CD-ROM / Abstracts Proceedings, 3-6 January 2005, Big Island, HI, USA*, Los Alamitos, CA, USA, 2005. IEEE Computer Society Press.

[EH09] David Emery and Rich Hilliard. Every Architecture Description Needs a Framework: Expressing Architecture Frameworks Using ISO/IEC 42010. In WICSA'2009 [WIC09], pages 31–40.

[EK03] Amnon Eden and Rick Kazman. Architecture, Design, Implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'2003), 3-10 May 2003, Portland, Oregon, USA*, pages 149–159, Los Alamitos, CA, USA, 2003. IEEE Computer Society Press.

[Erw98] Martin Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages & Computing*, 9(5):461–483, 1998.

[Erw10] Martin Erwig. A Language for Software Variation Research. In Eelco Visser and Jaakko Järvi, editors, *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'2010), 10-13 October 2010, Eindhoven, The Netherlands*, pages 3–12. ACM, 2010.

[EVLL08] Jacky Estublier, Germán Vega, Philippe Lalanda, and Thomas Leveque. Domain Specific Engineering Environments. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC'2008), 3-5 December 2008, Beijing, China*, pages 553–560. IEEE Computer Society, 2008.

[Fav04a] Jean-Marie Favre. CacOphoNy: Metamodel-Driven Architecture Recovery. In *WCRE'2004, Delft, The Netherlands*, pages 204–213. IEEE Computer Society, 2004.

[Fav04b] Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. In Bézivin and Heckel [BH04].

[Fav04c] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of the Fidus Papyrus and of the Solarus. In Bézivin and Heckel [BH04].

[Fav04d] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. In Martin Gogolla, Paul Sammut, and Jon Whittle, editors, *Proceedings of the 3rd UML Workshop on Software Model Engineering (WiSME'2004) at the 7th*

International Conference on the Unified Modelling Language: Modelling Languages and Applications (UML'2004), 11-15 October 2004, Lisbon, Portugal, 2004.

[FBJ+05] Marcos Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: A Generic Model Weaver. In Sébastien Gérard, Jean-Marie Favre, Pierre Alain Muller, and Xavier Blanc, editors, *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM'2005)*, CEA LIST, June 2005.

[FBV06] Marcos Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving Models with the Eclipse AMW plugin. In *Proceedings of the Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[FCK07] Davide Falessi, Giovanni Cantone, and Philippe Kruchten. Do Architecture Design Methods Meet Architects' Needs? In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'2007), 6-9 January 2007, Mumbai, Maharashtra, India*, page 5. IEEE Computer Society, 2007.

[FGH06] Peter Feiler, David Gluch, and John Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, February 2006. `http://www.sei.cmu.edu/reports/06tn011.pdf` (accessed on August 2014).

[Fin88] Anthony Finkelstein. Re-use of Formatted Requirements Specifications. *Software Engineering Journal*, 3(5):186–197, 1988.

[FN05] Jean-Marie Favre and Tam Nguyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes Theoretical Computer Science*, 127(3):59–74, 2005.

[Fow97] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Object-Oriented Software Engineering Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 11 2002.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Lionel Briand and Alexander Wolf, editors, *Proceedings of the Workshop on the Future of Software Engineering (FOSE'2007) at the 29th International Conference on Software Engineering (ISCE'2007), 23-25 May 2007, Minneapolis, MN, USA*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[Fra02] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[FVBS09] Jared Fortune, Ricardo Valerdi, Barry Boehm, and Stan Settles. Estimating Systems Engineering Reuse. In Roy Kalawsky, John O'Brien, Thanuja Goonetilleke, and Craig Grocott, editors, *Proceedings of the 7th Conference on Systems Engineering Research (CSER'2009), 20-23 April 2009, Leicestershire, UK*. Research School of Systems Engineering, Loughborough University, 2009.

[GA02] Jeff Garland and Richard Anthony. *Large-Scale Software Architecture: A Practical Guide using UML*. John Wiley & Sons, Inc., New York, NY, USA, December 2002.

[GA11a] Matthias Galster and Paris Avgeriou. Handling Variability in Software Architecture: Problems and Implications. In *Proceedings of the joint 9th Working IEEE/IFIP Conference on Software Architecture (WICSA'2011), 20-24 June 2011, Boulder, Colorado, USA*, pages 171–180. IEEE Computer Society, 2011.

[GA11b] Matthias Galster and Paris Avgeriou. The Notion of Variability in Software Architecture: Results from a Preliminary Exploratory Study. In Patrick Heymans, Krzysztof Czarnecki, and Ulrich Eisenecker, editors, *Proceedings of the 5th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'2011), 27-29 January 2011, Namur, Belgium*, ACM International Conference Proceedings Series, pages 59–67. ACM, 2011.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, 1995.

[GAWM11] Matthias Galster, Paris Avgeriou, Danny Weyns, and Tomi Männistö. Variability in Software Architecture: Current Practice and Challenges. *ACM SIGSOFT Software Engineering Notes*, 36(5):30–32, September 2011.

[GBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the 2nd Working IEEE/IFIP Conference on Software Architecture (WICSA'2001), 28-31 August 2001, Amsterdam, The Netherlands*, pages 45–54. IEEE Computer Society, 2001.

[GCK02] David Garlan, Shang-Wen Cheng, and Andrew Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computer Programming*, 44(1):23–49, 2002.

[GG14] International Society of Grid Generation, 2014. `http://www.isgg.org` (accessed on August 2014).

[GH05] Ralf Gitzel and Tobias Hildenbrand. A Taxonomy of Metamodel Hierarchies. Technical report, Department of Information Systems, University of Mannheim, Germany, 2005. `https://ub-madoc.bib.uni-mannheim.de/993` (accessed on August 2014).

[GLZ06] Jeff Gray, Yuehua Lin, and Jing Zhang. Automating Change Evolution in Model-Driven Engineering. *IEEE Computer*, 39(2):51–58, February 2006.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In Howard Johnson, editor, *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'1997), 10-13 November 1997, Ontario, Canada*, page 7. IBM, 1997.

[Gri00] Martin Griss. Implementing Product-Line Features with Component Reuse. In William Frakes, editor, *Proceedings of the 6th International Conerence on Software Reuse: Advances in Software Reusability (ICSR'2000), 27-29 June 2000, Vienna, Austria*, volume 1844 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2000.

[GS04] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Timely, practical, reliable. Wiley, 2004.

[GS07] Hassan Gomaa and Michael Shin. Automated Software Product Line Engineering and Product Derivation. In *Proceedings of the 40th Hawaii International International Conference on Systems Science (HICSS'2007), 3-6 January 2007, Big Island, HI, USA*, page 285. IEEE Computer Society, 2007.

[HA10] Neil Harrison and Paris Avgeriou. How do architecture patterns and tactics interact? a model and annotation. *Journal of Systems and Software*, 83(10):1735–1758, 2010.

[HAH12a] Uwe van Heesch, Paris Avgeriou, and Rich Hilliard. A documentation framework for architecture decisions. *Journal of Systems and Software*, 85(4):795–820, April 2012.

[HAH12b] Uwe van Heesch, Paris Avgeriou, and Rich Hilliard. Forces on Architecture Decisions - A Viewpoint. In WICSA'2012 [WIC12], pages 101–110.

[Hau13] Øystein Haugen. CVL: Common Variability Language or Chaos, Vanity and Limitations? In Stefania Gnesi, Philippe Collet, and Klaus Schmid, editors, *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'2013), 23-25 January 2013, Pisa , Italy*, page 1. ACM, 2013.

[HAZ07] Neil Harrison, Paris Avgeriou, and Uwe Zdun. Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4):38–45, 2007.

[Hil10] Rich Hilliard. On Representing Variation. In Ian Gorton, Carlos Cuesta, and Muhammad Ali Babar, editors, *Companion volume of the 4th European Conference on Software Architecture (ECSA'2010), 23-26 August 2010, Copenhagen, Denmark*, ACM International Conference Proceeding Series, pages 312–315. ACM, 2010.

[HKN+05] Christine Hofmeister, Philippe Kruchten, Robert Nord, Henk Obbink, Alexander Ran, and Pierre America. Generalizing a Model of Software Architecture Design from Five Industrial Approaches. In WICSA'2005 [WIC05], pages 77–88.

[HMMP10] Rich Hilliard, Ivano Malavolta, Henry Muccini, and Patrizio Pelliccione. Realizing architecture frameworks through megamodelling techniques. In *ASE'2010, Antwerp, Belgium*, pages 305–308. ACM, 2010.

[HMPO+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the 12th International Conference on Software Product Lines (SPLC'2008), 8-12 September 2008, Limerick, Ireland*, pages 139–148. IEEE Computer Society, 2008.

[HNS99] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Professional, November 1999.

[HP03] Günter Halmans and Klaus Pohl. Communicating the Variability of a Software-Product Family to Customers. *Software and Systems Modeling*, 2(1):15–36, March 2003.

[HT03] Patrick Heymans and Jean-Christophe Trigaux. Software Product Lines: State of the art. Technical report, Institut d'Informatique, FUNDP, Université de Namur, Belgium, September 2003. `http://www.inf.ufpr.br/silvia/topicos/artigostrab10/artigo1-S1e2.pdf` (accessed on August 2014).

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Oviedo. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, April 2004. `http://www.sei.cmu.edu/reports/04tr008.pdf` (accessed on August 2014).

[IEE00] IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-intensive Systems, 2000.

[IEE10] IEEE Std 1517-2010 Information Technology – System and Software Life Cycle Process – Reuse Process, August 2010.

[ISO93] ISO/IEC 2382-1:1993 Information technology – Vocabulary — Part 1: Fundamental terms, 1993.

[ISO98] ISO/IEC 10746-1 Information technology – Open Distributed Processing – Reference Model: Overview, 12 1998.

[ISO00] ISO 15704 Industrial automation systems – Requirements for enterprise-reference architectures and methodologies, 2000.

[ISO03a] ISO/IEC 9126 Software engineering – Product quality, Part 2: External metrics, 2003.

[ISO03b] ISO/IEC 9126 Software engineering – Product quality, Part 3: Internal metrics, 2003.

[ISO07] ISO/IEC 42010:2007 systems and software engineering – Recommended practice for architectural description of software-intensive systems, July 2007.

[ISO09] ISO/IEC 24765:2009 Systems and software engineering – Vocabulary, 2009.

[ISO11] ISO/IEC/IEEE 42010:2011 systems and software engineering – Architecture description, November 2011.

[ISO13] ISO. Frequently Asked Questions: ISO/IEC/IEEE 42010, version 5.7, November 2013. `http://www.iso-architecture.org/ieee-1471/faq.html` (accessed on August 2014).

[ISO14] ISO/IEC/IEEE 42010 resources site, 2014. `http://www.iso-architecture.org/42010/` (accessed on August 2014).

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

[Jac04] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[JB06] Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In Roberto Gorrieri and Heike Wehrheim, editors, *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'2006), 14-16 June 2006, Bologna, Italy*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In Stan Jarzabek, Douglas Schmidt, and Todd Veldhuizen, editors, *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'2006), 22-26 October 2006, Portland, Oregon, USA*, pages 249–254, New York, NY, USA, 2006. ACM.

[Jéz04] Jean-Marc Jézéquel. A MDA Approach to Model & Implement Transformations. In Bézivin and Heckel [BH04].

[Jéz12] Jean-Marc Jézéquel. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, 2012:24, 2012.

[JFL+07] Albin Jossic, Marcos Del Fabro, Jean-Philippe Lerat, Jean Bézivin, and Frédéric Jouault. Model Integration with Model Weaving: a Case Study in System Architecture. In *Proceedings of the 1st International Conference on Systems Engineering and Modeling (ICSEM'2007), 20-23 March 2007, Herzliyya, Israel*, pages 79–84, 2007.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture,*

Process and Organization for Business Success. ACM Press books. ACM Press / Addison-Wesley Publishing Co., 1997.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Revised Selected Papers of the Satellite Events of the 8th International Conference on Model Driven Engineering, Languages and Systems (MoDELS'2005) International Workshops, Doctoral Symposium, Educators Symposium, 2-7 October 2005, Montego Bay, Jamaica*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, Heidelberg, 2005. Springer-Verlag.

[Jos14] Albin Jossic. Modeling artifacts for DoDAF, 2014. `http://www.emn.fr/z-info/atlanmod/index.php/Atlantic#DoDAF_0.1` (accessed on August 2014).

[JRL00] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[JVB+10] Frédéric Jouault, Bert Vanhooff, Hugo Brunelière, Guillaume Doux, Yolande Berbers, and Jean Bézivin. Inter-DSL coordination support by combining megamodeling and model weaving. In Sung Shin, Sascha Ossowski, Michael Schumacher, Mathew Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'2010), 22-26 March 2010, Sierre, Switzerland*, pages 2011–2018. ACM, 2010.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces: An Initial Appraisal. In Robert Meersman and Zahir Tari, editors, *Industrial track of the Confederated International Conferences DOA/CoopIS/ODBASE – On the Move to Meaningful Internet Systems, October 30 - November 1 2002 California, United States*, volume 2519 of *Lecture Notes in Computer Science*. Springer, 2002.

[KBC12] Rick Kazman, Leonard Bass, and Paul Clements. The Changing Field of Software Architecture, December 2012. Pearson InformIT, `http://www.informit.com/articles/article.aspx?p=1994790` (accessed on August 2014).

[KBWA94] Rick Kazman, Leonard Bass, Mike Webb, and Gregory Abowd. SAAM: a method for analyzing the properties of software architectures. In Bruno Fadini, Leon Osterweil, and Axel van Lamsweerde, editors, *Proceedings of the 16th International Conference on Software Engineering (ICSE'1994), 16-21 May 2001, Sorrento, Italy*, pages 81–90. IEEE Computer Society / ACM Press, 1994.

[KC95] Paul Kogut and Paul Clements. Features of Architecture Description Languages. Technical report, Software Engineering Institute, Carnegie Mellon University, April 1995.

[KCD09] Philippe Kruchten, Rafael Capilla, and Juan Dueñas. The Decision View's Role

in Software Architecture Practice. *IEEE Software*, 26(2):36–42, 2009.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990. `http://www.sei.cmu.edu/reports/90tr021.pdf` (accessed on August 2014).

[Ken02] Stuart Kent. Model Driven Engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'2002), Turku, Finland, 15-18 May 2002*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, September 2000. `http://www.sei.cmu.edu/reports/00tr004.pdf` (accessed on August 2014).

[KKL+98] Kyo Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, January 1998.

[KM04] Mika Korhonen and Tommi Mikkonen. Assessing systems adaptability to a product family. *Journal of Systems Architecture*, 50(7):383–392, 2004.

[KOS06] Philippe Kruchten, Henk Obbink, and Judith Stafford. The Past, Present, and Future for Software Architecture. *IEEE Software*, 23(2):22–30, 2006.

[KOV03] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture Reconstruction Guidelines, third edition. Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 2003. `http://resources.sei.cmu.edu/asset_files/TechnicalReport/2003_005_001_14081.pdf` (accessed on August 2014).

[Kru95] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[Kru01] Philippe Kruchten. Common Misconceptions about Software A+rchitecture. *The Rationale Edge*, April 2001.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edition, December 2003.

[Kru04] Philippe Kruchten. An Ontology of Architectural Design Decisions in Software-Intensive Systems. In Jan Bosch and Henk Obbink, editors, *Proceedings of the 2nd Groningen Workshop on Software Variability Management (SVM'2004), 2-3 December 2004, Groningen, The Netherlands*, pages 54–61, December 2004.

[Küh06] Thomas Kühne. Matters of (Meta-)Modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture – Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 2003.

[Lar02] Robert Larrabee. Software Architecture: A Maturing Discipline – book review. *IEEE Software*, 19(1):100–101, January/February 2002.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2004.

[Lim98] Wayne Lim. *Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*. Prentice Hall PTR, 1998.

[Lin02] Frank van der Linden. Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, 19(4):41–49, 2002.

[LJA09] Peng Liang, Anton Jansen, and Paris Avgeriou. Knowledge architect: A tool suite for managing software architecture knowledge. Technical Report RUG-SEARCH-09-L01, SEARCH Group, University of Groningen, The Netherlands, February 2009. `http://iwi.eldoc.ub.rug.nl/FILES/root/2009/TechRepRuGLiang/2009TechRepRuGLiang.pdf` (accessed on August 2014).

[LKA+95] David Luckham, John Kenney, Larry Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[LSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Lud03] Jochen Ludewig. Models in software engineering – an introduction. *Software and Systems Modeling*, 2:5–14, 2003.

[MAS+03] James McGovern, Scott Ambler, Michael Stevens, James Linn, Elias Jo, and Vikas Sharan. *The Practical Guide to Enterprise Architecture*. The Coad Series. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[Mat04] Mari Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In Anthony Finkelstein, Jacky Estublier, and David Rosenblum, editors, *Proceedings of the 26th International Conference on Software Engineering (ICSE'2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 127–136. IEEE Computer Society, 2004.

[MCF03] Stephen Mellor, Anthony Clark, and Takao Futagami. Guest editors' introduc-

tion: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[McI68] Malcolm McIlroy. Mass-produced software components. *Proceeding of the NATO Conference on Software Engineering, Garmisch, Germany*, 1968.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC'1995), 25-28 September 1995, Sitges, Spain*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, London, UK, UK, 1995. Springer-Verlag.

[MKMG97] Robert Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1):43–52, 1997.

[MMYJ10] John McGregor, Dirk Muthig, Kentaro Yoshimura, and Paul Jensen. Guest Editors' Introduction: Successful Software Product Line Practices. *IEEE Software*, 27(3):16–21, 2010.

[MoD08a] UK Ministry of Defence. MODAF 1.2 – Minsitry of Defence Architecture Framework Version 1.2, September 2008. `http://www.modaf.org.uk/` (accessed on August 2014).

[Mod08b] ModelPlex Project. Deliverable D2.1.a: "Global Model Management Principles", March 2008. `http://docatlanmod.emn.fr/AM3/Documentation/D2-1-a_Global_Model_Management_Principles_v1-1.pdf` (accessed on August 2014).

[Mod09] ModelPlex Project. Deliverable D2.11.b: "Global Model Management Supporting Tool", October 2009. `http://docatlanmod.emn.fr/AM3/Documentation/D2-11-b_Global_Model_Management_Supporting_Tool_v3-0.pdf` (accessed on August 2014).

[MSG96] Randall Macala, Lynn Stuckey Jr., and David Gross. Managing Domain-Specific, Product-Line Development. *IEEE Software*, 13(3):57–67, 1996.

[MSUW04] Stephen Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[MT00] Nenad Medvidovic and Richard Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[Mul13] Guerrit Muller. Light Weight Architecture: the way of the future? (version 2.3). Embedded Systems Institute, Buskerud University College – article originally written as part of the Gaudí project on 2003, March 2013. `http://www.gaudisite.nl/LightWeightArchitectingPaper.pdf` (accessed on August 2014).

[NCB+14] Linda Northrop, Paul Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, John McGregor, and Liam O'Brien. A Framework for Software Product Line Practice, version 5.0, 2014. https://www.sei.cmu.edu/productlines/tools/framework/ (accessed on August 2014).

[NFTJ03] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Requirements by Contracts allow Automated System Testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'2003), 17-20 November 2003, Denver, CO, USA*, pages 85–98. IEEE Computer Society, 2003.

[NI07] Eila Niemelä and Anne Immonen. Capturing quality requirements of product family architecture. *Information & Software Technology*, 49(11–12):1107–1120, 2007.

[NN92] Hanne Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. Wiley professional computing. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[Nor99] Linda Northrop. A Framework for Software Product Line Practice. In Ana Moreira and Serge Demeyer, editors, *Proceedings of Object-Oriented Technology, ECOOP'1999 Workshop Reader, ECOOP'1999 Workshops, Panels, and Posters, 14-18 June 1999, Lisbon, Portugal*, volume 1743 of *Lecture Notes in Computer Science*, pages 365–366. Springer, 1999.

[Nor02] Linda Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, 2002.

[NTJ06] Clémentine Nebut, Yves Le Traon, and Jean-Marc Jézéquel. System Testing of Product Lines: From Requirements to Test Cases. In Timo Käkölä and Juan Dueñas, editors, *Software Product Lines - Research Issues in Engineering and Management*, pages 447–477. Springer, 2006.

[Nus01] Bashar Nuseibeh. Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115–117, 2001.

[OG11] The Open Group. TOGAF 9.1 – The Open Group Architecture Framework Version 9.1, 2011. http://pubs.opengroup.org/architecture/togaf9-doc/arch/ (accessed on August 2014).

[OLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.

[OM07] Femi Olumofin and Vojislav Misic. A holistic architecture assessment method for software product lines. *Information & Software Technology*, 49(4):309–323, 2007.

[OMAO00] Henk Obbink, Jürgen Müller, Pierre America, and Rob van Ommering. COPA: A Component-Oriented Platform Architecting Method for Families of Software-Intensive Electronic Products. In *Tutorials of the 1st International Conference on Software Product Lines (SPLC'2000) – Experiences and Research Directions, 28-31 August 2000, Denver, Colorado, USA*, 2000.

[OMG97] OMG. Unified Modeling Language (UML) Specification, version 1.1, ad/97-08-11, August 1997.

[OMG03] OMG. Model Driven Architecture (MDA) Guide, version 1.0.1, omg/03-06-01, June 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf` (accessed on August 2014).

[OMG08] OMG. Software & Systems Process Engineering Meta-Model Specification (SPEM), version 2.0, formal/2008-04-01, April 2008. `http://www.omg.org/spec/SPEM/2.0/PDF` (accessed on August 2014).

[OMG11a] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.1, formal/2011-01-01, January 2011. `http://www.omg.org/spec/QVT/1.1/PDF/` (accessed on August 2014).

[OMG11b] OMG. Meta Object Facility (MOF) Core Specification, version 2.4.1, formal/2011-08-07, August 2011. `http://www.omg.org/spec/MOF/2.4.1/PDF` (accessed on August 2014).

[OMG11c] OMG. Unified Modeling Language™ (UML) Infrastructure, version 2.4.1, formal/2011-08-05, August 2011. `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF` (accessed on August 2014).

[OMG12] OMG. Object Constraint Language (OCL), version 2.3.1, formal/2012-01-01, January 2012. `http://www.omg.org/spec/OCL/2.3.1/PDF` (accessed on August 2014).

[OMG14a] OMG. Common Variability Language (CVL), 2014. `http://www.omgwiki.org/variability` (accessed on August 2014).

[OMG14b] OMG. Model-Driven Architecture (MDA), 2014. `http://www.omg.org/mda/` (accessed on August 2014).

[Omm02] Rob van Ommering. Building Product Populations with Sofware Components. In Will Tracz, Michal Young, and Jeff Magee, editors, *Proceedings of the 22rd International Conference on Software Engineering (ICSE'2002), 19-25 May 2002, Orlando, Florida, USA*, pages 255–265. ACM, 2002.

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[PBR09] Daniel Perovich, Cecilia Bastarrica, and Cristian Rojas. Model-Driven Approach to Software Architecture Design. In *Proceedings of the 4th Workshop on*

Sharing and Reusing Architectural Knowledge (SHARK'2009) at the 31st International Conference on Software Engineering (ICSE'2009), 16-24 May 2009, Vancouver, Canada, SHARK'2009, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[Per06] Daniel Perovich. Unveiling Models. Technical Report TR/DCC-2006-11, Computer Science Department, University of Chile, September 2006. `http://swp.dcc.uchile.cl/TR/2006/TR_DCC-2006-011.pdf` (accessed on August 2014).

[Per07] Gilles Perrouin. *Architecting Software Systems using Model Transformations and Architectural Frameworks*. PhD thesis, University of Luxemburg (LASSY) – University of Namur (PReCISE), 2007.

[Pou97] Jeffrey Poulin. *Mesuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

[PRB09] Daniel Perovich, Pedro Rossel, and Cecilia Bastarrica. Feature Model to Product Architectures: Applying MDE to Software Product Lines. In WICSA'2009 [WIC09], pages 201–210.

[PS83] Helmuth Partsch and Ralf Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[PTTT09] Päivi Parviainen, Juha Takalo, Susanna Teppola, and Maarit Tihinen. Model-driven development: Processes and practices. Technical Report VTT Working Papers 114, VTT Technical Research Centre of Finland, February 2009.

[Put00] Janis Putman. *Architecting with RM-ODP*. Prentice Hall PTR, October 2000.

[PW92] Dewayne Perry and Alexander Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[QZX08] Zheng Qin, Xiang Zheng, and Jiankuan Xing. Architectural Styles and Patterns. In *Software Architecture*, Advanced Topics in Science and Technology in China, pages 34–88. Springer Berlin Heidelberg, 2008.

[RBHK+14] Pedro Rossel, Cecilia Bastarrica, Nancy Hitschfeld-Kahler, Violeta Díaz, and Mario Medina. Domain modeling as a basis for building a meshing tool software product line. *Advances in Engineering Software*, 70:77–89, April 2014.

[Rei84] Steven Reiss. An approach to incremental compilation. *ACM SIGPLAN Notices*, 19(6):144–156, June 1984.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology. Addison-Wesley Professional, second edition, 2005.

[ROR11] Rick Rabiser, Pádraig O'Leary, and Ita Richardson. Key Activities for Prod-

uct Derivation in Software Product Lines. *Journal of Systems and Software*, 84(2):285–300, 2011.

[Rot89] Jeff Rothenberg. The nature of modeling. In Lawrence Widman, Kenneth Loparo, and Norman Nielsen, editors, *Artificial intelligence, simulation & modeling*, pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[Rot90] Jeff Rothenberg. Prototyping as Modeling: What is Being Modeled? Technical report, RAND Corporation, July 1990. RAND Note N-3191-DARPA.

[RPB09] Pedro Rossel, Daniel Perovich, and Cecilia Bastarrica. Reuse of Architectural Knowledge in SPL Development. In Stephen Edwards and Gregory Kulczycki, editors, *Proceedings of the 11th International Conference on Software Reuse (ICSR'2009) – Formal Foundations of Reuse and Domain Engineering, 27-30 September 2009, Falls Church, VA, USA*, volume 5791 of *Lecture Notes in Computer Science*, pages 191–200. Springer, 2009.

[RSM+04] Roshanak Roshandel, Bradley Schmerl, Nenad Medvidovic, David Garlan, and Dehua Zhang. Understanding Tradeoffs among Different Architectural Modeling Approaches. In WICSA'2004 [WIC04], pages 47–56.

[RW05] Nick Rozanski and Eöin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, Boston, MA, 2nd edition, 2009.

[SC97] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC'1997), 11-15 August 1997, Washington, DC, USA*, pages 6–13. IEEE Computer Society, 1997.

[SC06] Mary Shaw and Paul Clements. The Golden Age of Software Architecture. *IEEE Software*, 23(2):31–39, 2006.

[Sch86] David Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[Sch06] Douglas Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SD07] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information & Software Technology*, 49(7):717–739, 2007.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[SEI14a] SEI/CMU. Attribute-Driven Design Method, 2014. http://www.sei.cmu.edu/architecture/add_method.html (accessed on August 2014).

[SEI14b] SEI/CMU. Community Software Architecture Definitions, 2014. http://www.sei.cmu.edu/architecture/definitions.html (accessed on August 2014).

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[SH04] Mark Staples and Derrick Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'2004), 30 November - 3 December 2004, Busan, Korea*, pages 176–183. IEEE Computer Society, 2004.

[Sha90] Mary Shaw. Elements of a design language for software architecture. Position Paper for IEEE Design Automation Workshop, 1 1990.

[Sha91] Mary Shaw. Heterogeneous Design Idioms for Software Architecture. In *Proceedings of the 6th International Workshop on Software Specification and Design (IWSSD'1991), 25-26 October 1991, Como, Italy*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259–284, 2004.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Professional, Boston, MA, USA, 1st edition, 1995.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, September 2003.

[SK09] James Scott and Rick Kazman. Realizing and Refining Architectural Tactics: Availability. Technical Report CMU/SEI-2009-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, August 2009. http://www.sei.cmu.edu/reports/09tr006.pdf (accessed on August 2014).

[SME09] Rick Salay, John Mylopoulos, and Steve Easterbrook. Using Macromodels to Manage Collections of Related Models. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *Proceedings of the 21st International Conference on Ad-*

vanced Information Systems Engineering (CAiSE'2009), 8-12 June 2009, Amsterdam, The Netherlands, volume 5565 of *Lecture Notes in Computer Science*, pages 141–155, Berlin, Heidelberg, 2009. Springer-Verlag.

[Som06] Ian Sommerville. *Software Engineering*. International Computer Science. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8th edition, 2006.

[Spe00] John Spencer. Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools. Technical report, The Open Group, September 2000.

[SPL02] *Proceedings of the 2nd Software Product Lines Conference (SPLC'2002), 19-22 August 2002, California, USA*, volume 2379 of *Lecture Notes in Computer Science*. Springer, 2002.

[SPL04] *Proceedings of the 3rd International Conference on Software Product Lines (SPLC'2004), 30 August - 2 September 2004, Boston, MA, USA*, volume 3154 of *Lecture Notes in Computer Science*. Springer, 2004.

[SSR01] *Proceedings of the Symposium on Software Reusability (SSR'2001) - Putting Software Reuse in Context, 18-20 May 2001, Toronto, Ontario, Canada*. ACM, 2001.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wein, 1973.

[SV02] Klaus Schmid and Martin Verlage. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software*, 19(4):50–57, July 2002.

[SWPH09] Ina Schaefer, Alexander Worret, and Arnd Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In Goetz Botterweck, Iris Groher, Andreas Polzer, Christa Schwanninger, Steffen Thiel, and Markus Völter, editors, *Proceedings of the 1st International Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE'2009), 24 August 2009, at the 13th International Software Product Line Conference (SPLC'2009), 24-28 August 2009, California, US*, page 14, Pittsburgh, PA, US, 2009. Carnegie Mellon University.

[TA05] Jeff Tyree and Art Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005.

[TABGH05] Antony Tang, Muhammad Ali Babar, Ian Gorton, and Jun Han. A Survey of the Use and Documentation of Architecture Design Rationale. In WICSA'2005 [WIC05], pages 89–98.

[TH02] Steffen Thiel and Andreas Hein. Systematic Integration of Variability into Product Line Architecture Design. In SPLC'2002 [SPL02], pages 130–153.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'2009), 23-26 June 2009, Enschede, The Netherlands*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[TLY12] Lei Tan, Yuqing Lin, and Huilin Ye. Quality-Oriented Software Product Line Architecture Design. *Journal of Software Engineering and Applications*, 5(7):5, July 2012.

[TMD09] Richard Taylor, Nenad Medvidović, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 1st edition, January 2009.

[TPT09] Susanna Teppola, Päivi Parviainen, and Juha Takalo. Challenges in deployment of model driven development. *International Conference on Software Engineering Advances*, 0:15–20, 2009.

[Tra88] Will Tracz. Software reuse myths. *ACM SIGSOFT Software Engineering Notes*, 13(1):17–21, January 1988.

[VCFM10] Juan Vara, Maria De Castro, Marcos Del Fabro, and Esperanza Marcos. Using weaving models to automate model-driven web engineering proposals. *International Journal of Computer Applications in Technology*, 39(4):245–252, October 2010.

[VG07] Markus Völter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th International Conference on Software Product Lines (SPLC'2007), 10-14 September 2007, Kyoto, Japan*, pages 233–242. IEEE Computer Society, 2007.

[VJBB09] Andrés Vignaga, Frédéric Jouault, Cecilia Bastarrica, and Hugo Brunelière. Typing in Model Management. In Richard Paige, editor, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'2009), 29-30 June 2009, Zurich, Switzerland*, volume 5563 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2009.

[VPB08] Andrés Vignaga, Daniel Perovich, and Cecilia Bastarrica. Extracting Object Interactions Out of Software Contracts Using Model Transformations. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT'2008), 1-2 July 2008, Zürich, Switzerland*, volume 5063 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2008.

[VTT14] VTT Technical Research Centre of Finland. Quality-driven Architecture Design and quality Analysis, 2014. `http://www.vtt.fi/qada` (accessed on August 2014).

[WBB+06] Rob Wojcik, Felix Bachmann, Leonard Bass, Paul Clements, Paulo Merson, Robert Nord, and Bill Wood. Attribute-Driven Design (ADD), Version 2.0. Technical Report CMU/SEI-2006-TR-023, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 2006. `http://www.sei.cmu.edu/reports/06tr023.pdf` (accessed on August 2014).

[WIC04] *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'2004), 12-15 June 2004, Oslo, Norway*. IEEE Computer Society, 2004.

[WIC05] *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'2005), 6-10 November 2005, Pittsburgh, PA, USA*. IEEE Computer Society, 2005.

[WIC09] *Proceedings of the joint 8th Working IEEE/IFIP Conference on Software Architecture (WICSA'2009) and 3rd European Conference on Software Architecture (ECSA'2009), 14-17 September 2009, Cambridge, UK*. IEEE Computer Society, 2009.

[WIC12] *Proceedings of the joint 10th Working IEEE/IFIP Conference on Software Architecture (WICSA'2012) and 6th European Conference on Software Architecture (ECSA'2012), 20-24 August 2012, Helsinki, Finland*. IEEE Computer Society, 2012.

[WL99] David Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Zac87] John Zachman. A Framework for Information Systems Architecture. *IBM Systems*, 26(3):276–292, 1987.

[ZPJ+11] Jiayi Zhu, Xin Peng, Stan Jarzabek, Zhenchang Xing, Yinxing Xue, and Wenyun Zhao. Improving Product Line Architecture Design and Customization by Raising the Level of Variability Modeling. In Klaus Schmid, editor, *Proceedings of the 12th International Conference on Software Reuse (ICSR'2011) – Top Productivity through Software Reuse, 13-17 June 2011, Pohang, South Korea*, volume 6727 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2011.