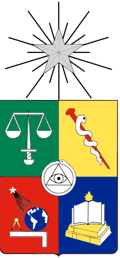


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION

SOFTWARE PRODUCT LINE MODEL FOR THE MESHING TOOL DOMAIN

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

PEDRO OSVALDO ROSSEL CID

PROFESOR GUIA:
MARÍA CECILIA BASTARRICA PIÑEYRO
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISION:
MARÍA CECILIA RIVARA
ROMAIN ROBBES
REINER HÄHNLE

Este trabajo ha sido parcialmente financiado por el proyecto MECESUP n.° UCH 0109

SANTIAGO DE CHILE
MARZO 2013

Resumen

Una malla es una discretización de la geometría de un cierto dominio. Las mallas pueden estar compuestas de diversos elementos: triángulos, cuadriláteros, tetraedros, etc. Una herramienta para la generación de mallas es un aplicación que permite crear, refinar, desrefinar, mejorar, suavizar, visualizar y posprocesar mallas y/o una región particular de ella, como también asignar valores físicos a los elementos de la malla (temperatura, concentración, etc.).

Las herramientas para la generación de mallas son complejas y sofisticadas, y construir una herramienta nueva desde cero o mantener una existente, demanda un esfuerzo enorme. Existe una necesidad y oportunidad para usar enfoques nuevos en el desarrollo de estas herramientas, de manera de reducir tanto el tiempo como los costos de desarrollo, sin comprometer la calidad. La experiencia en el desarrollo de estas herramientas provee la motivación para la construcción de otras nuevas mediante la reutilización del trabajo realizado durante los desarrollos previos. Estas herramientas comparten varias características y sus variaciones pueden ser manejadas sistemáticamente. Esto hace que el desarrollo de estas herramientas sea una buena oportunidad para aplicar el enfoque de Línea de Productos de Software (LPS). Los procesos existentes de LPS son generales y requieren usualmente una serie de pasos y documentación innecesaria en el dominio de las herramientas para la generación de mallas. Así, esta tesis propone un modelo de proceso de LPS específico para este tipo de herramientas.

Un proceso de desarrollo de LPS está centrado en la reutilización de software, e involucra principalmente dos fases: la ingeniería del dominio (ID) y la ingeniería de la aplicación (IA). El proceso presentado en este trabajo está centrado en dos etapas de la ID: el análisis del dominio (AD) y el diseño del dominio (DD). En el AD se define el modelo del dominio y el alcance de la LPS. En el DD la arquitectura de la línea de productos (ALP) es creada; esta arquitectura es válida y compartida por todos los productos en la LPS.

Un modelo de características es comúnmente usado para modelar el dominio. En este trabajo, el AD también ocupa un diccionario, escenarios, acciones y metas para proveer el razonamiento utilizado para la construcción del modelo de características. Esta tesis presenta un proceso riguroso para obtener el modelo del dominio. Este modelo es formalizado mediante condiciones de consistencia y completitud. El proceso de definición del alcance es presentado a través de un diagrama de actividad. Además, el enfoque presentado en esta tesis presenta explícitamente los diferentes productos de la LPS, estableciendo relaciones entre productos y las características de la LPS, lo que permite administrar el desarrollo del producto.

La etapa de DD se centra en la creación de la ALP, artefacto esencial para la construcción de productos de la LPS. Para ello, este trabajo provee un proceso deductivo y otro transformacional. En el primero, una ALP explícita es desarrollada, usando los artefactos producidos en el AD. Además, tanto la vista arquitectónica estructural como la de comportamiento son establecidas. Ambas vistas son generales y permiten la representación de cualquier producto dentro del alcance de la LPS. En el proceso transformacional, una ALP implícita es desarrollada usando reglas de transformación, las que han sido creadas usando artefactos producidos en el AD. En este proceso se produce la arquitectura para productos específicos, y la ALP es definida como la suma de todas las arquitecturas de los productos.

Tanto el AD como el DD son descritos en detalle, y la aplicación del modelo de la LPS es ilustrado a través de un ejemplo bien documentado en el dominio de las herramientas para la generación de mallas, el que tiene un grado relativamente alto de complejidad. En este ejemplo, un modelo del dominio formalizado es introducido, y la arquitectura es definida tanto para el proceso deductivo como para el transformacional.

Abstract

A mesh is a discretization of a certain geometry of varying dimensions. Meshes may be composed of different elements, e.g., triangles, quadrilaterals, tetrahedra, etc. A meshing tool is a software application that allows creating, refining, derefining, improving, smoothing, visualizing and postprocessing meshes and/or particular mesh regions, and also assigning physical values to mesh elements (temperature, concentration, etc.) depending on their intended use.

Meshing tools are complex and sophisticated software, and building a new tool from scratch as well as evolving an existing tool demands an enormous effort. There is a need and an opportunity for new approaches in meshing tool software development in order to reduce development time and costs without compromising quality. Experience developing meshing tools has provided the motivation for building new ones by reusing a large amount of the work done during those developments. Meshing tools share several characteristics and their variations can be managed in a systematic way. This makes meshing tool development an appropriate opportunity for applying Software Product Lines (SPL). Existing processes for engineering SPL are needlessly general and usually require a series of steps and documentation not necessary in the meshing tool domain. Therefore, this thesis proposes a SPL process model specifically for the meshing tool domain.

A SPL development process is centered in software reuse, and mainly involves two phases: domain engineering (DE) and application engineering (AE). The process presented in this work is centered in two stages of the DE phase: the domain analysis (DA) and the domain design (DD) stages. In the DA stage, the domain model and the scope of the SPL are defined. In the DD stage, the product line architecture (PLA) is produced; this architecture is valid and shared for all particular products in the SPL.

A feature model is commonly used to model the domain. In this work, the DA stage also uses a lexicon, scenarios, actions and goals that provide the rationale for building the feature model. This thesis presents a rigorous process for collecting SPL features, through the identification of the aforementioned artifacts. Furthermore, the domain model is formalized by providing consistency and completeness conditions. The scope definition process is presented through an activity diagram. Furthermore, the approach presented in this thesis explicitly presents the different products of the SPL, establishing relationships between products and the SPL features, which permits managing the product development.

The proposed DD stage focuses on the creation of the PLA, an essential artifact for building specific SPL products. To this purpose, this work provides a deductive process and a transformational one. In the deductive process, an explicit PLA is developed using the artifacts produced in the DA stage. Furthermore, both architectural structural and architectural behavioral views are established for meshing tool SPL. Both views are general and allow representing any product in the SPL scope. In the transformational process, an implicit PLA is developed using transformation rules, which have been created using the artifacts produced in the DA stage. In this process, only single product architectures are produced, and the PLA is defined as the sum of all single architectures.

Both DA and DD stages are described in detail, and the SPL model application is illustrated through a suitably documented example with a relatively high degree of complexity in the meshing tool domain. In this example, a formalized domain model is introduced, and architectures are defined using both the deductive and transformational processes.

A mi esposa Claudia,
y a mis hijos
Bruno,
Catalina y
José Tomás.

Agradecimientos

En primer lugar quiero agradecer a Dios y a la Mater por acompañarnos a mí a y mi familia en este largo viaje, por estar siempre con nosotros, aunque no nos diéramos cuenta.

A Claudia, mi esposa, porque ella no solo es el componente más importante en este desarrollo, sino que también es el *glue code* que nos une como familia. Gracias por soportarme, en todas las acepciones que esa palabra tiene.

A mis profesoras guías Cecilia y Nancy, por apoyarme en todo sentido, por la paciencia que tuvieron, por su dedicación, por compartir su conocimiento, por darme una palabra de ánimo o simplemente conversar.

A Mario Medina, por su generosidad. Sin su ayuda esta tesis no estaría tan bien escrita, en "perfecto inglés". La tesis está llena de lujos idiomáticos que dudo aprenderé a escribir, pero creo que hoy sé un poco más de inglés gracias a él.

A MECESUP, que a través del proyecto n.° UCH 0109 me apoyó económicamente durante gran parte de mi doctorado. También al Departamento de Postgrado y Postítulo de la Vicerrectoría de Asuntos Académicos de la Universidad de Chile, que me permitió pasar tres meses en el *Generative Software Development Lab* de la Universidad de Waterloo, Canadá.

A la Universidad Católica del Maule, que me dio la oportunidad de salir a perfeccionarme, y a la Universidad Católica de la Santísima Concepción, que me permitió terminar mis estudios.

A Valeria Herskovic, mi amiga y compañera de oficina, por ayudarme a terminar mi tesis. Siempre me sentí alentado a trabajar estando con ella. Creo firmemente que si no me hubiera cambiado de oficina ambos hubiésemos terminado nuestras tesis antes. Espero que sigamos trabajando juntos.

A Gilberto Gutiérrez, por alentarme a terminar mi tesis a través de su método "al menos 3 páginas por día". Lamento decir que con suerte fue solo 0,6 páginas por día.

A Krzysztof Czarnecki, que me recibió en su laboratorio en la Universidad de Waterloo. Él es mucho mejor persona que investigador, y como investigador es gigante.

A mis amigos Andrés Vignaga y Daniel Perovich por su ayuda, por las conversaciones, por su amistad y por su alegría.

A Angélica Aguirre, por su buena disposición y ayuda en sacar adelante este trabajo.

A todos, ¡muchas gracias!

Contents

Contents i

List of Tables v

List of Figures vi

List of ATL Rules viii

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Meshing Tools . . . 2
  - 1.1.2 The Opportunity . . . 4
  - 1.1.3 Software Product Lines . . . 5
- 1.2 Proposal . . . 6
  - 1.2.1 Hypothesis . . . 6
  - 1.2.2 Objectives . . . 7
    - 1.2.2.1 General Objective . . . 7
    - 1.2.2.2 Specific Objectives . . . 7
  - 1.2.3 Overview . . . 7
  - 1.2.4 Justification . . . 9
  - 1.2.5 Methodology . . . 10
  - 1.2.6 Results of the Thesis . . . 11
- 1.3 Document Structure . . . 12

2 Software Product Lines 13
- 2.1 Context: Software Reuse . . . 13
- 2.2 Software Product Lines . . . 15
- 2.3 Differences Between a SPL and Single System Development . . . 19
- 2.4 Advantages and Disadvantages of Using a SPL Approach . . . 20
- 2.5 SPL Stages . . . 23
  - 2.5.1 Domain Analysis . . . 27
  - 2.5.2 Domain Design . . . 28
  - 2.5.3 Domain Implementation . . . 28
  - 2.5.4 Application Requirements . . . 29
  - 2.5.5 Application Design . . . 30
  - 2.5.6 Application Coding . . . 30
- 2.6 Domain-Specific Languages . . . 31

3 Meshing Tool Domain **35**
3.1 Mesh Generation . . . 35
3.2 Mesh Generation Process . . . 37
3.3 Types of Meshes . . . 38
3.4 Algorithms for Mesh Generation . . . 39
3.4.1 Quadtree and Octree methods . . . 40
3.5 Algorithms for Generating an Optimal Mesh . . . 42
3.5.1 Refinement Algorithms . . . 43
3.5.2 Derefinement Algorithms . . . 44
3.5.3 Improvement Algorithms . . . 46
3.5.4 Optimization Algorithms . . . 47
3.6 Input/Output File Format . . . 48
3.6.1 Object File Format (off) . . . 49
3.6.2 Comsol Format . . . 49
3.7 Postprocessing Algorithms . . . 49
3.7.1 Triangle to Quadrilateral . . . 50
3.7.2 Quadrilateral to Triangle . . . 50
3.8 Visualization . . . 51
3.9 Developing Meshing Tools . . . 51

4 Domain Analysis for the Meshing Tool Domain **53**
4.1 The Domain Analysis Stage . . . 53
4.1.1 Domain Model . . . 54
4.1.2 Scope . . . 55
4.2 Domain Analysis Method for the Meshing Tool Domain . . . 60
4.2.1 Domain Model Construction Process . . . 60
4.2.1.1 Stakeholders . . . 61
4.2.1.2 Artifacts . . . 61
4.2.1.3 Activities . . . 63
4.2.2 Domain Model Definition . . . 64
4.2.2.1 Domain Model Formal Definition . . . 65
4.2.2.2 Consistent Domain Model Formal Definition . . . 67
4.2.3 Scope Construction Process . . . 67
4.2.3.1 Artifacts . . . 68
4.2.3.2 Activities . . . 70
4.2.4 Related Work . . . 71
4.3 Contributions of this Chapter . . . 72

5 Applying the Domain Analysis Method **74**
5.1 Meshing Tool Domain Model . . . 74
5.1.1 Lexicon . . . 75
5.1.2 Goals . . . 77
5.1.2.1 Business goal . . . 78
5.1.2.2 Goals . . . 78
5.1.3 Scenarios and actions . . . 78
5.1.4 Features . . . 80
5.1.4.1 Feature diagram . . . 81

5.1.4.2 Feature constraints . . . 83
5.1.5 Domain Model Consistency . . . 84
5.1.6 Domain Model Completeness . . . 85
5.1.6.1 Triangle . . . 85
5.1.6.2 TetGen . . . 86
5.1.6.3 Domain Model Completeness Evaluation . . . 86
5.2 Meshing Tool Scope . . . 87
5.3 Contributions of this Chapter . . . 89

6 Domain Design for the Meshing Tool Domain 92
6.1 Generalities . . . 92
6.1.1 Product Architecture . . . 92
6.1.1.1 Quality Attributes . . . 93
6.1.1.2 Viewtypes, styles and views . . . 94
6.1.2 Product Line Architecture . . . 95
6.1.2.1 Architecture Description Language (ADL) . . . 96
6.1.2.2 PLA Assessment . . . 99
6.2 Domain Design Method in the Meshing Tool Domain . . . 100
6.2.1 Deductive Process . . . 101
6.2.1.1 Quality Analysis . . . 102
6.2.1.2 Architectural Views Generation . . . 103
6.2.2 Transformational Process . . . 105
6.2.2.1 Domain Analysis . . . 107
6.2.2.2 Domain Design . . . 108
6.2.2.3 Application Requirements . . . 109
6.2.2.4 Application Design . . . 110
6.3 Contributions of this Chapter . . . 112

7 Applying the Domain Design Process 113
7.1 Deductive Process . . . 113
7.1.1 Quality Attribute Deduction . . . 113
7.1.2 Meshing Tools Structural View . . . 114
7.1.3 Meshing Tools Behavioral view . . . 116
7.2 Transformational Process . . . 118
7.2.1 Domain Analysis . . . 118
7.2.2 Domain Design . . . 118
7.2.3 Application Requirements . . . 120
7.2.4 Application Design . . . 120
7.2.5 Adding Quality Attributes to the Feature Model . . . 121
7.3 Contributions of this Chapter . . . 123

8 Conclusions and Future Work 125
8.1 Results . . . 125
8.1.1 Objectives . . . 125
8.1.2 Publications . . . 126
8.2 Contributions of this thesis . . . 127
8.3 Knowledge Transfer to Other Domains . . . 130

8.3.1 Domain Analysis 130
8.3.2 Domain Design 131
8.4 Future Work 131

References **134**

Appendices **153**

A Meshing Tool Derivation Process **154**
A.1 Introduction 154
A.2 Products Derivation Process 157
A.2.1 Considerations 157
A.2.2 Activity Diagram 158
A.2.3 Applications in the SPL Scope 158
A.2.4 Applications out of the SPL Scope 162
A.2.5 Domain Design Artifacts Missing 162
A.2.6 Domain Implementation Artifacts Missing 163
A.2.7 About Feedback 163

B Approaches for Building a SPL **164**
B.1 PuLSE 164
B.1.1 Deployment Phases 165
B.1.2 Technical Components 166
B.1.3 Support Components 167
B.2 Framework for Software Product Line Engineering 168
B.2.1 Product Management 171
B.2.2 Domain Requirements Engineering 171
B.2.3 Domain Design 171
B.2.4 Domain Realization 171
B.2.5 Domain Testing 172
B.2.6 Application Requirements Engineering 172
B.2.7 Application Design 172
B.2.8 Application Realization 172
B.2.9 Application Testing 173
B.3 Comments About the Approaches 173

C ATL Transformation Rules **176**
C.1 Rules for the Structural View 176
C.2 Rules for the Behavioral View 177

Index **180**

List of Tables

1.1 Costs of different commercial Meshing Tools 3

4.1 Product Map 68

5.1 Constraints among features 83
5.2 Relationships between Scenarios and Goals 84
5.3 Relationship between Actions and Features 84
5.4 Product Map for 2D Meshing Tools 90
5.5 Product Map for 3D Meshing Tools 91

7.1 Correspondence between Goals and Quality Attributes 113

List of Figures

1.1 Mesh simulating a Brain and MRI Image 3
1.2 Geometry of an Insulated Gate Bipolar Transistor 4
1.3 Domain and Application Engineering for Meshing Tool Domain 8

2.1 A view of a simple product line 17
2.2 Variability handling 17
2.3 Costs Comparison between Single System and Product Line Development . . 20
2.4 Time to Market comparison between Single System and Product Line Development 21
2.5 Activities for SPL 23
2.6 Impact of technology on the SPL phases 24
2.7 Software Development based on Domain Engineering 25
2.8 The Praise Reference Process 26

3.1 Triangle, Quadrilateral, Tetrahedron and Hexahedron 37
3.2 Structured and Unstructured Mesh 38
3.3 Conversion between Triangles and Quadrilaterals 39
3.4 Quadtree and Octree 40
3.5 Unbalanced Quadtree (a), and Balanced Quadtree (b) 41
3.6 Steps for Discretization of a 2D Domain 42
3.7 Triangle Partitions 43
3.8 Vertex Remove Operation 45
3.9 Different Triangulations for a Pentagon 45
3.10 Edge Collapse Operation 45
3.11 Element Open Operation 46
3.12 Diagonal Swap Operation 46
3.13 Laplacian Smoothing Algorithm 47
3.14 Mesh to be Represented in Different Formats 48
3.15 Example of a Mesh in Object File Format 49
3.16 Example of a Mesh in Comsol Format 50
3.17 Transformation from a Triangle to Quadrilaterals 50
3.18 Transformation from a Quadrilateral to Triangles 50

4.1 Domain Analysis from Existing Products 54
4.2 Generic Scoping Process 57
4.3 The scope of an SPL 58
4.4 Evolution of a SPL Scope 59
4.5 Domain Model Artifacts 62

4.6 Domain Model Construction Process 64
4.7 Scoping Construction Process 70

5.1 A Meshing Tool Domain Feature Diagram 81
5.2 A Meshing Tool Domain Feature Diagram, continuation 82
5.3 A Meshing Tool Domain Feature Diagram, continuation 82
5.4 A Meshing Tool Domain Feature Diagram, continuation 82
5.5 Added features according to completeness evaluation 87

6.1 Viewtypes and Styles 94
6.2 Overview of Domain Design Process 101
6.3 Domain Design Process with Explicit Architecture 102
6.4 Generate Architectural Views activity 104
6.5 Feature Model Metamodel 108
6.6 Domain Design Process with Implicit Architecture 109
6.7 Feature to Architecture Transformation Rule Metamodel 110
6.8 Feature Model Configuration Metamodel 110
6.9 Single Product Architecture Structural View Metamodel 111
6.10 Single Product Architecture Behavioral View Metamodel 111

7.1 Structural View 114
7.2 User Interface component 115
7.3 Algorithm component 115
7.4 Behavioral View 117
7.5 Feature-to-Architecture Transformation Rule for the Meshing Tool Feature, Structural View Part 119
7.6 Feature-to-Architecture Transformation Rules, Behavioral View Part 119
7.7 Feature Model Configuration for a Meshing Tool 120
7.8 Product Architecture Structural View - Fragment for Meshing Tool Feature 121
7.9 Product Architecture Behavioral View 122
7.10 Meshing Tool Feature Diagram Simplification 123
7.11 Distributed Product Architecture Structural View 124

A.1 Relationship between Domains and Products 154
A.2 Project-specific derivation models 155
A.3 Product Derivation Process 159
A.4 Check Requirements in relation to SPL Scope 160
A.5 Application Design Instantiation 160
A.6 Generate Application 161

B.1 PuLSE Overview 165
B.2 PuLSE Usage Phase 166
B.3 The Two-life-cycle Model of Software Product Line Engineering 168
B.4 Graphical Notation for Representing Variability 169
B.5 BAPO Model 170

List of ATL Rules

C.1 CSG Rule for the Structural View 176
C.2 Generate initial mesh Rule for the Structural View 176
C.3 Refine Rule for the Structural View 177
C.4 Meshing Tool Rule for the Behavioral View 177
C.5 CSG Rule for the Behavioral View 178
C.6 Generate initial mesh Rule for the Behavioral View 178
C.7 Refine Rule for the Behavioral View 178
C.8 Output Rule for the Behavioral View 178

Chapter 1

Introduction

1.1 Motivation

Developing any complex software from scratch on a one-by-one basis is expensive, slow and error prone, but nonetheless this is how many applications from diverse domains have traditionally been built. If this development task is not performed systematically following good software engineering practices, it may easily get out of control making it almost impossible to debug and even more difficult to modify.

`Meshing tool`

Meshing tools are highly complex software for generating and managing geometrical discretizations. Because of this, they have generally been developed by end users following ad-hoc methodologies and not by applying well-established software engineering practices. New tools are generally developed from scratch without taking software reuse in mind, even though they may involve algorithms and data structures already designed, implemented and tested elsewhere.

Lately, there has been considerable effort into applying software engineering concepts to meshing tool development, mainly through building general-purpose libraries that facilitate reuse. Also, object-orientation and design patterns have the potential to enhance software reuse at the code and design levels, and there is some experience in using these techniques for developing meshing tools.

`Software product line`

Software product lines is a development paradigm for planned massive reuse of software assets. These reusable assets are typically software components, but other commonly reused assets are software requirements, documentation, architecture, and test cases, among others.

Despite software reuse's undeniable benefits, it is not as widely practiced as it should. There are organizational, economical, and/or technical factors that directly or indirectly influence its adoption.

In the following sections, meshing tools and software product lines are presented, and the

opportunity for improving the methodological approach to meshing tool development tools using software product lines is discussed. Then, the thesis hypothesis, objectives, justification and methodology are presented. Finally, thesis results are presented and the structure of the thesis is shown.

1.1.1 Meshing Tools

A meshing tool is software for generating and managing meshes. A bidimensional mesh (2D) is composed of vertices defined by points, edges defined by segments and elements defined by triangles and/or quadrilaterals. A three dimensional mesh (3D) has also faces defined by triangles and/or quadrilaterals and the elements are tetrahedra and/or hexahedra. An element represents the smallest discretization unit.

Meshes

Meshing tools are inherently sophisticated software due to the complexity of the application requirements they have to fulfill. Examples of requirements are to model as exact as possible the geometry of the domain and to generate elements that satisfy geometric quality criteria. Meshing tools must possess specific functionality while still having an acceptable performance. Managing thousands and even millions of elements with a reasonable use of computational resources –mainly processor time and storage– becomes a must if the tool is to be usable at all. Lately, other qualities related to modifiability have also become relevant in meshing tool development.

Because of their complexity, the price of meshing tools can be high. This is mainly because these tools are used by people of diverse knowledge areas, and therefore it is necessary to consider several points of view and skills as development requirements. Furthermore, mesh generation algorithms are complex, and developers must have high programming skills to develop efficient and robust code, and enough mathematic background to be able to understand the application requirements.

Table 1.1 shows a list of general purpose commercial meshing tools, along with their corresponding license costs: GiD[1], ADINA User Interface[2], ANSA[3], Surfgen[4] and CM2 MeshTools[5]. The list of tools were obtained from Schneiders [200], who maintains a list of more than sixty commercial and eighty public domain, downloadable and university meshing tools. Single user license costs were obtained from each company's salespeople. In some cases, such as ADINA User Interface, there is a yearly maintenance fee after the first year; in others, licenses have an unlimited duration.

Meshing tool price

It is well known that any software development has risks, and these risks are a threat for the project's success. Moreover, every software project is error-prone, and meshing tool

[1]Price obtained from Mónica Fraile, Administration and Sales department, Compass Ingeniería y Sistemas. June 2011.

[2]Price obtained from Arash Mahdavi, ADINA R&D, Inc. June 2011.

[3]Price obtained from Thomas Hasiotis, Chief Financial Officer, BETA CAE Systems S.A. June 2011.

[4]Price obtained from Gayle Ingalls, Senior Compliance Officer, Analytical Methods, Inc. June 2011.

[5]Price obtained from Damien Lucas, Manager, Computing Objects. June 2011.

Name	Educational Licence (US$)	Commercial Licence (US$)
GiD	675	1.931
ADINA User Interface	1.400	9.702
ANSA	1.436	14.359
Surfgen	5.000	15.000
CM2 MeshTools	14.392	35.980

Table 1.1: Costs of different commercial Meshing Tools

development is not an exception. The algorithms used in these tools are in some cases very complex, must manage very large quantities of data, data processing is CPU and memory intensive, and subject to quality requirements regarding precision, exactitude and correctness.

Application domains

There are many application domains where meshing tools are used, ranging from mechanics design to medicine [78]. Each domain requires slightly different functionalities. For this reason, a variety of meshing tools have been built differing in their functionalities, algorithms, data representation, or input/output data format [200].

For example, during a brain tumor removal intervention, the accurate localization of the tumor is essential: small errors may have a large impact on patients and morbidity could increase dramatically. Once the skull is open, a brain deformation (brain-shift) occurs, such that pre-operative images do not correspond to the new conditions. Meshes can be used to simulate this brain-shift with the help of magnetic resonance imaging (MRI).

Figure 1.1 shows a final mesh used to simulate the brain-shift (a), with the initially scanned MRI image (b) [131].

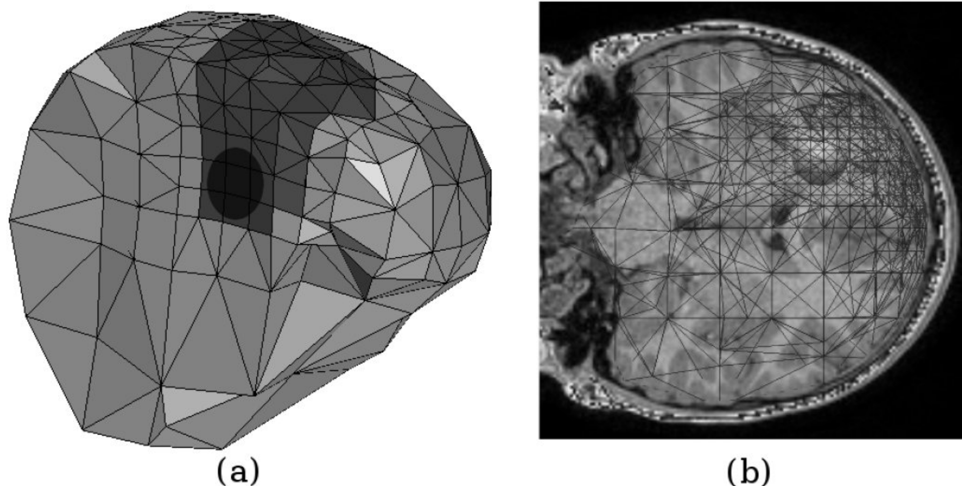


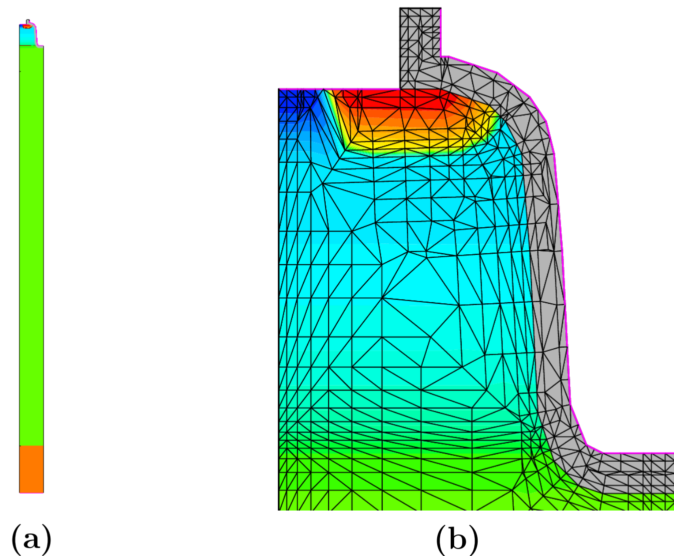
Figure 1.1: Mesh simulating a Brain and MRI Image

The semiconductor device simulation domain is another example where precision, exactitude and correctness is paramount.

The idea is to design and test a semiconductor device through a simulation process before

building it, thus avoiding or reducing the costs of using a real device. In this way, it is possible to design and test the semiconductor device with several different properties as many times as required, so that finally only satisfactory semiconductor devices will be built and experimentally tested. To this end, devices are simulated using meshes, usually representing them through very thin layers due to their geometry, and their physical properties [100].

Figure 1.2 shows the geometry of a semiconductor device (a) and the corresponding mesh for the device (b) [100].



(a) **(b)**

Figure 1.2: Geometry of an Insulated Gate Bipolar Transistor

1.1.2 The Opportunity

There is a need and an opportunity for new approaches in meshing tool software development in order to reduce development time and costs without compromising quality [18, 81, 212].

Meshing tools are very sophisticated, and therefore it would be advantageous for the product and the process to be able to reuse existing assets.

Experience developing meshing tools [18, 56, 99, 100, 101, 137, 141, 186, 210] has provided the motivation for building new ones by reusing all or a large amount of the work done during those developments. Meshing tools share several characteristics and it is possible to manage their variations in an established and systematic way. Smith and Chen [212] give a simple yet clarifying opinion about meshing tools: "although different in the specific details, all mesh generators can be abstracted as: input information then calculate a mesh discretization and finally output the results". These issues show that it is possible to abstract a model from the meshing tool domain that allows reuse. If a reuse oriented approach is used instead of a single system development approach, then these commonalities can be developed only once.

As an example, all mesh generation processes share the following main steps [18]:

Mesh generation steps

- generation of an initial mesh that fits the domain geometry;
- generation of an intermediate mesh that satisfies the density requirements specified by the user;
- generation of an improved mesh that satisfies the quality criteria;
- generation of the final mesh.

Therefore, these are some of the main commonalities among all members of a mesh generation tool family.

For Czarnecki [58], research and practical experience in software development suggest that achieving significant progress in software reuse requires centering efforts on modeling and developing a collection of similar software systems rather than individual systems.

According to Arango [4], "the more restricted the domain, the easier it is to develop a model of it". Meshing tool development is a well-bounded domain, small enough that software reuse can be applied with confidence, and at the same time general enough that its applications encompass several areas of science and engineering. Thus, the final products can be generated from high-level specifications. Automation is possible because the modeling language and generator only need to fit the requirements of one domain [231].

1.1.3 Software Product Lines

Currently, there are several techniques for reducing the development time and costs of software systems such as Agile Software Development, Prototyping, Component-based Software Engineering, among others [217]. However, there is no unique standard development method applicable to all problems or domains. It is possible to obtain better improvements by taking advantage of the specific characteristics of particular application domains [86].

`Software product line`

One of these techniques is Software Product Lines (also known as Software Families). According to Northrop and Clements [157], a software product line (SPL) is a set of software intensive systems that share a managed set of characteristics, and that satisfies the needs of a particular market segment or mission, being developed using a set of common core assets in a preestablished fashion. These core assets include the product line architecture, reusable software components, and domain models, among others. In a SPL, two main technical phases can be identified [50, 60, 174]: domain engineering, where reusable core assets are developed and maintained, and application engineering, where particular products are built by combining the assets already developed. Understanding and identifying both common and variable aspects play a central role during the domain engineering stage. Commonalities are requirements that must hold for all products in the SPL, while variabilities are requirements that may or may not be present in a particular product, and, as such, define how SPL products may vary [246].

`SPL phases`

Both domain engineering and application engineering have several substages or subprocesses, that differ in number or name depending on the author [60, 157, 174]. For Czarnecki and Eisenecker [60], the domain engineering has the following substages: domain analysis, domain design, and domain implementation; the application engineering has the following substages: requirements analysis, product configuration, and integration and test. However, all authors agree on the importance of the first stages. In particular, the most important stages are the Domain Analysis stage, as it is where the stakeholders define the limits of the SPL and establish the commonalities and variabilities present in it, and the Domain Design stage, where the general architecture shared by all products in the SPL is defined.

`Domain engineering substages`

`Application engineering substages`

A SPL is useful for an organization because it reduces both time and costs of production of new software and increases the quality of the released products by reusing core assets which have been already tested. The cost of developing and maintaining core assets is not borne by each product separately, but spread across all products in the SPL [90].

`SPL benefits`

This issue marks the difference between a development approach for a single product and for a product line. Several authors agree that a product line has a high upfront investment (development of reusable core assets, transformation of the organization, etc.) in comparison to a single product, but that the SPL's return on investment is higher when producing three products or more [30, 107, 157, 197, 236].

In spite of all its advantages, SPL is not a "silver bullet"; it has been known for several years that its applicability in a concrete domain is not easy, specially for small organizations, because of the amount of organizational changes required, high initial investment, need for trained professionals, etc.

Although commonalities among different tools [20, 212] show that it is possible to produce meshing tools by reusing well developed assets within a SPL approach [19, 20], this is not the way such tools have been developed in the past, as can be seen by reviewing the state of the art [104, 200]. Even though their performance and functionality are considered their most relevant qualities, it is also known that good software engineering techniques are needed as a means for maintainability. These software products are generally difficult to maintain, and slight variations in product requirements may need building the product again, with the consequent rework [104].

1.2 Proposal

1.2.1 Hypothesis

This thesis aims to prove the following hypothesis:

Hypothesis: Proper Domain Analysis (DA) and Domain Design (DD) processes in the SPL approach enable an effective means for developing Meshing Tools.

1.2.2 Objectives

1.2.2.1 General Objective

Build effective DA and DD processes for the Meshing Tool domain.

1.2.2.2 Specific Objectives

1. Identify the characteristics of meshing tools and of the way they are built.
2. Develop a DA process, describing the sequence of activities, artifacts and roles involved.
3. Identify architectural needs of meshing tools, considering the shared quality attributes.
4. Develop a DD process, describing the sequence of activities, artifacts and roles involved, in a manner consistent with the DA process.
5. Validate the proposed DA and DD processes.

1.2.3 Overview

This work proposes a model for building domain analysis and domain design processes and their associated artifacts, starting from meshing tool domain knowledge.

The model deals with the main two phases in a SPL: Domain Engineering and Application Engineering. Within these phases, it focuses on the stages of Analysis and Design.

The model developed in this thesis is summarized in Figure 1.3.

The model provides two paths for creating the different artifacts produced by the associated process in the domain engineering and the application engineering phases, starting from the domain knowledge and customer needs. Both paths have as their main input the domain model produced during the domain analysis stage, which incorporates the common and variable abstractions elicited in this stage. Once the domain model is established, it is possible to build, during the domain design stage, a common architecture for every product in the SPL and then to instantiate a particular product architecture that takes into account the costumer's needs, during the application design stage. Alternatively, the proposed model allows instantiating a particular application's requirements in the application requirements stage, and then using it as the basis for deriving a particular product architecture in the application design stage.

The first path has the advantage of producing an explicit product line architecture which can then be instantiated for every required single product architecture, according to the user's needs. Furthermore, this common architecture can be assessed, making it unnecessary

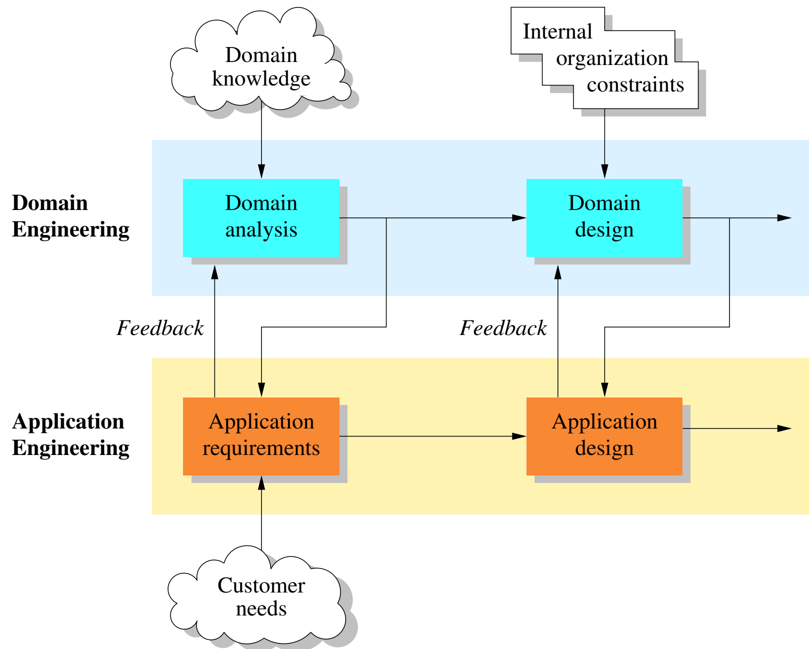


Figure 1.3: Domain and Application Engineering for Meshing Tool Domain

to assess each particular product architecture. Building this common architecture is no easy task, requiring a large effort and tradeoffs among stakeholders, but the benefits grow with the number of products to be designed.

On the other hand, the second path has the advantage that is easier to build the architecture for only one product. Each new architecture is built on demand only when it is required. However, it is necessary to assess every single architecture independently from other architectures. Moreover, the common architecture is implicit, and is constituted by the sum of all single architectures.

Regarding Sections 1.2.1 and 1.2.2, this thesis does not address the question of which path to follow. However, the choice of one or the other is guided by certain considerations. For example, the first path should be chosen if there is extensive knowledge about the domain, clear software quality attributes, a considerable quantity and variety of implemented components and relaxed development time constraints. The second path should be chosen otherwise.

Finally, it can be noted that the process has feedback, from application engineering to domain engineering. Customers' needs are always changing, and this feedback allows considering new organizational or business opportunities and thus the evolution of the SPL.

According to the SPL process presented by van der Linden [234], and in accordance to this thesis' hypothesis and objectives, there are two missing phases: domain implementation, where code generators and software components are built for future integration in particular products, and application implementation, where particular software products are built using artifacts developed during the domain implementation. Nevertheless, the models and processes are documented using rigorous notation, thus easily allowing the execution of the

subsequent stages.

This thesis does not intend to develop a totally new approach for developing SPLs. Instead, the idea is to take the best practices from some existing SPL approaches or their individual stages, e.g. domain analysis or domain design, and adapt them to the particular application domain of meshing tools.

In that sense, one source of ideas was the PuLSE methodology [22]. This methodology is applicable to any domain, encompasses several processes and subprocesses, and covers all aspects of the development of a SPL. As such, it generates several artifacts at each step of its processes, which are then necessary for other steps of the methodology. Furthermore, in order to accomplish all processes involved in PuLSE, a significant amount of domain information is necessary. However, these features of PuLSE do not impact positively the meshing tool stakeholders, mainly scientists as mathematicians, physicists, mining engineers, as they are not used to applying complex software engineering methodologies for software development [118]. Rather, they are usually more interested in quality attributes such as performance, correctness, and precision of the results. Nevertheless, lately they are usually willing to sacrifice these attributes to a certain degree in order to attain other desirable attributes such as maintainability or a shorter development time[6].

1.2.4 Justification

SPL is an approach for systematic and organized reuse within a certain application domain, and has been successfully applied in several different domains [157]. SPL's goals are to reuse as much as possible the same software assets, i.e., requirements, architecture, and components in all products of the line, thus decreasing development costs, increasing productivity, and achieving and maintaining a desired quality level of products [154]. Also, van Ommering and Bosch [240] highlight the need for using a SPL so as to work in a coordinated, proactive and systematic way with shared architecture, components and infrastructure for reuse.

In the meshing tool domain, other authors have made important efforts in the SPL direction [18, 81, 212, 213, 214], further justifying the adoption of a SPL approach as a solution.

The application domain also complies with the four drivers for SPL as exposed by van Ommering and Bosch [240]:

SPL drivers

Manage Size and Complexity: Meshing tools are sophisticated software because they manage large number of bidimensional or tridimensional elements and they are applied in different kinds of domains. Many different algorithms are necessary for producing, processing and visualizing meshes.

[6] Nikos Chrisochoides. Personal communication, January 2007. On Quality Attributes in the Meshing Tool Domain.

Obtain High Quality: Meshing tools need to deal with several quality attributes. Some of them are performance and maintainability with its variants, such as modifiability, extensibility, portability, and interoperability [18, 20].

Manage Diversity: Existing meshing tools are used in diverse application domains. These tools share several features, algorithms and interfaces which can be studied to extract commonalities and variabilities. A feature model can then be used for domain model formalization.

Lead-Time Reduction: In general, many meshing tools are developed from scratch. This in turn discards all knowledge acquired from previously built tools. The main solution to obtain lead-time reduction is software reuse [240].

Finally, Smith and Chen [212] and Smith et al. [214] show that meshing tools meet the three hypotheses for a SPL as proposed by Weiss and Lai [247]:

SPL hypotheses

Redevelopment Hypothesis: Even though, in the meshing tool domain, there are many commonalities present in requirements, architecture, design, implementation and documentation, several meshing tools have been developed without applying reuse techniques.

Oracle Hypothesis: The possibility of predicting the changes that are likely to be needed by a system over its lifetime is given by the possibility of building a model that supports that prediction. In the meshing tool domain, this is dealt with by the SPL scope definition and by the feature model, which are part of the domain model.

Organizational Hypothesis: Similarly to the previous hypothesis, the organizational hypothesis is supported by the possibility of producing a methodological approach that rules every activity in the SPL development. It is shown in part by the proposed processes for the domain analysis and domain design stages.

1.2.5 Methodology

The methodology consists of five main activities that are carried out in order to study and understand the meshing tool domain and the SPL approaches, create processes for both domain analysis and domain design stage, produce architectures with intermediate artifacts as SPL domain model and scope, and evaluate all the artifacts produced by both stages.

Some of these activities overlap in time. A summary of the sequence of activities is presented below.

Study of the SPL development: The first activity is the study of existing SPL development methods as well as the diverse internal processes, i.e., the process where the reusable core assets are developed (domain engineering) and the process where the

products are built (application engineering), with their respective input and output artifacts, considering initially a general vision of the SPL without particular internal process and artifacts, or any particular application domain. This activity provides an initial understanding of the potentiality of the SPL approaches and their difficulties for a successful adoption. Additionally, it is necessary to study individual stages of the SPL, such as domain analysis and domain design.

Study of the meshing tool domain: This activity consists of the study of existing literature, some meshing tools to get a good understanding of the domain, and conversations with meshing tool developers and users. The information gathered is used to build a domain model and to define the limits of the domain, so as to properly describe the features and the way that the applications work.

SPL model definition: The next activity consists of defining the SPL development model for the meshing tool domain. The model covers both the domain engineering and the application engineering phases. Activity diagrams are built for the process of domain analysis, domain design, and products instantiation through the application engineering phase. The SPL model definition is obtained through an iterative process, making it necessary to review the development model several times.

SPL model application: Once the SPL development model is built, it is applied to a complete domain analysis and domain design. Each step of each process is accurately and carefully followed, and the artifacts for each process are produced: domain model, scope, and product line architecture with a state machine for the behavioral view. The artifacts produced show how the model works, and some amendments are introduced to the model.

SPL model validation: Even though building the SPL development model is, in some sense, a validation of the chosen approach at a domain engineering level, this is complemented at the application engineering level by the instantiation of selected particular artifacts both at the application requirements and application design stages.

1.2.6 Results of the Thesis

The following results are achieved in this thesis:

- Meshing tool domain characterization.
- Domain analysis process.
- Domain model formal definition.
- Meshing tool product line architectures.
- Domain design process.
- Transformations for generating meshing tool product architectures.

1.3 Document Structure

This document is organized as follows.

- Chapter 2 shows what a software product line is, describing its stages, some SPL approaches, and their main advantages and disadvantages.
- Chapter 3 presents a description of the meshing tool domain.
- Chapter 4 presents the meshing tool domain analysis stage, describing its process graphically and its output model formally.
- Chapter 5 presents an application example of the domain analysis process described in the previous Chapter.
- Chapter 6 presents the meshing tool domain design stage, describing its process graphically, reviewing the related work, and establishing guidelines for building the product line architecture and transformations for automating product architecture creation.
- Chapter 7 presents the application of the domain design process to the same example shown in Chapter 5.
- Chapter 8 presents the conclusions of this thesis, as well as directions for future research work.

Chapter 2

Software Product Lines

This chapter presents an overview of software product lines, situating them in the context of software reuse. Its advantages and disadvantages are explored, as well as the differences between a software product line and single system development approach. Finally, its stages are presented and discussed in detail.

2.1 Context: Software Reuse

Even though there is no generally accepted definition of software reuse, Fortune et al. [84] give a couple of definitions about reuse in general, and define software reuse as:

Software reuse

> *"the use of systems artifacts and processes in the development of solutions to similar problems."*

Essentially, the purpose of software reuse is to reduce costs of software production and maintenance, improve software quality, reliability and productivity, and to reduce time to market [85, 217]. However, there are several costs and problems associated with software reuse, such as increased maintenance costs, lack of tool support, costs of component libraries creation and maintenance, among others [217].

According to de Almeida et al. [66], the software reuse process is a key factor in improving quality and productivity, because it creates software systems from existing software pieces, thereby reducing significantly software development effort, component duplication and results overlap in each phase of the software life cycle.

There are at least three issues to be addressed for software reuse to be successful [14]:

1. Creating items so they are reusable in the future (item characteristics).

2. Finding appropriate reusable items (environment characteristics).
3. Integrating them to build a new system (both item and environment characteristics).

These issues arise because it is necessary to develop said items according to standards, to know exactly which items are necessary, to develop the right items, to know the way or mechanism in which the items will be reused (e.g. use an instance of a general item, or modify the item to fulfill new requirements), and to give support for accessing, modifying and porting the items.

Another important aspect when developing software with reuse is the granularity of the units to be used. In this sense, three options should be considered [217]:

1. Application system reuse: The whole application system may be reused either by incorporating it without changes into other systems or by developing application families.
2. Component reuse: Components of an application, ranging from sub-systems to single objects, may be reused.
3. Object and function reuse: Software components that implement a single well defined object or function may be reused.

Considering these three options, probably the most effective reuse technique is the first one, because it implies the reuse of coarse-grained assets that can be quickly configured when building new systems. However, it is also the hardest to achieve, requiring a well-defined process and a thorough understanding of the domain.

Software reuse benefits

In relation to the benefits of software reuse, and considering all the aforementioned considerations, de Almeida [65] establishes that software reuse has a positive impact on software quality, as well as on cost and productivity, and summarizes those benefits as follows:

Improvements. Software reuse results in improvements in quality, productivity and reliability.

- Quality. Error fixes accumulate from reuse to reuse.
- Productivity. Productivity increases when less code has to be developed.
- Reliability. Using well-tested components increases the reliability of a software system.

Effort Reduction. Software reuse provides a reduction in redundant work and development time, which yields a shorter time to market.

- Redundant work and development time. Developing every system from scratch means the redundant development of many parts. This can be avoided when these parts are available as reusable assets and can be shared among systems.

- Time to market. Using reusable assets can result in a reduction of time to market.
- Documentation. Reusing software components reduces the amount of documentation to be written but increases the importance of what is written.
- Maintenance costs. Fewer defects and less maintainability of the system can be expected when proven quality components are used.
- Team size. If many components can be reused, then software systems can be developed by smaller teams, leading to better communication and increased productivity.

There are several different software engineering techniques aimed at reducing the development time and cost of software systems. However, there is no single successful development method that is applicable to all problems or domains. Even though improvements in software development have been considerable, existing generic solutions have reached a point where it is difficult to make them better. Thus, further improvements must make use of knowledge of the specific application domains' characteristics.

An idea used in several engineering areas is to base software development on components reuse. Clearly, for the success of this strategy, it is necessary to consider reuse in the early stages of development, that is, in the requirements engineering and/or software design processes. Furthermore, it is important to have reuse goals, such as the reuse of requirements, design, code or tests, among others. Several software development paradigms as structured programming, object oriented programming, and component based development have been used since the 1960s and are usually cited as examples of successful software reuse. However, they focus mainly on code reuse and thus do not contribute to reuse at other development stages [40].

Sommerville [217] lists several approaches that support software reuse and software product line (SPL) is one of them. Furthermore, SPL is a kind of application system reuse, as was pointed out above by [217]. In fact, SPL development has proven to be an effective way to benefit from reuse and variation, and it has allowed many organizations to reduce costs, time and increase quality [90].

2.2 Software Product Lines

Maybe Parnas was one of the first researchers to talk about program families, a concept that addresses variability in non-functional characteristics [236], which is similar to SPL. Parnas [168] defines program families as

Program family

> *"sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members."*

This definition puts explicit emphasis in one of the key aspects in the reuse approach, that is, the common characteristics shared among different products. In Parnas's view, these characteristics correspond to reusable components.

Another similar definition is given by Asikainen et al. [8]. For them, a software product family

Software product family

> *"consists of a common architecture, a set of reusable assets used in systematically producing, i.e., deploying, products belonging to the family, and the set of products thus produced."*

This definition indicates which are the constituent parts of the product family, and emphasizes the systematic way in which the products are built.

Probably the most generally accepted definition is the one stated by Northrop and Clements [157], which states that a SPL

Software product line

> *"is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."*

Some researchers establish a difference between software product lines and software families. In particular, a software product line is designed to satisfy a given market, whereas a software family can be the basis for several product lines aimed at different markets [60]. In this thesis, these terms will be considered synonymous, as these issues are not relevant to the scope of this work.

SPL example

When a SPL is developed, a collection of related products is developed from generic core assets and product-specific core assets. Figure 2.1, obtained from Gorton [90], shows an example of a simple product line. In the figure, two different calculators are developed using the same core asset internal boards. The different functionalities of the two calculators are provided by each of their product specific assets, that include the two different kinds of buttons that provide the individualized interface to the generic functionality.

The existence of commonalities among different products is necessary to create a common base infrastructure for all products in the SPL. At the same time, product variabilities are also necessary, as they allow having several products which differ in some aspects or characteristics. These variabilities must be such that they do not compromise the existence or development of the SPL and its reuse infrastructure by being more than the commonalities [69]. In other words, the commonalities must be larger than the variabilities, in the sense that their reuse in all products justifies the time and effort invested in their development.

There are several techniques for representing these commonalities and variabilities. Feature models [60, 115] are probably the most widely known and used technique. Two other

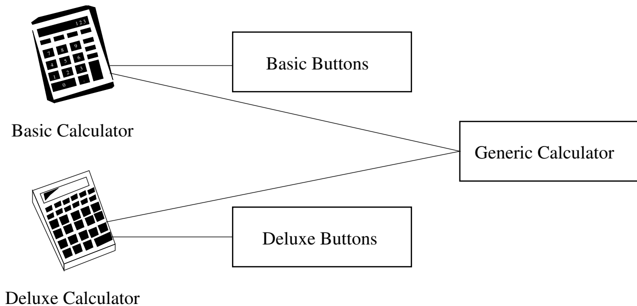


Figure 2.1: A view of a simple product line

popular techniques are FAST Commonality Analysis, proposed by Weiss [246], and the Orthogonal Variability Model, proposed by Pohl et al. [174]. In general, all these techniques are used as a specific method within the domain analysis process. This is explained further in Section 4.2.4.

A SPL development process consists typically of two phases: domain engineering and application engineering.

Domain engineering definition

The domain engineering phase is where the common and variable parts (core assets) are defined, developed and maintained. This phase is typically called the development process **for reuse**. Classically, it follows a waterfall life cycle [232]. Its inputs usually are preexisting components, documentation, products, and product constraints, among others, and its outputs are usually the domain model, the product line scope, the core assets and the production plan. On the other hand, the application engineering phase is where the applications of the line are built by reusing domain assets produced in the domain engineering phase. This phase is typically called the development process **with reuse**, and it also follows a waterfall life cycle.

Application engineering definition

Product commonalities and variabilities are generally defined in the domain engineering phase, and product specific parts are handled exclusively in the application engineering phase. This is shown in Figure 2.2, obtained from van der Linden et al. [236].

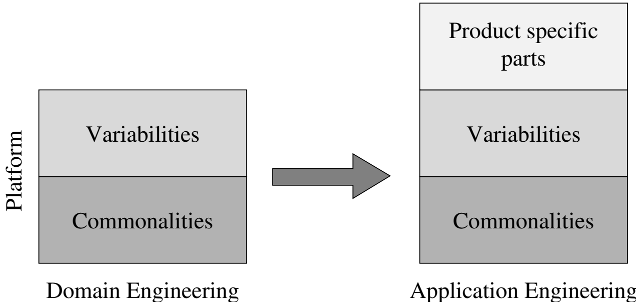


Figure 2.2: Variability handling

Svahnberg et al. [224] conducted a thorough research on variability, in which they emphasize the factors to be considered when selecting an appropriate method for implementing variability, and provide a taxonomy of techniques that can be used to implement variability.

SPL adoption

SPL adoption involves moving from developing software systems via a single system to developing a family of software systems. According to McGregor et al. [136], organizations typically have used two kinds of strategies for introducing SPL: a heavyweight strategy and a lightweight strategy. In the first one, an organization assigns specific teams of specialists to produce core assets, such as the product line architecture and components. In a lightweight strategy, an organization first creates some products (in a traditional manner) and then mines all products created so far to extract common characteristics (i.e. core assets). For Krueger [126], a lightweight strategy is more convenient because it is a lower-risk strategy that implies a small upfront investment, since the products already exist.

Additionally to the two strategies described above, other authors such as Gorton [90] and Schmid and Verlage [197] explain situations where it is possible to adopt SPL. For example, the first author cites two situations:

1. *Green Fields*: Where no products initially exist. In this case, it is important to initially define what products will be possible to build (the scope) and how they will be constructed (the production plan). Then, it will be easier to plan and evaluate several investment options.
2. *Ploughed Fields*: Where a collection of related legacy products have been developed without reuse in mind. In this case, it is highly probable that the organization already has a good understanding of the scope of the SPL, which will be driven by the existing products' functionality and future production plans.

Adoption of SPL development in an organization usually faces several problems, as people in charge of building it frequently underestimate the management commitment and involvement required to succeed. Additionally, organizations that are new at implementing SPL usually have inappropriate organizational structure and processes, lack of training, have insufficient and inappropriate resources, incompatible development processes, cultural resistance, and lack of carefully constructed plans for product line adoption [52, 158]. Furthermore, some organizations tend to jump right into product line architecture and component development activities without defining the scope of the SPL.

Catal [40] summarizes common barriers to the adoption of SPL process in an organization. The most relevant are:

- Lack of practical resources: There are several books and articles that explain the process of the SPL, but there are not many that combine both the process and practical techniques for developing SPL.
- The necessity of change: Organizational and process change are necessary for a successful product line initiative. This necessity for change prevents the senior management group from adopting the SPL approach quickly.
- Abstract statements in case studies: Most published SPL case studies include mostly abstract statements and it is very hard to adopt their methodology with the information given.

- Lack of SPL experts and high cost of training: There are not enough SPL experts and the cost of training is very high for developing countries. Moreover, a successful SPL initiative requires not only training but also extensive practice.

Finally, van der Linden et al. [236] present some principles to be considered for a successful SPL adoption.

- Variability management: individual systems should be considered as variations of a common theme. This variability is made explicit and must be systematically managed.
- Business-centric: SPL must aim at thoroughly connecting the engineering of the product line with the long-term strategy of the business.
- Architecture-centric: the technical side of the software must be developed in a way that allows taking advantage of similarities among the individual systems.
- Two-life-cycle approach: the individual systems must be developed based on a software platform. These products as well as the platform must be engineered and have their individual life-cycles.

2.3 Differences Between a SPL and Single System Development

There are several differences between a SPL and single system development. The obvious one is that SPL is geared to developing several software products instead of just one. However, other important differences stand out:

- SPL aims to develop reusable software explicitly. Indeed, it is possible to see a SPL as a generic system that must be configured or instantiated to produce concrete systems or components to be reused in different systems [60].
- SPL promotes scale economies, by taking economic advantage of the fact that many of its products are very similar [157].
- Time to market in a SPL is shorter than for single system development because new systems are built from core assets and only the functionalities that are unique to specific products must be developed [90].
- A defect in a core asset only needs to be fixed once. This in turn benefits every product in the SPL that uses this core asset [90].

For Pohl and Metzger [175], there are two essential differences. The first one is that SPL has two development processes: a domain engineering process, where the commonalities and the variability of the SPL are defined and the domain artifacts are built, and the application engineering process, where SPL applications are derived from the domain artifacts. The second one is the explicit definition and management of variability through variation points and variants: a variation point indicates and specifies what can vary, and a variant defines a concrete variation.

2.4 Advantages and Disadvantages of Using a SPL Approach

SPL advantages

The best known benefit of a SPL approach is a reduction in development costs. This is supported by several authors [6, 30, 107, 157, 197, 236], and is illustrated in Figure 2.3, obtained from Pohl et al. [174].

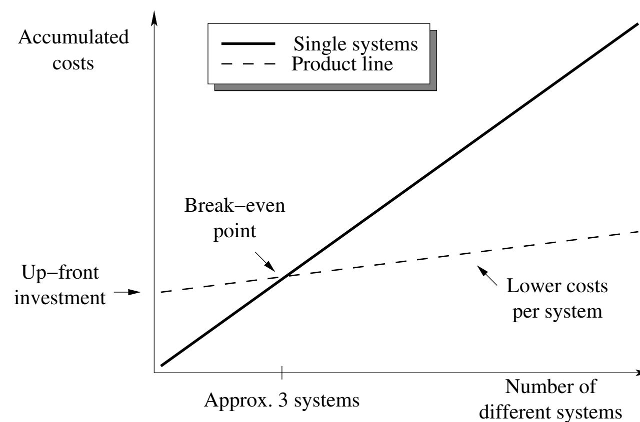


Figure 2.3: Costs Comparison between Single System and Product Line Development

The figure shows the accumulated costs needed to develop n different systems. The solid line shows the costs of developing the systems independently, and the dashed line the costs for SPL. When only a few systems are developed, the costs for SPL are high, due mainly to the high initial investments, such as the definition of the product line scope, the development of reusable core assets, and the creation of a production plan. SPL development costs per product decrease as the number of products increases, as core assets are reused in several different systems. The break-even point at which SPL is less expensive than single system development is generally understood to occur at approximately 3 systems.

For Gorton [90], the most obvious benefit is an increase in productivity, as the cost of developing and maintaining core assets is distributed across all products in the SPL. Thus, an organization can capture these economies of scale to benefit from the development of a large number of products.

Time to market

Another commonly identified benefit is a reduction of time to market. Time to market is

simply the time it takes to get a product from an idea to the marketplace. For Kasunic [117], it is the length of a project in work days, excluding times when the project is not active due to work stoppages. The time starts when user requirements have been baselined, and ends when the first software installation is done.

Figure 2.4 obtained from Pohl et al. [174] shows the time to market for both types of developments.

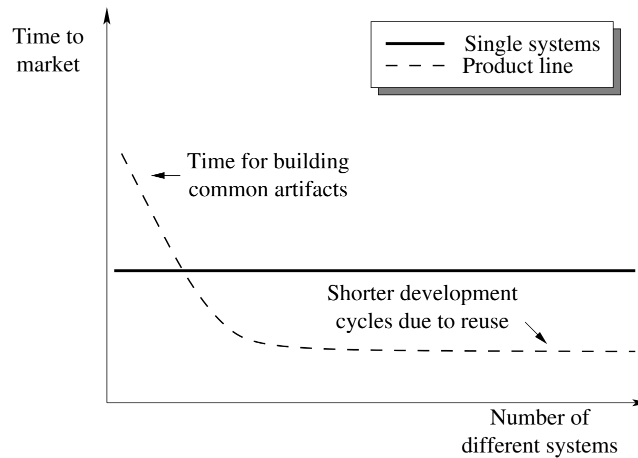


Figure 2.4: Time to Market comparison between Single System and Product Line Development

Time to market for single system development is considered constant. Time to market for SPL is initially higher because of core assets development. Once they are built, the time to market for each new system produced by the SPL is reduced as core assets are reused. Thus, only those parts of new products that do not exist as core assets must be developed, which need probably to be tailor-made. The customers only need to wait for the development of those functionalities that are unique to their needs [90, 174].

According to Northrop and Clements [157] and Trigaux and Heymans [232], organizational competitive advantages are also obtained from core assets reuse. For instance, up to 100% of the components in the core asset base are used in each product, so fewer people are required to build products. Also, focusing personnel training on the core assets can lead to a better domain comprehension, and to shorter training time.

Pohl et al. [174] and van der Linden et al. [236] present three other benefits:

- Reduction of maintenance costs: Whenever a core asset is changed, its changes can be propagated to all products in which the core asset is being used. Moreover, the overall amount of code and documentation to be maintained is reduced.
- Quality improvement: Core assets are used, reviewed and tested in many products of the SPL. Therefore, there are more opportunities for fault detection and correction. Also, a new software product has a large number of proven components. The defect density of such products can be expected to be lower than products that are developed from scratch [90].

- Usability improvements in the final products: User interface consistency can be improved by using the same building blocks when implementing the same kind of interactions, thus making it easy for the customer to switch from one SPL product to another.

SPL disadvantages

The main disadvantages of the SPL approach are related to the costs of developing the core assets. Additional costs associated with the requirements core asset may include the costs of capturing the requirements for several systems, as this may require sophisticated analysis and intense negotiation among the stakeholders so as to agree on both common requirements and variation points that are acceptable for all products [157]. Likewise, the architecture core asset may include additional costs because it must support the variations inherent in the product line, which imposes additional constraints on it.

Trigaux and Heymans [232] mention at least four disadvantages of the SPL approach:

- Resistance to changes: It is not easy to change an organization from a single product development approach to a SPL approach. Many people need to learn the new approach, and change their way of thinking or confronting the new processes. In [54], Cohen shows that around 40% of the organizations in his study present some kind of resistance to changes.

- The need for a global view of the complete product line architecture: few software engineers know the application domain in depth. With this scenario it is difficult to generate a product line architecture. It is a problem because SPL are generally built around a product line architecture, and some organizations use the architecture as a mechanism for organizing software development, of both assets and products [54]. The problem increases due to the lack of guidelines, techniques and tools to represent or validate product line architectures.

- Problems with the scope definition: The scope must be carefully defined, as the domain can change. A too narrow scope does not justify the cost of core assets development and maintenance; likewise, it is almost impossible to reuse core assets if the scope is too broadly defined. Scope definition will be described in more detail in Section 4.1.2.

- Benefits are not visible immediately: SPL development needs a high initial investment. Thus, it is necessary to develop several products to obtain a return on investment (ROI). This was shown in Figure 2.3; SPL needs to build three or more products to appreciate its real advantages.

Another important disadvantage of SPL is SPL deterioration over time due to the SPL evolution. This deterioration is commonly called *erosion*, i.e., the deviation from product line models up to the point where key properties no longer hold [71]. According to Thao [229], the evolution in SPL may cause problems in the time dimension (referring to the evolution of shared code and the product itself over time) and/or in the space dimension (referring to the variability among the products in the product line). SPL evolution must also deal with

SPL evolution

diverse issues such as customer's needs and technology, which can be seen as functionality addition or change [130]. For Dhungana et al. [72], there are two important issues in SPL evolution: maintenance is done by multiple teams, where the knowledge about variability is distributed across multiple stakeholders and teams, and the fact that the evolution of different parts of a product happens at different speeds.

Thus, for a correct SPL evolution, the SPL maintainer has to perform an accurate previously specified sequence of steps [143], not necessarily in a standard form in any SPL domain. In any case, SPL evolution has been recognized as a research challenge [143, 174].

Finally, there are several approaches for measuring the possible economic benefits of using a SPL, as studied by Ali et al. [1] and Böckle et al. [30]. The evaluation of economic benefits of SPL is beyond the scope of this thesis.

2.5 SPL Stages

During the years, researchers have proposed different approaches for SPL development, with different numbers of phases. For instance, Northrop and Clements [157] identify three phases or activities: Core Asset Development, Product Development and Management. Figure 2.5, obtained from the authors above, presents these phases.

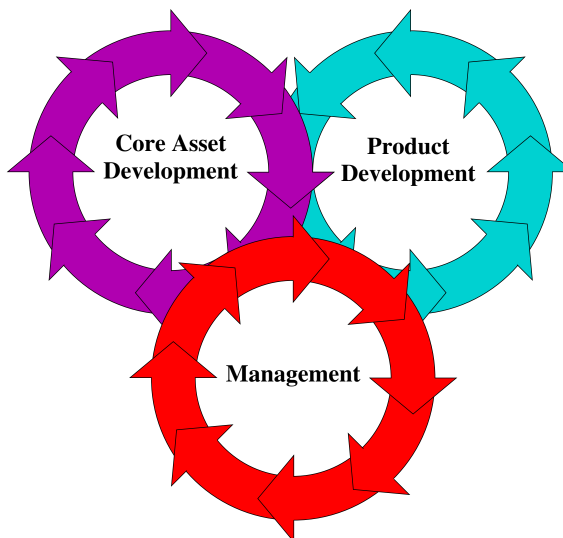


Figure 2.5: Activities for SPL

Figure 2.5 shows these three strongly linked and iterative phases. They can occur in any order, and there is a clear and strong feedback loop between the core assets and the products. In that sense, core assets must be updated when new products are developed. After several cycles, more generic core assets can be obtained, which in turn contribute to the development of new products probably not initially considered. Each phase, in turn, has several more detailed stages inside, which can deliver products, too. Additionally, there is a

permanent need and worry for management to invest time and resources in the development and update of the core assets.

Other authors such as Pohl et al. [174] and Czarnecki and Eisenecker [60] identify two phases in SPL development. In particular, Pohl et al. refers to the core asset development phase as the Domain Engineering phase, and to the product development phase as the Application Engineering phase, and they include the management phase inside the Domain Engineering phase.

As stated by Czarnecki and Eisenecker [60] and Pohl et al. [174], the application engineering phase is a process based on the results of the domain engineering phase, in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability. In this sense, it is important that the investments necessary to develop the core assets during the domain engineering phase are outweighed by the benefits of deriving individual products in the application engineering phase [70]. It is not easy to reach this goal. An alternative mentioned by Deelstra et al. is to invest in research and technology in the SPL domain engineering phase, in such a way that these investments decrease costs in the application engineering phase.

Figure 2.6, obtained from Deelstra et al. [70], shows the impact of the application of technology in the domain engineering and application engineering phases.

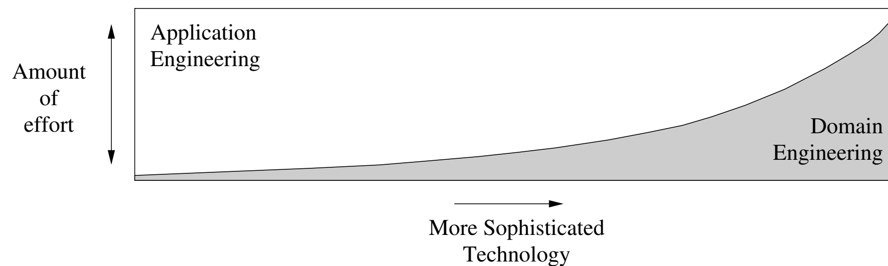


Figure 2.6: Impact of technology on the SPL phases

If little effort is invested in the domain engineering phase (left side of Figure 2.6), i.e., the standardized reuse infrastructure provides generic behavior, contains no domain specific functionality, and product variability is managed as in single system development, then a large effort will be invested in the application engineering phase. On the other hand, large investments in the domain engineering phase (right side of Figure 2.6), i.e., the reuse infrastructure captures all commonalities and variabilities and the product line architecture is well-defined with no products derived out of architecture, then a large investment in the application engineering phase is not as necessary.

This thesis considers domain engineering and application engineering as the two essential phases in the SPL development. These phases, in turn, have other sub-phases or stages, which in turn also may produce outputs or products. There is no agreement on the name or number of these stages. For instance, Czarnecki and Eisenecker [60] and van der Linden [234] present three stages in the domain engineering phase and three stages in the application engineering phase, but they differ in the names of the stages. On the other hand, Pohl et al.

[174] present domain engineering and application engineering phases with four stages inside each one. These differences and similarities are presented in the following paragraphs.

Standard domain and application engineering phases are presented by Czarnecki and Eisenecker [60] by means of Figure 2.7.

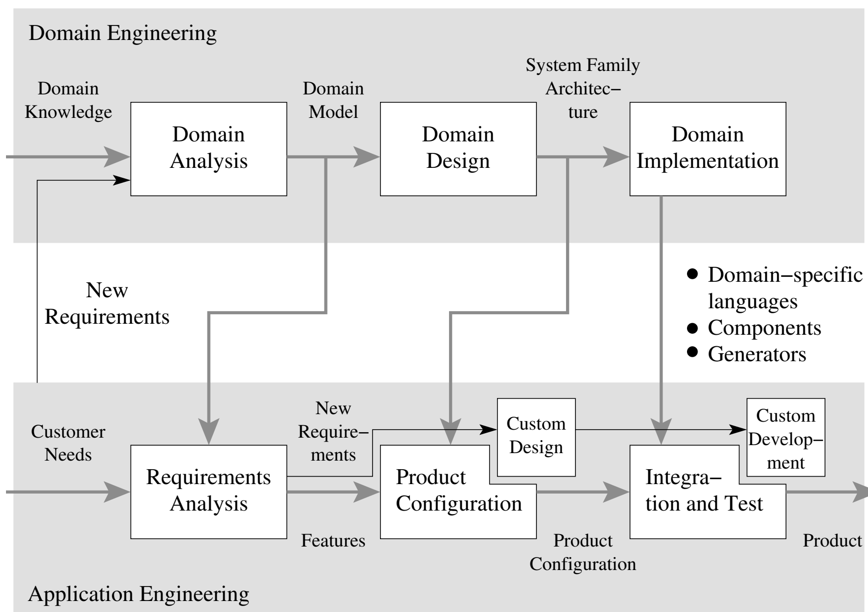


Figure 2.7: Software Development based on Domain Engineering

This figure shows the two phases working in parallel. Domain knowledge is an input for domain analysis and represents any kind of knowledge, such as stakeholders knowledge, and information about existing systems and components. The domain engineering phase only produces reusable core assets, and it does not produce any product or application. That task is supported by the application engineering phase, which receives the customers' needs for a specific product and constructs it.

Figure 2.7 also shows it is possible to incorporate to the SPL new products not initially considered. Initially, the product line is able to produce certain preestablished products. If some customers' needs are out of the scope of the SPL, new requirements are specified and then it is necessary to evaluate whether it is worth incorporating these needs in the form of core assets to the reuse infrastructure, or whether it is enough to develop these new requirements and translate them into a custom design and custom implementation.

Van der Linden [234] shows a similar SPL process in his article, which is illustrated in Figure 2.8.

Similarly to Figure 2.7, there are three stages in each phase. However, there are some important differences between them:

- Feedback: Feedback in Figure 2.7 is explicit and produced from unsatisfied customers' needs in the application engineering phase to the domain engineering phase. Feedback

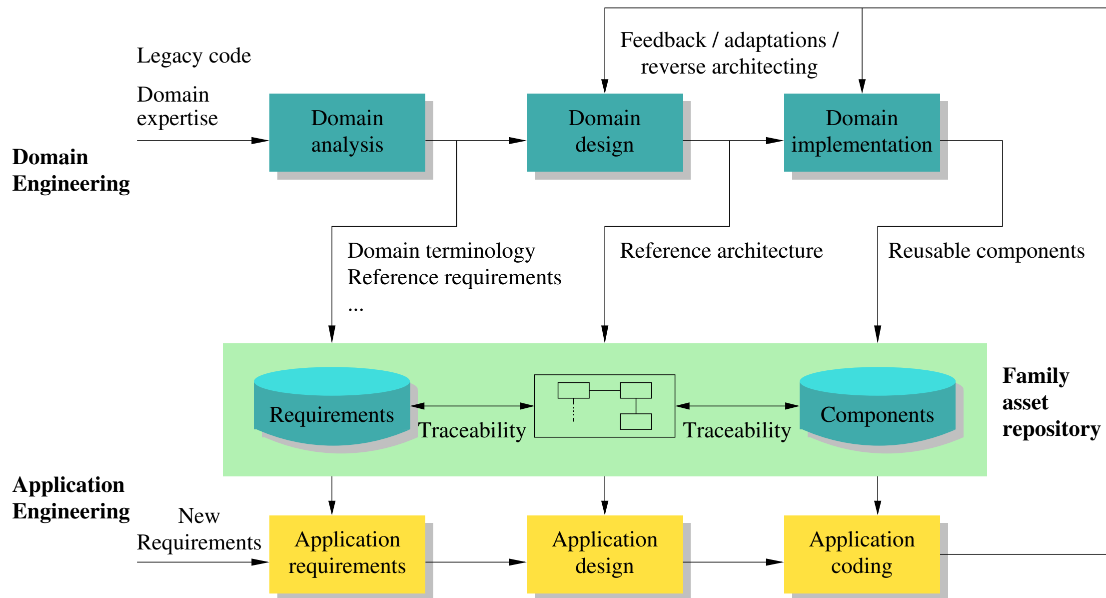


Figure 2.8: The Praise Reference Process

enters the domain analysis stage, and changes are propagated to the domain design and domain implementation stages. In Figure 2.8, feedback is produced from the application coding stage to the domain design and domain implementation stages. From the nature of the process involved, it is necessary to consider both views of feedback as complementary, because issues or needs for change can occur at any stage of the application engineering phase, and their solutions can involve several stages in the domain engineering phase.

- Repository: Figure 2.8 shows a family asset repository between the domain engineering and the application engineering phases, storing all core assets produced by the domain engineering phase. These assets can later be used by the application engineering phase according to its needs. On the other hand, Figure 2.7 does not show the existence of a repository explicitly. Instead, assets such as domain specific languages, components and generators are mentioned between the domain implementation and integration and test stages. All assets are important, not only those produced by the domain implementation stage. When all core assets are stored in a repository, there is a greater chance of improving software production, and software and SPL maintenance.

- Traceability: Traceability aids the development process by helping determine which assets are necessary for which set of product requirements, and the maintenance process by helping determine which versions of which assets are used in which systems. It is not mentioned explicitly in Figure 2.7, but vertical traceability and horizontal traceability are assumed. In contrast, traceability is an important issue in the Praise Reference Process described in Figure 2.8, which explicitly shows only the horizontal traceability.

These three elements are important in any SPL approach. Particularly, the feedback between stages is shown explicitly in Section A.2.7, according to Figure 1.3 in Section 1.2.

The six stages of the SPL development approach are described in the following subsections.

2.5.1 Domain Analysis

Domain analysis is the process by which information used to develop software systems is identified, captured, and organized, with the purpose of making it reusable for creating new systems [178]. According to Czarnecki and Eisenecker [60] and Pohl et al. [174] the main goals of domain analysis are:

Domain analysis purposes

- define the domain scope;
- define a set of reusable, configurable, common and variable requirements for the systems in the domain;
- develop precise documentation of the requirements.

The input to the domain analysis stage is generically represented by the domain knowledge, which in turn has diverse sources such as existing systems, domain experts, manuals, systems, prototypes, customers, known requirements of future systems, among others. The output is a domain model, consisting mainly of a domain definition or scope, a domain lexicon and a feature model. This domain model will be the input for the domain design and application requirements stages.

Northrop and Clements [157] define scope as a description of the products that will constitute the product line or that the product line is capable of including. Clearly, determining the scope is a process that requires the participation of a wide range of stakeholders as developers, customers, and users, among others.

The domain lexicon or domain dictionary [226] represents the identification and definition of terms used in the domain model. This domain vocabulary must be used consistently by all stakeholders in the domain to allow a good understanding of the domain and to avoid misunderstandings, inconsistencies and ambiguities in its usage.

The feature model is a tree-like structure that contains a series of constraints among the included features. A feature is a distinguishable characteristic of a concept that is relevant to some stakeholders [60].

For Pohl et al. [174] domain analysis is composed of three steps:

Domain analysis steps

- Commonality analysis: Its goal is to identify which requirements are common to all SPL applications.
- Variability analysis: Its goal is to identify which requirements differ among the applications, and to determine these differences precisely.
- Variability modeling: This activity is concerned with modeling variation points, variants, and their relationships.

As the domain analysis process is the first stage, obviously any mistake at this stage is propagated to all other stages of the SPL development process. In particular, the correct definition of a suitable scope is crucial because a well-defined scope will determine the economical viability of an organization, by allowing it to generate products as required by the market.

2.5.2 Domain Design

According to Czarnecki and Eisenecker [60] the main goals of the domain design stage are:

Domain design purposes

- develop an architecture for the SPL,
- create a production plan.

The inputs to this stage are provided by the domain analysis stage and are the scope and the domain model and the stage's outputs are the architecture and the production plan.

The architecture of the SPL is the artifact that defines the overall decomposition of the products into the main components, capturing the commonalities between products and facilitating the variability description [31]. This architecture, called product line architecture or reference architecture, determines the structure and the texture of the SPL applications. The structure determines the static and dynamic decompositions that are possible and valid for all applications of the product line. The texture is the collection of common rules guiding the design and realization of the parts, and how they are combined to build applications [174].

It is known that SPL developments are architecture-centered [164], but Gorton [90] indicates that the existence of a product line architecture is not always necessary for SPL development success. Gorton states that, even though products in a SPL will have some type of architecture, it is not necessary to have a product line architecture, and presents mechanisms for achieving reuse and variation without a product line architecture. Extending the idea of software development without an explicit product line architecture, Perovich et al. [169] and Rossel et al. [189] present a transformational approach for deriving particular architectures. Section 6.2.2 will explore this aspect in more detail.

The SPL production plan is a description of how core assets are used to develop a product in a product line [41]. This plan is created to ensure that the products will be built using the core assets correctly and appropriately.

2.5.3 Domain Implementation

The main goal of the domain implementation stage is to implement reusable core assets [174].

Domain implementation purpose

It does not produce or develop concrete software applications. However, in the feature-oriented programming paradigm (also known as feature-oriented software development) [3, 21, 176], it is common to find different levels of granularity in core assets. In that sense, UML documents, process diagrams, makefiles, object-oriented classes, modules, and concrete applications can be considered as core assets. Thus, this stage considers that a concrete software application has been developed first. In any case, this paradigm is out of the scope of this thesis.

The inputs to this stage are provided by the domain design stage. The outputs are several core assets, such as components, interfaces, code and GUI generators, domain specific languages, database tables, protocols, documentation, among others [60, 174].

Particularly, Pohl et al. [174] establish explicitly that interface design and component design are part of the domain implementation stage. Furthermore, both interface and component implementation, and compilation are carried out at this stage. The domain implementation stage does not produce applications or final products for customers; instead, it provides an reuse infrastructure for the application production process.

According to de Almeida et al. [67], there has not been a lot of effort addressing this stage, in contrast to the domain analysis and domain design stages. These authors present a domain implementation approach based on the Open Service Gateway Interface (OSGi). OSGi is a Java-based interface specification that defines a standardized, component-oriented, computing environment for networked services. Basically, the work presented by de Almeida et al. [67] provides a systematic way for implementing components for reuse within a SPL through a set of principles that software engineers can keep in mind while performing activities to implement and document software components.

2.5.4 Application Requirements

Application requirements purposes

The main goal of this stage is to elicit and to document the requirements for a particular application and at the same time reuse, as much as possible, the domain analysis artifacts [174]. In words of van der Linden [234], application requirements determines what the product should be.

The inputs for this stage are provided by the domain analysis stage: the scope and the domain model. Moreover, it is possible to have other inputs as specific requirements for a particular application that have not been captured during the domain analysis stage. The output of this stage is the requirements specification for a given particular application.

Some activities to be carried out in this stage as stated by Pohl et al. [174] are:

- Communicating the commonality and variability of the product line: It is important that the stakeholders know the SPL capabilities. With this information, the stakeholders can decide which variants to select for each variation point.

- Evaluating differences between domain and application requirements: Sometimes, stakeholders requirements for a given application cannot be completely satisfied by the domain model. These requirements and their impact on the SPL must be evaluated according to determine the effort required to build them. The decision to implement these new requirements is taken by the stakeholders.

- Documentation of application requirements: This documentation is a result of the previous activities. The documentation includes the new requirements, both those present in the commonality and variability described in the domain analysis stage and also those unsatisfied requirements that were accepted by the stakeholders.

2.5.5 Application Design

The main goal of this stage is to produce the application architecture [174]. For van der Linden [234], the application design stage selects the components needed to make the product.

Application design purposes

The inputs to this stage are the product line architecture, which is provided by the domain design stage, and the application requirements specification, which is provided by the application requirements stage. The output of this stage is the application architecture for a particular product.

The application architecture is not only produced from the product line architecture but also by taking into account the aspects related to application specific requirements that are not usually considered by the domain design.

2.5.6 Application Coding

The main goal of this stage is to develop applications that can be used by the customers in the domain [174]. According to van der Linden [234], the application coding stage combines components using the infrastructure and possibly additional product-specific code.

Application coding purposes

The inputs to this stage consist of the application architecture generated by the applications design stage and the reusable output artifacts from the reusable platform generated by the domain implementation stage. The output consists of a completely operational application.

This stage uses artifacts produced in the domain implementation stage, such as code and GUIs generators and domain specific languages. They allow generating applications in an automatic or semiautomatic way, depending on how many custom developments are necessary.

2.6 Domain-Specific Languages

Because of the complexity of the applications that must be developed to satisfy the needs of the users and customers, some authors [33] argue that the next level of abstraction that programming languages must conform to is that level most familiar to domain experts and users, i.e., based on abstractions and notations known to them. This abstraction level can be reached via domain-specific languages.

DSL definition

For van Deursen et al. [239], a domain-specific language (DSL) is a programming language or a specific executable language that provides an expressive power focused on, and usually restricted to a particular domain. For Bryant et al. [33] a DSL is a language tailored to a specific application domain, that offers a greater expressiveness and ease of use compared with general-purpose languages in its domain of application.

Normally, DSLs are small languages, more declarative than imperative, and more attractive than general-purpose languages when applied to the same application domain in order to achieve [96]:

- easier program understanding, writing, and reasoning,
- enhanced productivity, reliability, reusability, maintainability,
- easier verification,
- reduced semantic distance between the problem and the program.

DSL examples

The idea of working with specific-purpose languages instead of general-purpose languages is not new: through the years, several DSLs have been developed such as Cobol for business computing, Fortran for scientific computing [220], LaTeX for typesetting, HTML for hypertext web pages, the Backus-Naur Form (BNF) for syntax specification, SQL for database queries and manipulation [222], Excel for spreadsheets, Make for software building, MATLAB for technical computing, and VHDL for hardware design [142].

The DSLs previously mentioned and many others more can be grouped by their application domain, which in turn can be divided in five main groups [239]:

- Software Engineering: Financial products, software architectures, and databases.
- Systems Software: Description and analysis of abstract syntax trees, video device driver specifications, cache coherence protocols, data structures in C, and operating system specialization.
- Multi-Media: Web computing, image manipulation, 3D animation, and drawing.
- Telecommunications: String and tree languages for model checking, communication protocols, and telecommunication switches.

- Miscellaneous: Simulation, mobile agents, robot control, solving partial differential equations, and digital hardware design.

For example, in the software engineering domain, Medvidovic and Taylor [140] present several architecture description languages (ADLs) which are particular DSLs for describing the software architecture of a software system, i.e., the domain of software architecture. Thus, and according to the same authors, an ADL is focused on the high-level structure of the overall software system rather than the implementation details of any specific source module of that system.

It is known that DSLs can reduce dramatically the time and cost of software development and software maintenance, increase the software productivity, and assist programmers and users to write more concise and descriptive programs [33, 96]. However, software maintenance costs can increase or the system developed with the DSL can be difficult to maintain, if the underlying domain model assumed by the DSL changes [237].

The DSM Forum [80] justifies the use of domain-specific modeling techniques due to the productivity increase. Industrial experience has shown that domain-specific modeling is between 5 and 10 times more productive than current software development practices. It is mainly because the working programming abstraction level has been raised. For example, one sentence in Java or C++ corresponds to a much larger set of instructions in Assembler language.

According to Cánovas and Cabot [38], any DSL is composed of three main elements: an abstract syntax, a concrete syntax, and a semantics.

DSL elements

1. Abstract syntax: It defines the main concepts of the language, relationships, and constraints.
2. Concrete syntax: It defines the language notation, and it can be textual, graphical or a mix of both.
3. Semantics: It provides the meaning of each expression, and that meaning must be an element in some well-defined and well-understood domain [95].

When the idea to create a DSL for a particular domain arises, it is necessary to take into account if the problem to be solved or the domain itself deserves the creation of a DSL. In this sense, Sprinkle et al. [220] give a checklist for determining if a problem or domain requires the use or creation of a DSL. Some elements of the checklist are:

- The domain is well defined.
- The domain has repetitive elements or patterns, such as multiple products, features, or targets.

- An intuitive or well-accepted representation is already defined.
- The implementation or specification must serve as documentation.
- Amortization of effort justifies investment in DSL creation.

Moreover, the same authors indicate several considerations related to the use of DSL. Some of them are:

- The effort to create and maintain the language and related code generators.
- The effort for domain experts to learn the language.
- The productivity increase compared to general purpose solutions.
- The quality improvement compared to general purpose solutions.
- The number of expected users or implementations.

With all these things in mind, the task of building a DSL seems difficult and error-prone. To deal with this task, some authors such as Cánovas and Cabot [38], Mernik et al. [142], and Strembeck and Zdun [222] present, with varying knowledge depth levels, processes for building DSLs. In particular, a DSL development process is composed of seven phases: decision, analysis, design, implementation, testing, deployment, maintenance [142, 242, 244]. In the decision phase, the decision whether or not to build a DSL is made considering the needs of the domain; in the analysis phase, the application domain is analyzed and defined, collecting domain information and integrating it into a suitable domain model; in the design phase, both the architecture and language are designed, considering its syntax and semantics; in the implementation phase, the DSL and supporting run-time system are constructed, selecting from different approaches for the DSL development such as interpreter, compiler/application generator, preprocessing, extensible compiler/interpreter, Commercial Off-The-Shelf (COTS), and hybrid, among others; in the testing phase, the DSL evaluation is done, by making sure everything is in the working order; in the deployment phase the DSL and applications constructed with it are used; finally, in the maintenance phase the DSL is updated to reflect new requirements.

DSL development process phases

It is important to know that DSL development is not necessarily a sequential process. In this sense, sometimes phases overlap [242].

Similar to other types of software development, the first stages or phases are very important. Special consideration must be taken with the analysis phase of the DSL development. In the analysis phase, it is necessary to obtain problem domain knowledge. To reach this goal, a domain analysis is required. There are several suitable domain analysis techniques for the DSL development context such as FORM [116] and FAST [246]. Mernik et al. [142] and Čeh et al. [242] provide a list of other domain analysis techniques used in DSL development.

In particular, the use of feature models as a model that captures the commonalities, variabilities and the relationships among them is a good option for the analysis phase of the DSL development [142, 238, 242], although other authors think that feature models work at a level that is too general to properly identify necessary concepts involved in a DSL [33, 133].

In any case, either creating a DSL or adapting one, the adoption of the DSL approach for the production of applications has advantages and disadvantages. The advantages can be pointed as follows [33, 91, 237, 239]:

DSL advantages

- DSLs allow the solutions to be represented in the language and abstraction level of the problem.
- DSLs increase productivity, reliability, maintainability, reusability, quality and portability of the produced applications.
- DSLs assist programmers and end-users in writing more concise, descriptive, and platform-independent applications.
- DSLs contain domain knowledge, and this characteristic permits the preservation and reuse of this knowledge.
- Changes made to the applications are easy because they are developed automatically using the DSL and the intention of the applications is close to the domain.
- The tedious activity of writing long programs is automated.
- Domain-specific knowledge is explicitly available, and it is not hidden in the source code of the applications.
- The domain expert can understand, validate and modify the applications by adapting the domain-specific descriptions.

On the other hand, the disadvantages are [33, 237, 239]:

DSL disadvantages

- The cost of designing, implementing, and maintaining a DSL, and the cost of teaching its users are high.
- Developing integrated development environments (IDEs) for DSLs from scratch is too costly.
- The difficulty of finding the suitable scope for a DSL.
- The difficulty of striking a balance between domain-specificity and general-purpose programming language constructs.
- The potential loss of efficiency when compared with hand-coded software.
- An organization that is active in several domains may need a large number of DSLs, increasing the cost of development and maintenance.

Chapter 3

Meshing Tool Domain

Understanding the domain is a key factor in successful software product development. In particular, the meshing tool domain is very complex, involving concepts such as meshes, refinement and derefinement algorithms, mesh quality criteria, and mesh formats, among others. In this chapter, several meshing tool concepts are reviewed and defined.

3.1 Mesh Generation

Mesh generation algorithms play an important role in finite element methods (FEM), biomechanics, virtual surgery and geometric modeling [248]. Particularly, triangulations (product of triangular mesh generation) are widely used as a basic methodology in different fields such as finite element methods, computer graphics, geometric modeling, geographical information systems, terrain modeling and real time rendering [185].

The International Society of Grid Generation establishes that mesh (grid) generation is

Mesh generation

> *"that discipline of applied science and engineering which is devoted to the discretization of fields associated with the 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."*

This definition puts emphasis in the variety of domains and disciplines where mesh generation is applied.

Mesh generation is an essential prerequisite for the numerical analysis of engineering problems or fields, including those related to physical models described by partial differential equations (PDE) [88].

A PDE is an equation involving one or more partial derivatives of a dependent variable u [27] and independent variables x_i, with $u = u(x_1, x_2, ..., x_n)$. The general form of a PDE for u may be expressed as:

$$F(x_1, ..., x_n;\ u, u_{x_1}, ..., u_{x_n}, u_{x_1x_1}, u_{x_1x_2}, ..., u_{x_1x_n}, u_{x_2x_1}, u_{x_2x_2}, ..., u_{x_1x_1x_1}, ...) = 0$$

where

$$u_{x_i} = \frac{\partial u}{\partial x_i}, \quad u_{x_ix_j} = \frac{\partial u^2}{\partial x_i \partial x_j}, \quad u_{x_ix_jx_k} = \frac{\partial u^3}{\partial x_i \partial x_j \partial x_k}, \quad ...$$

Solving a PDE over a continuous domain requires the following five steps [145]:

1. Geometric modeling: The continuous domain is approximated using a simple, discrete description.
2. Mesh generation: The interior of the domain is decomposed into a mesh M of simple and well-shaped elements.
3. Approximation: A system of linear or nonlinear equations is formed over M for the governing PDEs.
4. Solution: The system of equations is solved, and the error of the solution is estimated.
5. Adaptive refinement: If necessary, the mesh is refined and steps 4 and 5 are repeated over the refined mesh.

In step 2, a mesh is generated as an intermediate step of a numerical method to compute or simulate physical quantities over the original domain. Not all meshes perform equally well in the subsequent numerical computation, and therefore, numerical and discretization errors depend on the geometric shape and size of the mesh elements [145]. Sometimes, mesh generation can be seen as a bottleneck for a numerical process (e.g. PDE resolution) in the sense that a failure in the mesh generation jeopardizes any subsequent numerical simulation [88].

This thesis is centered neither in the resolution of PDEs, nor in the development of particular algorithms for mesh generation or processing. A more detailed explanation of the main PDE solution methods, such as the finite difference method and the finite element method, can be found in Bernatz [27]. Furthermore, an introductory review to mesh generation can be found in Frey and George [88].

3.2 Mesh Generation Process

A mesh is a discretization of a domain [144], and is generally composed of a set of many small adjacent elements, typically triangles or quadrilaterals in two dimensions and tetrahedra or hexahedra in three [25].

Mesh definition

Figure 3.1 shows different kinds of mesh elements.

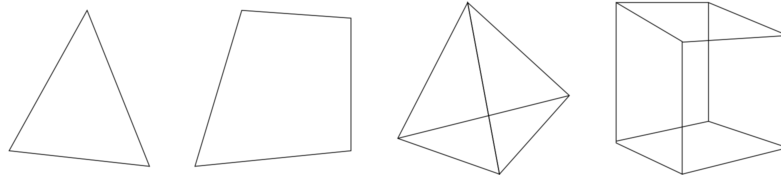


Figure 3.1: Triangle, Quadrilateral, Tetrahedron and Hexahedron

A meshing tool is a highly complex piece of software for generating and managing meshes [188]. Such tools are inherently sophisticated software due to the complexity of the concepts involved, the large number of interacting elements they manage, and the application domains where they are used.

Meshing tool

According to Douglass et al. [78], the mesh generation process has three main steps: geometry representation, discretization and mesh modification.

Mesh generation steps

1. Geometry representation: Mesh generation begins with a faithful representation of the geometry of the physical problem being modeled. There are three ways to represent the geometry:

   (a) Constructive solid geometry (CSG): It defines a domain using a formula of set theory operations over a collection of primitive geometric shapes (spheres, boxes, cones, etc.).

   (b) Boundary representation (b-reps): It defines the geometry in terms of piecewise low-degree polygons fitted to its boundaries, such as a nonuniform rational B-spline (NURBS) representation as from a CAD or CSG model.

   (c) Domain decomposition representation (dd-reps): It describes the geometry of the boundaries by a discretization into nonoverlapping polyhedra, from CAD or CSG models.

   Any flaws in the geometry must be repaired prior to mesh generation, a process called geometric healing.

2. Discretization: It is the decomposition of the physical domain into linear or low-order polynomial volumes such as triangles, squares, tetrahedra, hexagons, among others. The result of a discretization is a faithful representation of the actual physical domain.

The type of discretization used (the sort of element shapes and sizes used) depends on the material properties within and across material boundaries, whether the physical processes being simulated have preferred directions, whether the problem must be decomposed for parallel processing, among others.

3. Grid modification: It involves postgeneration modifications to the mesh to improve its quality. Quality modifications to the mesh may be a one-time event done prior to submitting the mesh to the equation solver, or may be a dynamic process either done explicitly for each step in the solver or implicitly linked to the physical variables to be solved for, all as one large problem.

3.3 Types of Meshes

The meshes are categorized in two types [26, 78, 145, 228]: structured and unstructured. This thesis is focused on unstructured meshes.

Structured and unstructured meshes

For Miller et al. [144] the difference between structured and unstructured meshes lies in the size of the elements: in the former, elements are squares of equal size; in the latter, there are no constraints on the size of the elements.

A structured mesh is one in which all elements are geometrically alike [228]. In two dimensions, in many cases it is simply a square mesh deformed by some coordinate transformation, where each vertex of the mesh, except at the boundaries, has an isomorphic local neighborhood [25]. An unstructured mesh has a varying local topology [228], and is frequently a triangulation with arbitrarily varying local neighborhoods [25].

Figure 3.2, obtained from Bern and Plassmann [26], shows a structured mesh and an unstructured mesh side by side.

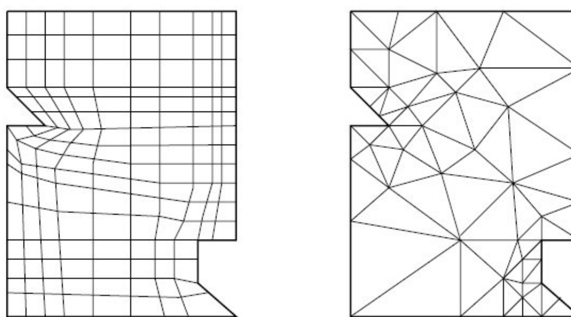


Figure 3.2: Structured and Unstructured Mesh

Owen [165] established that all interior nodes (vertices) of a structured mesh have an equal number of adjacent elements, while an unstructured mesh allows any number of elements to meet at a single node.

The previous idea is clearly illustrated in Figure 3.2: each vertex of the structured mesh, with the exception of boundary vertices, is surrounded by four quadrilaterals; on the other hand, each vertex of the unstructured mesh is surrounded by a variable number of triangles.

Sometimes, the shape of the mesh elements is considered enough for categorizing meshes as structured or unstructured. For example, in two dimensions a structured mesh typically uses quadrilaterals, while an unstructured mesh uses triangles. However, there is no reason for using different element shapes for both structured and unstructured meshes [26, 165]. It is always possible to transform a quadrilateral mesh into a triangular mesh, and vice versa. Figure 3.3, obtained from Bern and Plassmann [26], shows the division of quadrilaterals into triangles on the left side of the figure, and the division of triangles into quadrilaterals on the right side of the figure.

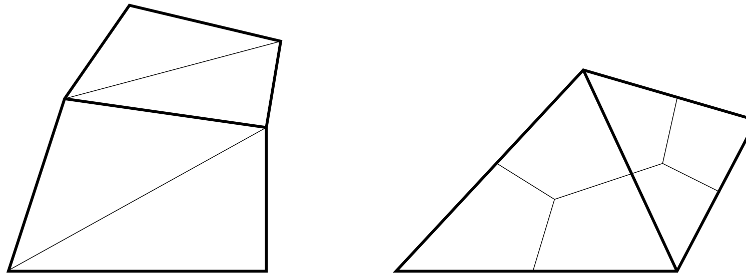


Figure 3.3: Conversion between Triangles and Quadrilaterals

Advantages and disadvantages

Both kinds of meshes have advantages and disadvantages. Structured meshes are easy to generate and manipulate, facilitating the use of simple data structures to reduce programming complexity [26, 228], they require less computer memory, their coordinates can be calculated (rather than explicitly stored) and have more control over the sizes and shapes of elements [26]. On the other hand, 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 [25, 145]. Unstructured meshes offer more convenient mesh adaptability and a better fit to complicated domains or objects with complicated topologies [26, 78]. However, some aspects of the theory behind unstructured meshes are not as well understood as those for structured meshes [145].

3.4 Algorithms for Mesh Generation

Unstructured mesh generation involves the creation of mesh points and their relevant connectivities. Its stages are summarized as follows [88]:

1. Domain boundary definition: The boundary discretization is represented by a polygonal or polyhedral approximation to the boundary of the real domain.
2. Specification of an element size distribution function: This function is a control element to ensure the fulfillment of certain properties in the mesh generation, such as local element sizes.

3. Generation of a mesh respecting the domain boundaries, or the boundary discretization: The mesh generation itself by means of a given method, e.g. quadtree.

4. Optimization of the element shapes (optional): In general, good quality meshes cannot be generated directly, and a post processing step is required to optimize the mesh with respect to the element shapes.

Most techniques of 2D and 3D mesh generation fall into three categories [26, 78, 165]: Quadtree/Octree, Delaunay, and Advancing Front.

In the following section, quadtree and octree based methods are presented to illustrate the stages of a mesh generation algorithm. Delaunay and advancing front methods can be reviewed in Bern and Eppstein [25], Douglass et al. [78], Frey and George [88], and Owen [165].

3.4.1 Quadtree and Octree methods

These methods are also called decomposition methods [88]. Quadtree is used in two dimensions and octree in three dimensions.

A quadtree is a recursive partition of a region of the plane into axis-aligned squares. The root square covers the entire region. Any square, including the root, can be divided into four child squares, by splitting it with horizontal and vertical line segments through its center. Then, the collection of squares forms a tree, with smaller squares at lower levels of the tree, called leaf squares [25].

Quadtree definition

Similarly, an octree is the three dimensional version of a quadtree. The root of an octree is a box covering the entire domain Ω, over which the discretization is applied. An octree is constructed by recursively and adaptively dividing a box into eight childboxes, by splitting it with hyperplanes normal to each axis through its center [227].

Octree definition

Figure 3.4 shows a quadtree on the left side of the figure and an octree on the right side. Figures were obtained from Bern and Eppstein [25] and Teng [227] respectively.

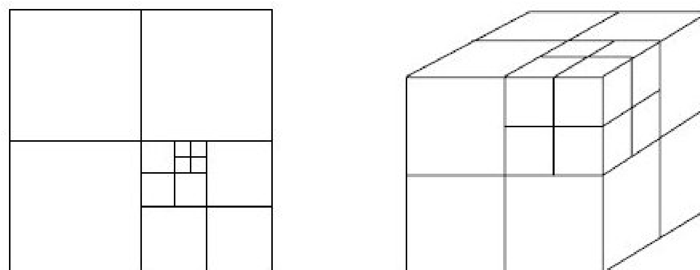


Figure 3.4: Quadtree and Octree

The concept of a balanced quadtree is important in quadtree mesh generation, because it

Balanced quadtree

generates a well shaped mesh for any input 2D domain [228]. The condition for a balanced quadtree is that the squares sharing a portion of a side with a leaf square B must be at most twice the size (side length) of B [25].

Figure 3.5 shows both unbalanced and balanced quadtrees.

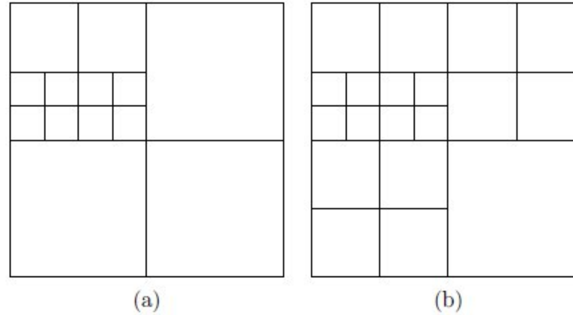


Figure 3.5: Unbalanced Quadtree (a), and Balanced Quadtree (b)

According to Frey and George [88], the classical quadtree/octree-based technique involves the following steps:

1. Initializations.
    - Boundary discretization.
    - Definition of the size distribution function (if available).
2. Tree decomposition.
    - Initialization: the tree representation is derived from a box enclosing the domain.
    - Recursive subdivision of the box up to a satisfactory criterion.
3. Tree balancing: necessary to limit the difference between neighboring cells to only one level (the so called 2:1 rule).
4. Cell meshing using predefined patterns (internal cells) and local connections (boundary cells).
5. Optimization: topological and geometrical modifications.

These steps can be better understood by examining Figure 3.6, obtained from Frey and George [88]. A 2D domain Ω is presented in (i); the domain Ω is recursively decomposed into a set of disjoint and variable sized cells (squares), which represent a partition of a bounding box $\mathcal{B}(\Omega)$ of the domain Ω, as is shown in (ii). In this figure, each cell contains only one boundary point. Other points are on the side of some cell. Then, each terminal cell is decomposed into a set of elements (triangles or/and quadrilaterals) whose union constitutes the final mesh of the domain Ω, as shown in (iii). Finally, an optimization stage is used to improve the shape quality of the mesh elements, yielding the figure shown in (iv).

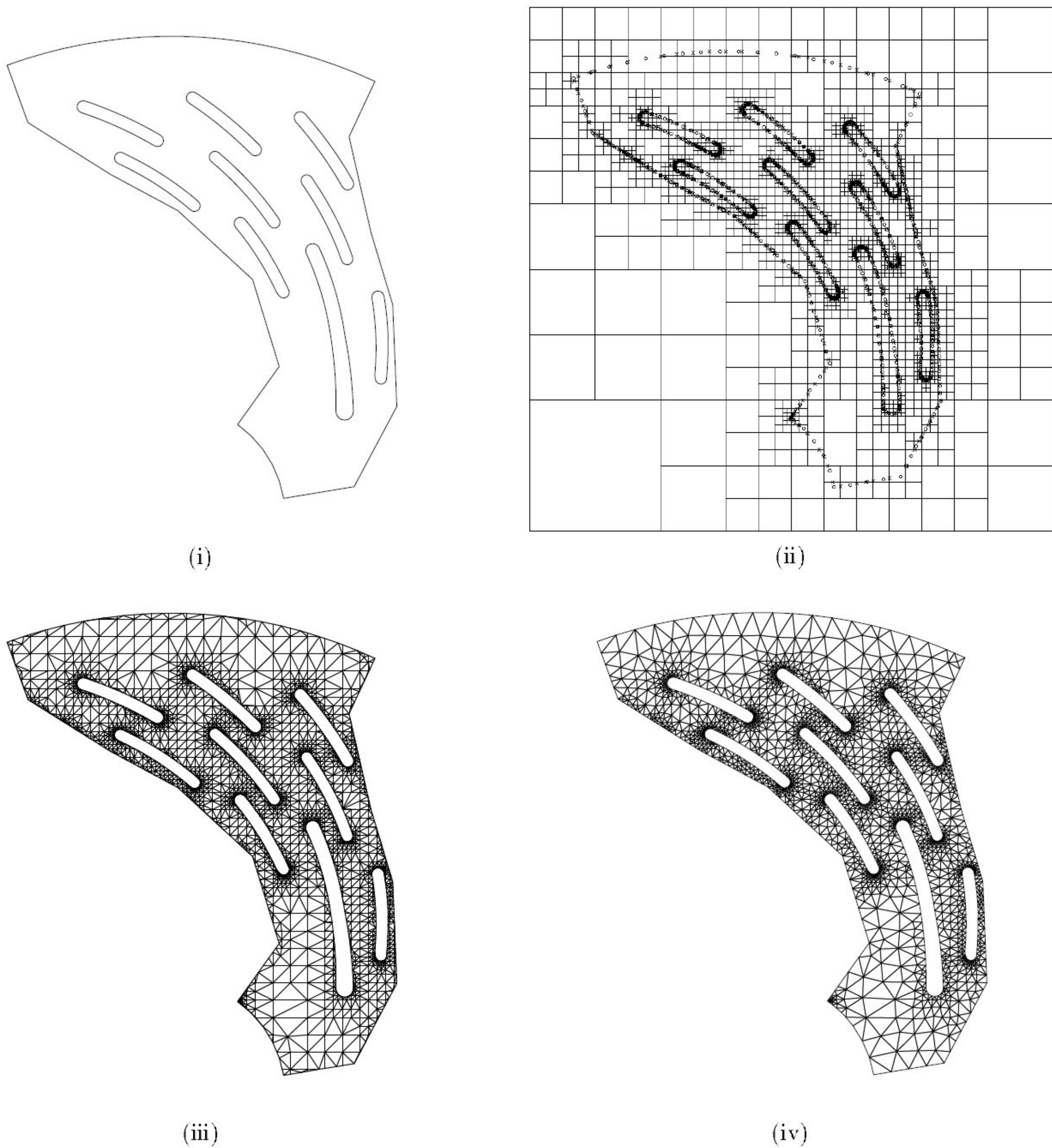


Figure 3.6: Steps for Discretization of a 2D Domain

3.5 Algorithms for Generating an Optimal Mesh

As was sketched in the previous section, mesh generation algorithms do not usually generate an optimal mesh. Thus, an optimization stage is generally incorporated into the mesh generation process in order to improve the quality of the mesh elements.

The following section reviews algorithms for improving the quality of an initial mesh.

3.5.1 Refinement Algorithms

The mesh refinement process usually involves adding more elements (points, triangles, quadrilaterals, etc.) to the whole mesh (global refinement), or to selected regions of interest or of high error (local refinement).

Refinement definition

For Owen [165], refinement is defined as any operation performed on the mesh that effectively reduces the local element size. Methods have been proposed for both 2D meshes (triangles and quadrilaterals) and 3D meshes (tetrahedra and hexahedra).

Triangle partitions

2D refinement algorithms have at their core several techniques for dividing triangles. According to Plaza and Rivara [173] and Suárez et al. [223] there are at least six partitions of a triangle, as illustrated in Figure 3.7.

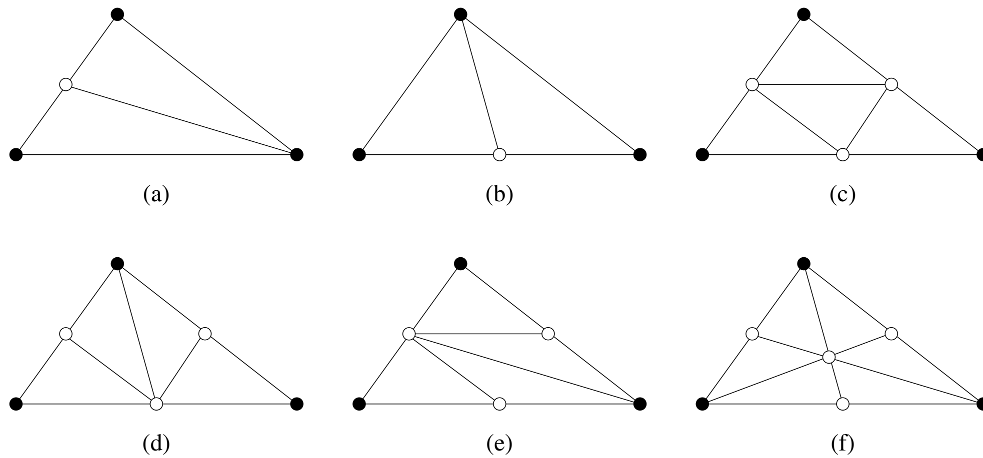


Figure 3.7: Triangle Partitions

The following definitions, taken from Suárez et al. [223] and Plaza and Rivara [173], explain the triangle partitions.

Definition 3.1 (Simple edge bisection)**.** *The simple bisection consists in dividing the triangle into two sub-triangles by the union of the midpoint of one of the edges with its opposite vertex (Part (a) of Figure 3.7).*

Definition 3.2 (Longest edge bisection)**.** *The longest-edge bisection consists in dividing the triangle into two sub-triangles by the union of the midpoint of the longest-edge with its opposite vertex (Part (b) of Figure 3.7).*

Definition 3.3 (The 4-triangles similar partition)**.** *The triangle is divided into four similar triangles by connecting the edge midpoints by means of line segments parallel to the edges of the triangle. Therefore, the resulting sub-triangles are similar to the original one (Part (c) of Figure 3.7).*

Definition 3.4 (The 4-triangles longest edge (4T-LE) partition)**.** *The triangle is bisected by its longest edge, followed by the edge bisection of the resulting triangles by the remaining original edge of the triangle (Part (d) of Figure 3.7).*

Definition 3.5 (The 4-triangles shortest edge partition)**.** *The triangle is bisected by its shortest edge, followed by the edge bisection of the resulting triangles by the remaining original edge of the triangle (Part (e) of Figure 3.7).*

Definition 3.6 (Barycentric partition)**.** *The barycentric partition consists in connecting the barycenter of the original triangle with the vertex and edge midpoints of it (Part (f) of Figure 3.7).*

Similarly, there are partitions and methods for tetrahedra refinement in 3D. For instance, Plaza and Rivara [173] describe the 8-tetrahedra longest-edge (8T-LE) partition, the 3D Freudenthal-Bey partition, and the 3D barycentric partition.

Owen [165] states that the above mentioned 2D and 3D strategies fall into the category of edge bisection strategies. Owen presents another category for refinement approaches, called point insertion. According to the author, this kind of approach consists of inserting a single node at the centroid of an existing element, triangle or tetrahedron, dividing it into three or four elements, respectively.

More information about refinement algorithms can be found in Miller et al. [146], Plaza and Carey [171], Plaza et al. [172], Plaza and Rivara [173], Rivara [184], Rivara [185], Shewchuk [207], Staten and Canann [221], and Teng and Wong [228].

3.5.2 Derefinement Algorithms

The mesh derefinement or mesh coarsening process usually involves removing mesh elements (points, triangles, quadrilaterals, etc.) from the whole mesh, or from selected regions.

An initial mesh commonly contains some regions that have only a few elements, and/or some other regions that have many elements. Sometimes, due to the nature of the problem to be solved and the domain, some of these regions must be refined and others derefined. For example, mesh refinement increases the number of elements and vertices, which in turn can also increase the computation time for the entire model. In order to maintain a similar computation time, while increasing resolution and accuracy in the results, it is sometimes necessary to refine the main areas of interest and to derefine other mesh regions [205].

Two commonly used simplification operations are vertex remove and edge collapse [53].

`Simplification operations`

In a triangular mesh, the vertex remove operation consists in removing a single vertex and all the triangles touching it. This creates a hole that must be filled with a new set of triangles. Figure 3.8, obtained from Cohen [53], illustrates the process.

`Vertex remove`

In this case, the vertex has six adjacent triangles, and the elimination of that vertex creates a hole with six sides. In this example, the six triangles around the vertex are replaced by a new triangulation with four triangles.

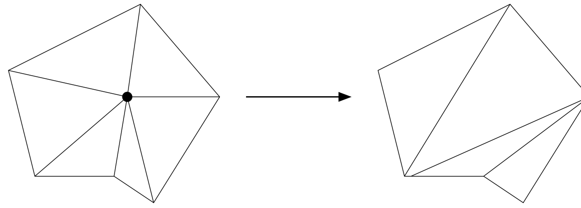


Figure 3.8: Vertex Remove Operation

The number of possible retriangulations for a $n+2$-sided hole is given by the Catalan sequence, that describes the number of unique ways in which a convex planar polygon can be triangulated. In the case of a $n+2$-sided concave planar polygon, the number of possible triangulations is bound by the corresponding Catalan sequence.

Catalan sequence

The Catalan sequence is:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\ n!} \qquad n \geq 0$$

For instance, if a vertex is removed leaving a 5-sided concave hole, then $C_n = C_3 = 5$, i.e., there are five possible triangulations, which are shown in Figure 3.9.

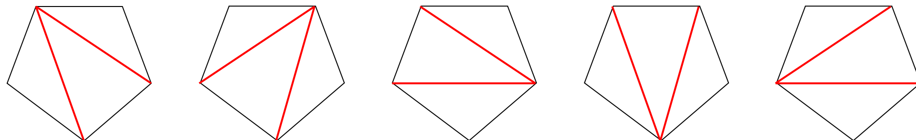


Figure 3.9: Different Triangulations for a Pentagon

Edge collapse

On the other hand, edge collapse consists in merging an edge's two vertices into a single vertex. With this, the triangles containing both of these vertices transform into 1D edges, and therefore are removed from the mesh. Each application of this operation reduces the number of triangles in the mesh by two.

Figure 3.10, obtained from Cohen [53], illustrates the process.

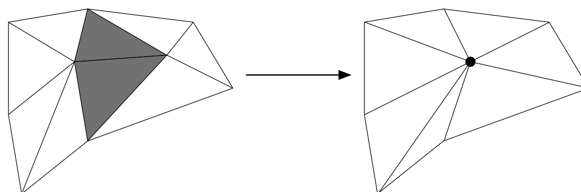


Figure 3.10: Edge Collapse Operation

More information about derefinement algorithms can be found in Bern and Plassmann [26], Cohen [53], Miller et al. [145], Miller et al. [146], and Shepherd et al. [205]

3.5.3 Improvement Algorithms

For Owen [165], mesh improvement, also called mesh clean-up, refers to any process that changes the element connectivity.

Improve-ment definition

According to Canann et al. [36], and considering interior triangular mesh improvement, all improvement algorithms can be based on three operations: element collapse, element open, and diagonal swap, which can be implemented as a third basic operation or as a combination of the two previous operations. In general, the element collapse operation is based on the edge collapse operation described in Section 3.5.2. Thus, it is possible to consider it as a derefinement operation instead of an improvement operation[1].

The element open and diagonal swap operations are described in the following paragraphs.

The element open operation adds elements to the mesh. In Figure 3.11, adapted from Canann et al. [36], it is possible to see that the element open operation along $a - b - c$ adds a new vertex d', three new edges $a'd'$, $b'd'$ and $c'd'$ and two new elements P and Q.

Element open

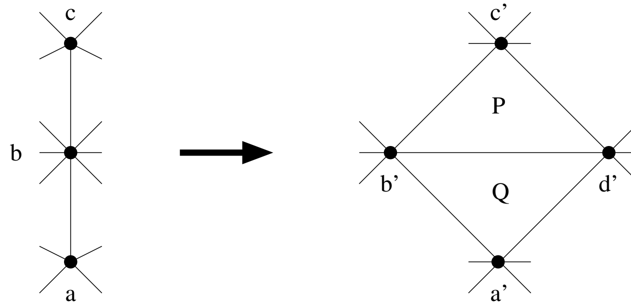


Figure 3.11: Element Open Operation

The diagonal swap operation swaps an existing shared edge between two edge-adjacent triangles, in such a way that two new edge-adjacent triangles are built. Figure 3.12, adapted from Canann et al. [36], shows the operation: triangles abd and bcd share edge db; this shared edge is swapped with edge $a'c'$, producing the triangles $a'b'c'$ and $a'c'd'$.

Diagonal swap

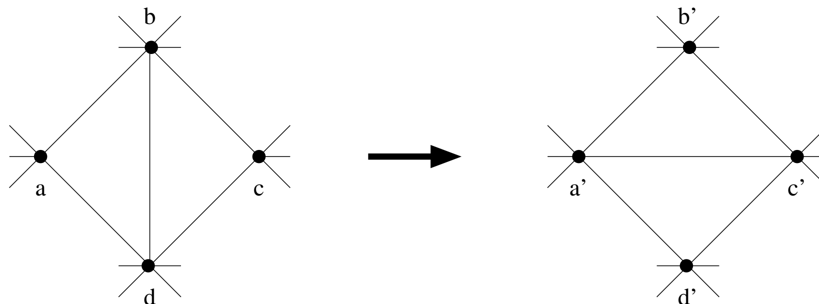


Figure 3.12: Diagonal Swap Operation

[1]Nancy Hitschfeld-Kahler. Personal communication, August 2011. On Edge Collapse and Element Collapse Operations.

On the other hand, quadrilateral meshes have similar operations, which are discussed in depth in [37].

An interesting point to be considered is that several refinement algorithms can also be used as improvement algorithms[2]. More information about improvement algorithms can be found in [37], Hitschfeld et al. [100], Kinney [122], Owen [165], Rivara [183], and Staten and Canann [221]

3.5.4 Optimization Algorithms

Optimization or smoothing refers to any process that adjusts vertex locations while maintaining the element connectivity [165] with the purpose of improving the quality of the mesh without changing its topology [228]

Optimization definition

According to Owen [165], there are a variety of optimization techniques, classified in four categories:

1. Averaging methods.
2. Optimization-based methods.
3. Physically-based methods.
4. Mid-node placement.

Laplacian smoothing, a commonly used averaging technique is hereby shown as an example of an optimization algorithm [228] and illustrated in Figure 3.13, obtained from Vollmer et al. [245]. This heuristic consists of moving each internal point (e.g., q_i) to the barycenter (p_i) of its direct neighbors (q_{j1}, q_{j2}, q_{j3}, q_{j4}, q_{j5}, q_{j6}).

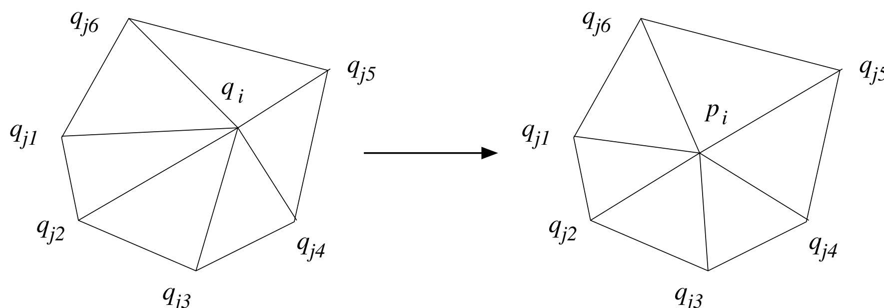


Figure 3.13: Laplacian Smoothing Algorithm

[2]Nancy Hitschfeld-Kahler. Personal communication, May 2008. On Refinement and Improvement Algorithms.

From the Figure 3.13, q_i represents the current point (before the application of the algorithm) and p_i the new position (after the application of the algorithm), which is given by the following formula:

$$\mathrm{p}_i = \begin{cases} \frac{1}{|Adj(i)|} \sum_{j \in Adj(i)} \mathrm{q}_j, & i \in V_{var}, \\ \mathrm{q}_i, & i \in V_{fix}. \end{cases}$$

where $Adj(i)$ denotes the set of adjacent vertices of one vertex i, V_{var} denotes the set of movable vertices, that is, the set of vertices whose position can be modified, and V_{fix} the set of vertices that are fixed [228].

Other examples of optimization techniques belonging to other categories presented by Owen [165] can be studied in his article.

More about optimization algorithms can be found in the work of Bern and Eppstein [25], Djidjev [75], Knupp [124], Owen [165], and Vollmer et al. [245]

3.6 Input/Output File Format

When meshes are generated, they are mainly stored in RAM using a suitable data structure. Once they conform to the different quality criteria, it is necessary to store them in a persistent media, using some specific format, so they can be used by other applications.

There are several file formats useful for both input and output data. Some of them are described by Phillips et al. [170]. In the following sections two of them will be shown. Their descriptions were obtained from Melo [141], Phillips et al. [170] and Silva [210], and the example used for describing each format is shown in Figure 3.14 and was adapted from Melo [141].

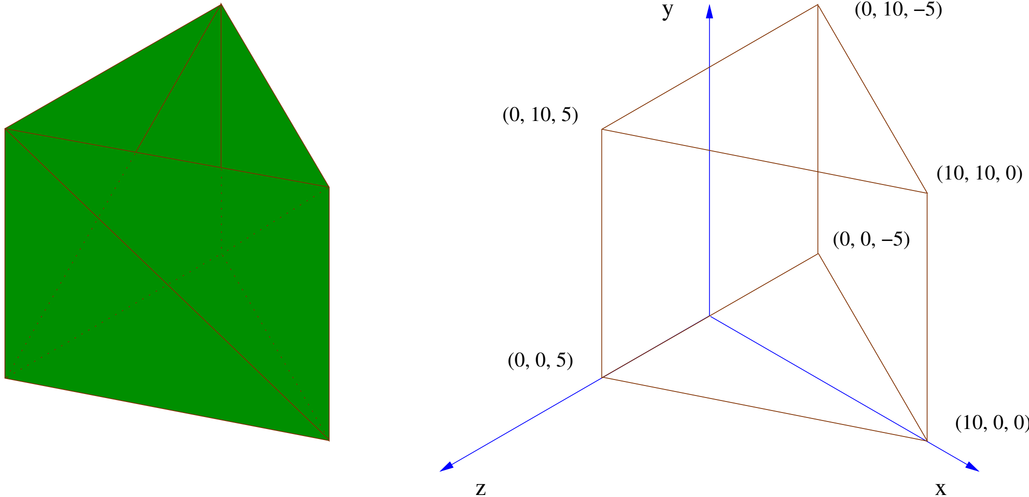


Figure 3.14: Mesh to be Represented in Different Formats

3.6.1 Object File Format (off)

Mesh data is stored in plaintext, with the first line or header of each file generally containing descriptive information, as shown in Figure 3.15. The first row contains information about the file's format type. The second row has information about the number of vertices, faces and edges, which is used mainly to verify that the number of vertices in the file correspond to the data in the file. The following six rows are the vertices themselves in a x y z format. Finally, information about the faces is provided. The first element is the number of the vertices for the face, following by the vertex indices in the range *[0..Number_of_Vertices – 1]*. At the end of the row, the color specification of the face is indicated in the R G B format. In this case, all faces are green.

```
OFF
6 6 12
10 0 0
0 0 5
0 0 -5
10 10 0
0 10 5
0 10 -5
3 0 4 1 0 1 0
3 0 3 4 0 1 0
3 1 5 2 0 1 0
3 1 4 5 0 1 0
3 2 3 0 0 1 0
3 2 5 3 0 1 0
```

Figure 3.15: Example of a Mesh in Object File Format

3.6.2 Comsol Format

In this format, data is stored in plaintext, as shown in Figure 3.16. The file consists of two parts, indicated by the delimiters `% Coordinates` and `% Elements`. In the example, six vertices are described in a x y z format after the delimiter `% Coordinates`. The delimiter `% Elements (triangular)` marks the start of the list of faces and their type. Each face's vertices are referenced in the range *[1..Number_of_Vertices]*.

3.7 Postprocessing Algorithms

Mesh postprocessing is performed to modify some of its characteristics as a last step before storing it. In this sense, it is possible to identify at least two common postprocessing tasks: transforming a triangular mesh into a quadrilateral one, and viceversa [141].

```
% Coordinates
10 0 0
0 0 5
0 0 -5
10 10 0
0 10 5
0 10 -5
% Elements (triangular)
2 5 1
5 4 1
3 6 2
6 5 2
1 4 3
4 6 3
```

Figure 3.16: Example of a Mesh in Comsol Format

3.7.1 Triangle to Quadrilateral

This process transforms a triangular mesh into a quadrilateral one. The algorithm consists of dividing each triangular face in three quadrilaterals, inserting a vertex in the centroid of the triangle and a vertex in each edge of the triangle, as shown in Figure 3.17. This algorithm is applied to all triangular faces of the mesh. When dividing an edge, it must be considered that the edge belongs to two faces, and each edge must be divided only once.

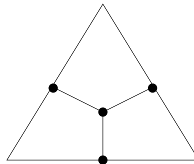


Figure 3.17: Transformation from a Triangle to Quadrilaterals

3.7.2 Quadrilateral to Triangle

This process transforms a quadrilateral mesh into a triangular one. The algorithm consists of adding a diagonal to the quadrilateral, as shown in Figure 3.18. The algorithm must be applied to all quadrilateral faces of the mesh.

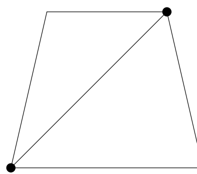


Figure 3.18: Transformation from a Quadrilateral to Triangles

3.8 Visualization

Nowadays, it is very common to find meshing tools that integrate visualization functionalities in order to show the results of the mesh generation process immediately to the users. All meshing tools shown in Table 1.1 in page 3 incorporate visualization capabilities. However, separate mesh generators and mesh visualizers are also common. Well-known examples are TetGen [208] and TetView [209], currently maintained by the same research group. The following paragraphs briefly describe several mesh visualization tools.

TetView: TetView is a program for viewing tetrahedral meshes and piecewise linear complexes (PLC). It was written in C++, specifically for viewing and analyzing the input and output files of TetGen [209], even though it can also show meshes in at least nine formats, among them **off** and **mesh** (MEDIT tool format).

MEDIT: OpenGL-based visualization software, developed to visualize numerical simulation results on unstructured meshes in two and three dimensions, and surfaces [87]. It uses its own format (**mesh**) and an old data format from previous versions (**msh2**). Additionally, it supports a **gis** format, specially designed for terrain description.

GeomView: Written in C, it can be used as a standalone viewer or to display input data from another program that is running simultaneously. It manages more than ten file formats, among them **off** and **mesh** [170].

3.9 Developing Meshing Tools

Only in the last fifteen years has meshing software development been approached from the software engineering point of view mainly by applying object-oriented design and programming. In general, it is a known fact that there is a gap between software engineering developers and computational scientists that use or develop scientific software such as meshing tools [118].

`Traditional software development`

Some recent works include the development of a software environment for the numerical solution of partial differential equations (Diffpack) [32], the design of generic extensible geometry interfaces between CAD modelers and mesh generators [166, 225], the design of object-oriented data structures and procedural classes for mesh generation [147], the computational geometry algorithm library CGAL [83], the definition of an optimal OO mesh representation that allows the programmer to build efficient algorithms (AOMD) [181], and algorithms that can be used independently of the concrete mesh representations [28], as well as a tool to support these algorithms (Grid Algorithms Library) [29]. More recently, formal methods have also been used to improve reliability of mesh generation software [81].

`With reuse software development`

On the other hand, from a reuse point of view, some recent works include the analysis and design of product families of meshing tools [18, 20, 188] taking into account some implementation aspects and formalizing the domain analysis, the use of a known approach called

commonality analysis [211, 212, 213, 214, 216] developed by Weiss [246] and that consists of systematically identifying and documenting both the common and variable characteristics that all program family members share, and the terms used by describing the family. A good usage example [249] of the commonality analysis approach is developed by a Finite Element Analysis software.

Furthermore, there is interest in improving the quality of software development in the field of scientific computing [215] by focusing attention on quality attributes such as reliability, performance, understandability, reusability, verifiability and maintainability, among others.

Finally, Chui et al. [43] present a component-oriented software toolkit that was developed using a systematic software engineering approach based on the component object model (COM), which enables the combination of existing mesh generation algorithms in an easy way.

Chapter 4

Domain Analysis for the Meshing Tool Domain

In the following sections, the domain analysis stage is presented, along with its definition and its corresponding substages. The particular process used to define both the meshing tool domain model and the scope is presented through UML activity diagrams. The input and output artifacts are described, and the domain model is formalized.

4.1 The Domain Analysis Stage

In the context of SPL, domain analysis is the first stage within the domain engineering phase. The term *domain analysis* was introduced for the first time by Neighbors [152] as

Domain analysis

> *"the activity of identifying the objects and operations of a class of similar systems in a particular problem domain."*

According to Prieto-Díaz [178], the difference between system analysis and domain analysis is that the former is centered on a specific system while the latter is concerned with all systems in an application area.

Any domain analysis approach has four goals [111]:

Domain analysis goals

1. to identify and document requirements;
2. to produce an integrated and concise model that captures the requirements;
3. to build a requirements document that supports planning a good development strategy for a particular product;

4. to produce a domain model that may be used for developing specific applications, as well as for guiding the development and evolution of a reuse infrastructure.

As mentioned in Section 2.2, most SPL efforts are centered on areas where several single product developments already exist. This, in turn, makes it possible to start a domain analysis using these products as a basis.

Figure 4.1 illustrates this domain analysis process [111]. The space axis shows the set of requirements from existing single systems at the same point in time; the time axis captures the changes of the requirements on a particular system. The gray ellipse represents the scope of the SPL, i.e., the products that can be produced as part of the product line. The scope will be addressed in depth in Section 4.1.2.

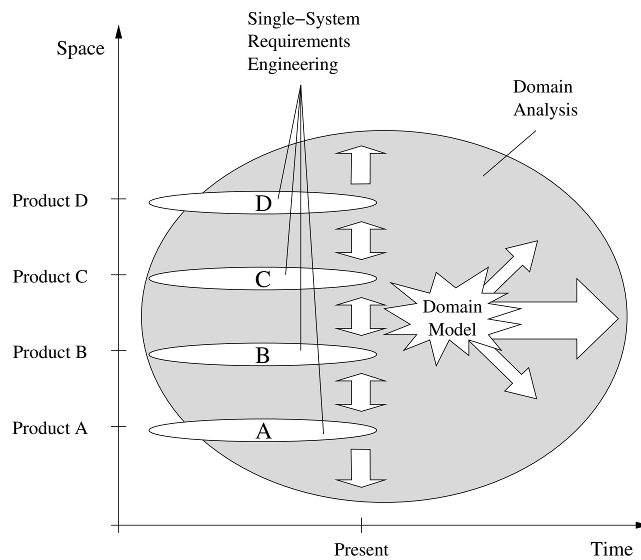


Figure 4.1: Domain Analysis from Existing Products

In contrast, sometimes organizations do not have access to a sufficiently large number of existing products on which to base the SPL. In these cases, the approach presented in Figure 4.1 fails, and the need for more elements or artifacts in the domain analysis process is evident. This thesis presents strategies to overcome this issue.

The domain analysis stage consists of two substages: scoping and domain modeling. Scoping is the process of defining which products are part of the SPL and which are not. Domain modeling is the process whereby commonalities and variabilities are identified, captured and organized in a domain model with the purpose of characterizing the domain [178]. In the following sections, both these substages are presented in more detail.

4.1.1 Domain Model

The domain model is a product of the domain analysis stage. A domain model is a definition of the functions, objects, data, and relationships in a particular domain [115]. Domain

modeling has been identified as one of the most important factors for the success of software reuse [4], as it allows recording and organizing existing domain knowledge.

For Czarnecki and Eisenecker [60], a

Domain model

> *"domain model is an explicit representation of the common and the variable properties of the system in a domain, the semantics of the properties and the domain concepts, and the dependencies between the variable properties."*

Even though the above definition puts emphasis in the explicit representation of the common and variable characteristics of the existing or future applications in the domain, it is unusual to find an exact and standard description about what artifacts compose the domain model and what is the sequence of steps to be followed for building it; this varies from one author to another [35, 60, 66, 174, 226]. The PuLSE methodology [22] is an important exception, as it provides a thorough description of the domain model building through a complete series of process diagrams which define the products, data and control flow.

The activities for domain model development depend on those artifacts identified as belonging to the model. For example, both de Almeida et al. [66] and Campos and Zorzo [35] define the feature model as the only domain model artifact. The feature model identifies the common and variable features of the domain concepts and the dependencies between the variable features [59]. According to Campos and Zorzo [35], producing the feature model as a domain model artifact involves the following three activities:

1. Commonality analysis: identifies which features (requirements) are common to all applications of the domain.
2. Variability analysis: identifies which features (requirements) are variables to all applications of the domain.
3. Domain modeling: establishes relationships among common and variable features in a suitable model.

On the other hand, Taylor et al. [226] state that there are at least two important artifacts to be considered in a domain model: a domain dictionary, which identifies and defines the terms used in the domain model, and a feature model. They do not explain how to build these artifacts, but rather present examples to illustrate each artifact.

4.1.2 Scope

Domain analysis is a creative activity that involves determining, among other things, the scope of the family to be built [60]. Decisions about scope have an important business

impact [234] and determines the long-term viability of the product line [157]. In that sense, the SPL scope is a core asset of the domain engineering activity [157].

Determining the scope of the system, or scoping, is an activity that bounds a system or set of systems by defining those behaviors or aspects that are "in" and those behaviors or aspects that are "out", in such a way that the product line satisfies its business goals.

Northrop and Clements [157] define scope as a description of the products that will constitute the product line or that the product line is capable of including. So, in a simple form, the scope may consist of an enumerated list of product names. For John and Eisenbarth [109]

Scope

> *"Product Line Scoping (PLS) is the process of identifying and bounding capabilities (products, features) and areas (subdomains, existing assets) of the product line where investment into reuse is economically useful and beneficial to product development."*

In summary,

$$PLS = \bigcup_{i=1}^{n} Fi, Si, Ai \in P$$

with

F: Features,

S: Subdomains,

A: Assets,

P: Products of the Product Line,

n: number of the products.

Scope process

Scoping is one of the first activities to be done when a new SPL is started, because it consolidates knowledge about the domain that is needed to start other activities in the SPL. Scoping is an activity that must occur in a continuous way during the life cycle of the SPL [110], due to new business opportunities, market and technology trends, new insights, the organization's market position [60], and new market demands [234], among others.

A core task in scoping is identifying which future components, i.e., functionalities in the SPL, are likely to have a high return on investment [195]. It is not enough to identify potential products for them to be included within the reuse infrastructure. In this sense, candidate

products must not only share commonalities but also be sufficiently different to justify the investment in SPL, and at the same time their differences must not be so drastic that may make their engineering from a common infrastructure simply not economically feasible [69]. If the scope is chosen to be too big, investment will be wasted on assets that will later on not be reused at all or at least not sufficiently often to pay back the initial investment. On the other hand, if the scope is chosen too small, components meant to be reusable will be designed in such a way that they do not support reuse across all the relevant products, or else certain reuse opportunities will be missed [157, 195].

The scope process can be split into activities that belong to three categories as shown in Figure 4.2, obtained from [109].

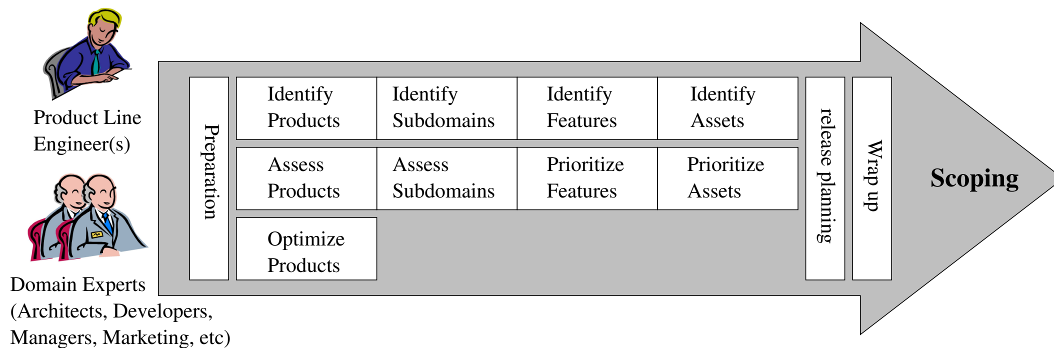


Figure 4.2: Generic Scoping Process

These categories are:

1. Identification activities: They list and describe the respective elements, as shown in the first row of Figure 4.2.
2. Assessment activities: They judge the appropriateness of the elements (e.g., subdomains of the product line) against a set of standardized factors (e.g., amount of variability) and produce a ranking between the elements based on this judgment. They are shown in the second row of Figure 4.2.
3. Optimization activities: They do not only produce a ranking but try to improve the setup of the product line (e.g., by finding an optimized combination of features for the product line products from a marketing point of view). They are shown in the third row of Figure 4.2.

Scoping classification

There are several kinds of scoping classifications. John et al. [110], Schmid [195] and van der Linden [234] discuss the first three variants of scoping; Schmid et al. [196] add the fourth one:

- Product family scoping: This aims at identifying the particular products that ought to be developed as well as the features they should provide.

- Domain scoping: This is the task of bounding the domains (i.e., coherent clusters of functionality) that are supposed to be relevant to the product line.
- Asset scoping: This aims at identifying the particular implementation components (i.e., functional parts of the product line) that should be developed in a reusable manner.
- Life cycle scoping: This aims at reviewing the SPL with respect to evolving requirements and vice versa.

On the other hand, Czarnecki and Eisenecker [60] identified two kinds of scope with respect to the software systems in the domain:

- Horizontal scope (system category scope): Addresses the question "How many different systems are in the domain?"
- Vertical scope (per-system scope): Addresses the question "Which parts of these systems are in the domain?"

These classifications can be integrated by noting that *Product family scoping* is related to Czarnecki and Eisenecker's *Horizontal scope*, and that *Asset scoping* is related to *Vertical scope*.

The scope process defines three product categories. The SPL scope includes all products consistent with the variability defined by the SPL's core assets. To avoid losing business opportunities and maximize the benefits, the SPL scope should be near both the market interests of the company and the full range of functionalities provided by the current products. The markets of interest restrict the scope to a new category that includes only those products that are interesting, marketable or economically viable. The last category includes only those products that have been actually built. Finally, there may be existing products that utilize custom assets requiring out-of-scope variations. In this case, the SPL scope may be augmented to include these features, if economically feasible. These categories are illustrated in Figure 4.3, taken from Gorton [90].

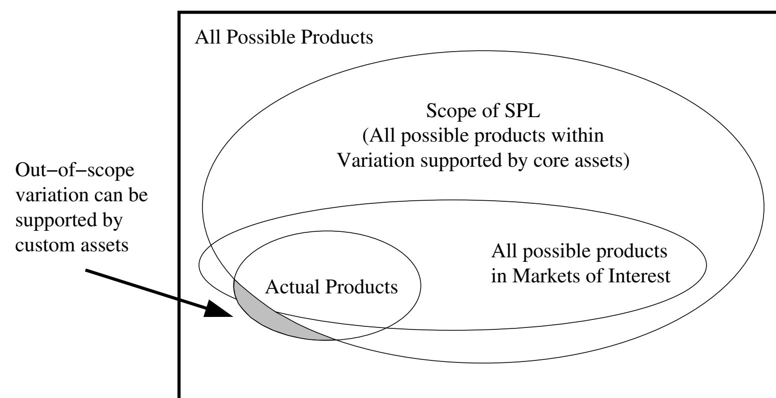


Figure 4.3: The scope of an SPL

Stake-holders

As part of the scope process, it is necessary to identify the domain's stakeholders [51, 60, 154]. This identification is not a static process consisting of just documenting who the stakeholders are, but rather it is a dynamic process of incorporating everyone involved in the domain. Scoping requires participation from a wide range of stakeholders including managers, developers, customers, users, methodologists, and subject-matter experts. Without sufficient stakeholder participation, it could be impossible to develop the domain model, the product line architecture and other core assets.

Inputs from stakeholders are required to avoid building an inappropriate scope for the SPL. Also, their involvement increases the awareness of product line efforts [157]. Some authors such as [22, 69] rely on information provided by stakeholders in the scoping, even though they do not explicitly perform a stakeholder identification activity. Furthermore, Niemelä and Immonen [154] state the importance of defining a stakeholders list to be able to access their domain knowledge. The scope process proposed in this thesis builds a stakeholders list similar to Niemelä and Immonen, i.e., a list that considers three types of stakeholders: business stakeholders, domain stakeholders (domain experts) and product stakeholders. These stakeholder categories are used in Section 4.2.1.1.

Scope evolution

The scoping involves gathering information from different sources, such as existing systems, domain knowledge, and documentation, among others. Therefore, it is necessary to iterate and contrast all available information. Figure 4.4 represents the evolution of a product line scope as a function of time [51].

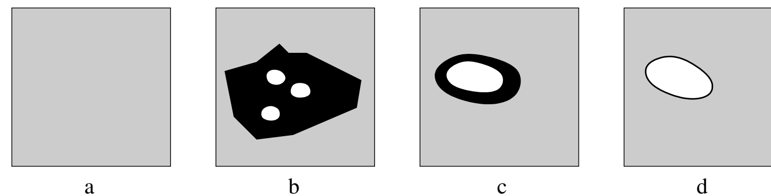


Figure 4.4: Evolution of a SPL Scope

The square in part **a** of Figure 4.4 represents every possible software system in the domain. Part **b** shows the system space divided into three parts: systems that are outside the SPL scope (gray), systems that are inside (white), and systems that may or may not be inside the scope (black). The process continues narrowing down the “not sure” space by carefully defining more of the “out” and “in” spaces, until conceptually it looks like part **c**. Part **d** shows a completely precise scope, where the products that are “in” and “out” of the scope of the SPL have been totally defined.

It seems part **c** is more desirable than part **d**, because **d** is more constrained than **c** and may cause the organization to lose business opportunities previously unpredicted.

According to the process described in the two previous paragraphs, it is clear that scoping is one of the most critical success factors in building a SPL [182]:

1. It influences the development effort during later changes of the reusable platform (common assets of the whole SPL for supporting software development for several products).

Scoping represents the planning of reuse and therefore influences ROI.

2. If it is performed according to the customers' needs, it enables a short time to market and low development costs.

3. By planning reusability, it leads to a higher process maturity for the reusable platform. Thus, software quality and evolvability are improved. Evolvability enables a longer usage time for the parts of the SPL and therefore influences the ROI too.

A recurrent question is the difference between requirements and scope. Clements [47] gives some ideas:

Requirements vs Scope

- Scope includes aspects of the system that would not appear in even the most complete requirements specification, aspects related to business goals or construction constrains.
- They are defined at different times. The scope is defined early enough so that a business case can be built and used to see if the SPL is economically viable. The requirements are specified as a prelude to actual development.
- Scope and requirements have different consumers. The scope is written for people such as marketers, who need to see what they will be asked to sell but do not require full statements of product behavior. The requirements specification is used by the architect and the developers of core assets and products who do need to know exact behavior.
- There is no mandate for the scope to be completely precise. Furthermore, a completely precise scope overly constrains the SPL development process.

4.2 Domain Analysis Method for the Meshing Tool Domain

This section describes the domain analysis method proposed in this thesis for the meshing tool domain. It is divided into four subsections. First, the process for producing the SPL domain model is presented and the domain model is formalized using Z schemas. Then, the process for determining the SPL scope is shown. Finally, several domain analysis methods are discussed, as well as other approaches that have been used to build meshing tools.

4.2.1 Domain Model Construction Process

Processes are defined by identifying roles, artifacts and activities. The artifacts are those work products that need to be generated in order to complete the domain model. Activities are those that are carried out by the stakeholders to build the artifacts. This section describes each of these elements, identifies both the relevant stakeholders and their respective roles, and includes an activity diagram to depict the complete process.

4.2.1.1 Stakeholders

Building a domain model for a complex domain is hard, so people involved should have a clear idea about the role they should play. In any SPL, stakeholders can be classified into three groups [154]: business, domain, and product stakeholders. The following list specifies relevant stakeholders for each group specifically for the meshing tool domain. This list is by no means exhaustive, as domain stakeholders and domain experts will be specific to each particular domain.

- Business Stakeholders: Customers, end users.
- Domain Stakeholders (Domain Experts)
  - Semiconductors modeling: Physicist, electrical engineer, computational engineer.
  - Tree growth modeling: Forest engineer, biologist.
  - Car design modeling: Physicist, mechanical engineer, computational engineer, mathematician.
  - Facial modeling: Computational engineer, film producer.
  - Atmospheric phenomena modeling: physicist, geophysicist, meteorologist.
- Product Stakeholders: Domain analysts, product managers, family architects, product architects, component designers, component developers, component integrators, product developers, product maintainers.

For the process of building a domain model for the meshing tool SPL, the most relevant stakeholders are the domain experts, who provide all the domain knowledge, and the domain analysts, who put together all the knowledge in a structured and organized way.

4.2.1.2 Artifacts

This thesis defines a domain model based on goals, scenarios and features, similar to Park et al. [167], and adds a lexicon for describing the concepts within the domain. Figure 4.5 summarizes the different elements of the domain model and their relationships, which are formalized in Section 4.2.2.

The approach followed in this thesis considers not only the feature model as the main artifact of the DA process, but also the goals, scenarios, actions and lexicon. These elements and the relationships among them allow preserving explicitly the rationale for building both the domain model and the domain scope. This approach is quite suitable for the meshing tool domain, because it is a well established domain, with a very well-known terminology, and the existence of already developed applications, in which the use scenarios are easy to find.

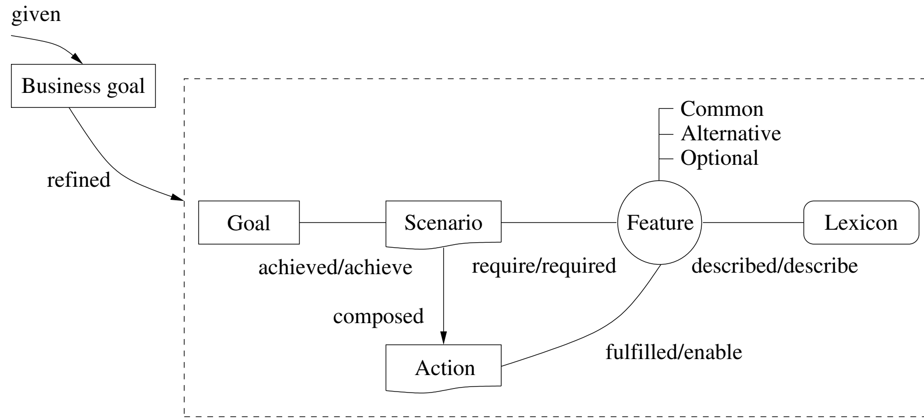


Figure 4.5: Domain Model Artifacts

The business goal states the purpose for developing products as part of a product family and thus gives context to the domain model artifacts. The business goal is unique for the whole SPL, but there may be several particular goals. As the business goal is not part of the domain model, it will not be formalized as the rest of the domain artifacts mentioned in Section 4.2.2.

The set of goals identified as part of the domain model must fulfill the purpose of the business goal as a whole. These goals are stated in natural language, as they are provided mainly by the domain expert in her/his own terms.

Scenarios can be typically divided into development scenarios, which are those followed whenever a product of the SPL is being built, and use scenarios, which are followed by particular products while they are being executed. This thesis considers only use scenarios, and uses structured natural language to establish scenarios as a sequence of atomical actions, similarly to Kim et al. [119]. This decision is made considering that, in general, it is easier to review already developed applications and register the sequences of actions and conforming scenarios than to find correctly documented applications and to find in that documentation the actions followed to develop the applications. Meshing tools are not the exception. A large variety of meshing tools can be found in Schneiders [200], and most of them do not include complete development documentation.

In this particular domain, those data storage, parameters, functionalities or user interactions identified for the potential products in the SPL are considered features; they may be either common for all products in the SPL (mandatory), optional or alternative. Features are generally inspired by components already developed and present in existing products as well as by concepts from the application domain. As is usual practice in SPL development, this thesis documents software features using a feature model which includes a tree-like structure as well as a series of constraints among the included features [60, 115]. In other domains, such as Computer Supported Cooperative Work (CSCW), it is common to find other kinds of features, such as communication, interoperability, and awareness, among others [97]. Applying the DA approach presented in this thesis to the CSCW domain requires evaluating how well this approach fits the CSCW domain. As a result, some changes may be required.

The lexicon is, by definition, a natural language explanation of the concepts involved in the domain. This work proposes that at least all identified features should be described in the lexicon.

4.2.1.3 Activities

Even though the existing literature is rich in proposals for domain model specification, it is sorely lacking in the description of the actual steps to be performed in order to build the model. In contrast, this work proposes a rigorous process for guiding the activities that need to be carried out in order to build this model. This process is specified using the UML activity diagram shown in Figure 4.6[1].

Stakeholder identification is the first activity in the process of building the domain model, so as to extract their knowledge and use it building in the domain model and the SPL scope building. It is an important activity in the DA because it determines the people involved in the development and use of the different SPL products. The stakeholders also define the business goal, taking into account their knowledge about the domain and the software products to be developed.

The domain experts and the domain analysts should interact in order to identify and specify goals, features, scenarios, actions, and terms of the lexicon. To this end, a variety of sources of information may be considered complementing the domain experts' knowledge such as available components developed within the domain, external information (e.g. emerging technologies within a domain, market information and literature) and optionally systems information (e.g. system documentation and existent systems developed in the domain). Additionally, considering that the domain model process is part of the domain analysis process, which in turn is an iterative process, it is necessary to consider the features out of the scope as an optional input to the domain model: they are useful when new products are being evaluated for their incorporation to the SPL scope, as part of the product derivation process (Section A.2).

Once the first set of goals, scenarios, actions, features and lexicon is available, the domain analyst needs to establish the relationships among these model elements. The most important relationships are between goals and scenarios, between actions and features, and among features themselves.

Once these activities are done, the domain expert checks for completeness by analyzing if the model elements captured are enough for building all the expected products. Meanwhile the domain analyst checks for consistency by verifying that the domain model satisfies all the consistency conditions. If any of these conditions (completeness or consistency) does not hold, then the process should iterate. Otherwise the domain model can be considered to be ready and we can proceed to the next step in the SPL development.

[1] An early version of the domain analysis process was validated in November 2007 by Todd L. Veldhuizen from Electrical and Computer Engineering Department, University of Waterloo.

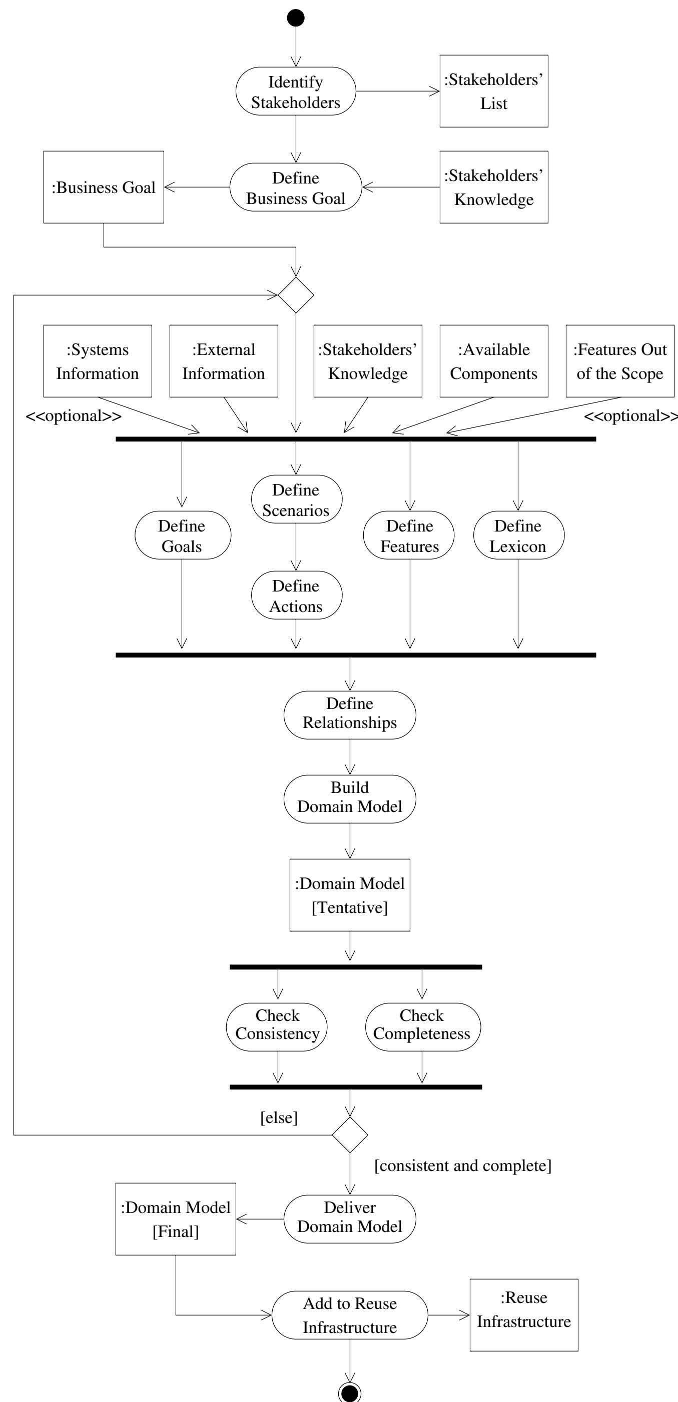


Figure 4.6: Domain Model Construction Process

4.2.2 Domain Model Definition

The proposed domain model includes features, goals, scenarios and lexicon, along with the relationships among these elements. The lexicon defines the domain vocabulary, and allows a better and common understanding for all stakeholders involved in the domain [60]. Several other authors agree on the need for a lexicon in the development of the SPL [115, 129, 214, 246]. This work requires the lexicon define at least all features.

The definitions of features, goals and scenarios are inspired by and adapted from Park et al. [167]. In the context of a product line, a goal is an objective of the business, the organization or the systems that some stakeholder hopes to achieve with that product line.

A scenario is a possible behavior limited to a set of interactions with the purpose of achieving some goals with a product of the product family. Consequently goals can be achieved through the execution of scenarios. Scenarios capture real requirements as they describe real situations or concrete behaviors in terms of actions. Thus, a scenario is generally composed of a sequence of one or more actions corresponding to user or system interactions with products in the product line.

Features are characteristics and abstractions of product functionalities, parameters, data storages and user interactions in a SPL visible to stakeholders, and thus they can be viewed as effects achieved by some product behavior (external or internal). As such, a feature is an attribute of a system that directly affects end-users [115]. Product features are related to scenarios through actions, i.e., features are required to implement actions. Features are defined as part of the lexicon.

4.2.2.1 Domain Model Formal Definition

The proposed domain model formal definition uses Z schemas for defining the elements that form part of the domain model and their relationships.

In particular, the use of Z schemas to describe the domain model mainly aims to describe properly, rigourously and in a non-ambiguous manner the relationships among the different artifacts belonging to the domain model, which were sketched in Figure 4.5. There are several approaches to describe the particular artifacts. For example, Kim et al. [120] and Rolland et al. [187] present approaches for goals and scenarios that add structure and rigor, preserving proximity with the stakeholders by the use of natural language. For the feature model, a non exhaustive list of approaches can be obtained from Classen et al. [46], Czarnecki et al. [61] and Janota et al. [107]. The list ranges from graphical (FODA [115]), textual (FMP [2]) and formal (TVL [46]). However, there are no unique approaches to describe the integration of all the domain model's artifacts used in this thesis in a suitable, non-ambiguous and rigorous fashion. The only exception may be the work of Kaindl [114].

Since features can be defined as mandatory (commonalities), or optional and/or alternative (variabilities), this work defines *Feature* as a name and a type, as specified in the *Feature* schema.

$TYPEF ::= GroupedFeature \mid SolitaryFeature \mid RootFeature$

```
┌─ Feature ──────────────────────────
│ name : seq char
│ type : TYPEF
└────────────────────────────────────
```

This thesis also considers *GOAL*, *ACTION*, and *DESCRIPTION* as primitive types, and it defines *SCENARIO* as a sequence of *ACTION*.

$$[GOAL, ACTION, DESCRIPTION]$$

$$SCENARIO == \mathrm{seq}\, ACTION$$

DomainModel

$Goals : \mathbb{P}\, GOAL$
$Scenarios : \mathbb{P}\, SCENARIO$
$Features : \mathbb{P}\, Feature$
$Actions : \mathbb{P}\, ACTION$
$Lexicon : Feature \nrightarrow DESCRIPTION$
$By_Scenario : GOAL \leftrightarrow SCENARIO$ [a]
$By_Feature : SCENARIO \leftrightarrow Feature$ [b]

$Actions = \bigcup_{s \in Scenarios} \mathrm{ran}\, s$ [e]
$\mathrm{dom}\, Lexicon \subseteq Features$ [f]
$\mathrm{dom}\, By_Scenario \subseteq Goals$ [g]
$\mathrm{ran}\, By_Scenario \subseteq Scenarios$ [h]
$\mathrm{dom}\, By_Feature \subseteq Scenarios$ [i]
$\mathrm{ran}\, By_Feature \subseteq Features$ [j]

The schema *DomainModel* defines the elements that form part of the domain model. In this schema the variables are *Goals*, *Scenarios*, *Features*, *Actions* and *Lexicon*. The schema also includes the relationships between goals and scenarios (*By_Scenario* [a]), and between scenarios and features (*By_Feature* [b]), both inspired by the work of Kaindl [114].

Notice that defining the *Lexicon* as a partial function from *Feature* to *DESCRIPTION* is a simplification, since it would eventually be necessary to define other concepts of the application domain as well. Defining the *Features* just as a set of *Feature* is also a simplification because it does not include the feature model structure.

The invariants that any *DomainModel* must satisfy are those stated in the lower part of the schema. The identified actions are those derived from the defined scenarios [e]. The *Lexicon* includes the definition of some of the identified features [f], as there may be some features that are not defined yet. Only those goals, scenarios and features that have been identified as part of the *DomainModel* can be related by the *By_Scenario* and *By_Feature* relations [g,h,i,j].

Domain Models also include constraints on their elements. The *ConstrainedDomainModel* extends the *DomainModel* with the *Requires* [c] and *Excludes* [d] relationships that establish that two features must be together in a product, or cannot be together in a product, respectively. Both, *Requires* and *Excludes*, can only relate already identified *Features* [k,l,m,n]. Although these relationships are not used for consistency checking, they are very important for characterizing the domain and for correct product instantiation.

ConstrainedDomainModel

DomainModel
$Requires : Feature \leftrightarrow Feature$ [c]
$Excludes : Feature \leftrightarrow Feature$ [d]

$\text{dom}\, Requires \subseteq Features$ [k]
$\text{ran}\, Requires \subseteq Features$ [l]
$\text{dom}\, Excludes \subseteq Features$ [m]
$\text{ran}\, Excludes \subseteq Features$ [n]

4.2.2.2 Consistent Domain Model Formal Definition

A consistent domain model is one that may be used as a basis for subsequent steps within the SPL development. Although we may have a transient inconsistent domain model, in the end it needs to be consistent. The *ConsistentDomainModel* schema refines the *ConstrainedDomainModel* schema by adding new constraints. It includes the previous schema *ConstrainedDomainModel*, and also includes the definition of another relationship (*Attached* [o]) between *Actions* and *Features*.

ConsistentDomainModel

ConstrainedDomainModel
$Attached : ACTION \leftrightarrow Feature$ [o]

$\text{dom}\, Lexicon = Features$ [p]
$\text{dom}\, By_Scenario = Goals$ [q]
$\text{dom}\, By_Feature = \text{ran}\, By_Scenario = Scenarios$ [r]
$\text{dom}\, Attached = Actions$ [s]
$\text{ran}\, Attached = Features$ [t]

Within a *ConsistentDomainModel*, all identified *Features* are described in the *Lexicon* [p], all identified *Goals* have a series of related *Scenarios* [q], all identified *Scenarios* contribute to a certain goal and may also be fulfilled with the set of identified *Features* [r]. Finally, all identified *Actions* are attached to at least one feature [s], and all *Features* are attached to at least one action [t]. These conditions are used for consistency checking, which is one of the termination conditions of the proposed process.

4.2.3 Scope Construction Process

The scope construction process is executed once the domain model construction has finished. From a theoretical point of view, separating these two processes is understandable, as it seems that they can be developed in parallel. However, experience in building domain models and scope suggests that both processes must be carried out sequentially, as all iterations with

stakeholders occur in the domain model construction process, and the domain model that is an input to the scoping construction process must be a finished artifact.

4.2.3.1 Artifacts

The *SPL Scope* is constituted by the **Business Goal**, the **Feature Model** and the **Product Map**. Clements [51] states that the scope is refined in terms of products' observable behaviors and quality attributes but does not include goals and scenarios from the domain model. This thesis expands the work done by Rossel et al. [188]: it formally defines relationships among goals, scenarios, and features, and therefore the domain designer can derive behavior and quality attributes from features, if necessary. These three elements are considered scope elements because each one of them constrains the domain one more level, starting from the business goal that is wide but delimited and finishing with the product map that is narrow but coherent with the necessities of the stakeholders.

It is widely accepted that the feature model and the product map are part of the SPL Scope, as can be seen by reviewing the domain analysis methods mentioned in Section 4.2.4. On the other hand, several authors [51, 66, 69, 157, 182, 193, 195] talk about the importance of the business goal in the scoping. Some authors take a different perspective to working with business goals [66, 69, 193, 195]. They identify several business goals and operationalize them into a set of benefit functions. This thesis deals with only one business goal, which in turn gives the context for the domain model and the SPL scope.

The business goal is given by the stakeholders. The feature model is generated by the domain model process. The product map must be built by the domain experts and analysts. This work borrows several ideas for product map construction from Schmid [195] and DeBaud and Schmid [69]. Furthermore, it adopts the idea of extending the feature model with optional extra-functional features such as cost, time and development resources from Trinidad et al. [233]. These extra-functional features may sometimes be hard or impossible to determine accurately for every feature in a given product. However, as cost and development time information are useful when defining the products in the scope, this thesis incorporates this information into the product map instead of into the feature model. Moreover, if the feature diagram is large, drawing the diagram with extra-functional features can be difficult and cumbersome.

Sub-Domain	**Feature**	**Priority**	**Extra-F. F. (Opt.)**		**Existing**	**Planned**		**Potential**
			Time	**Cost**	P1	P2	P3	P4
Sub-Domain 1	Feature 1.1	4	3 days	100 US$	X	—	X	X
	Feature 1.2	1	6 days	200 US$	—	X	—	—
	Feature 1.3	3	2 days	80 US$	—	X	—	X
...	...	...	...	...	...	...	...	...
Sub-Domain n	Feature n.1	2	10 days	300 US$	X	—	X	X
	Feature n.2	5	4 days	100 US$	X	X	X	—
	...	...	...	...	...	...	...	...

Table 4.1: Product Map

Table 4.1 presents the format of the product map. The columns hold the software products, and the rows hold all features. Stakeholders fill the table, indicating which features are

present in which products.

Similarly to other authors [22, 66, 69, 195], three types of products were identified:

1. Existing products: Products that have been developed prior to the start of the SPL project.
2. Planned products: Products where the requirements are rather clear, but development has not yet started.
3. Potential products: Products for which no clear requirements exist yet, but these products are seen as relevant, for example, because they address new market segments.

These kinds of products fit well with the meshing tool domain, as there are several already existing meshing tools, and it is easy to deduce their features. Similarly, and considering the previously identified existing products, several more features can be derived or expanded from them to constitute the planned products. Finally, potential products arise from the feature model and market necessity.

Once the features for each product are established, the stakeholders must prioritize the features so as to know which features will be implemented as part of the next development cycle, which features will be implemented in a later development cycle, or which ones will not be implemented for the moment [182]. Feature prioritization is also used by Czarnecki and Eisenecker [60] and de Almeida et al. [66].

How to assign priorities to the features? Obviously, this is a subjective activity which depends on the expertise of the stakeholders involved in the process. The following paragraph gives some guidelines for helping stakeholders in this activity.

- If the feature is present in many products, then its priority is high.
- If the feature is important for developing several products, then its priority is high.
- If the feature is only present in potential products, then its priority is low.
- If the feature is mandatory, then its priority is high.
- If the feature is optional, then its priority is low.

Finally, the extra-functional features in the product map, such as time and development costs, must be filled. As with feature priorities, these values must be decided by the stakeholders, and will be useful when building a production plan in the domain design step.

The scope definition process described above is *Feature Oriented*: it first identifies the characteristics that are needed in the SPL, considering several kinds of inputs as shown in

Figure 4.6. Then, it summarizes those characteristics found in the domain analysis process in a table such as Table 4.1. In contrast, PuLSE-Eco is a *Product Oriented* process where the first step is to identify the products and their characteristics [69]. PuLSE-Eco uses a table similar to Table 4.1 at the beginning of its process, and later iterates over this table to build the scope. Finally, Niemelä and Immonen [154] describe a *Goal Oriented* process for determining the scope.

The scoping presented in this thesis seems more time-consuming and resource-intensive than PuLSE-Eco. Not only must it determine the features, but it also must determine scenarios, actions, goals and lexicon, while guaranteeing consistency and completeness. Nevertheless, it benefits from these actions by generating a domain model and a SPL scope that are consistent and with a high probability complete, consistency and completeness being two qualities that are difficult to reach at the same time.

4.2.3.2 Activities

The activities for the scoping process are shown in the UML activity diagram of Figure 4.7.

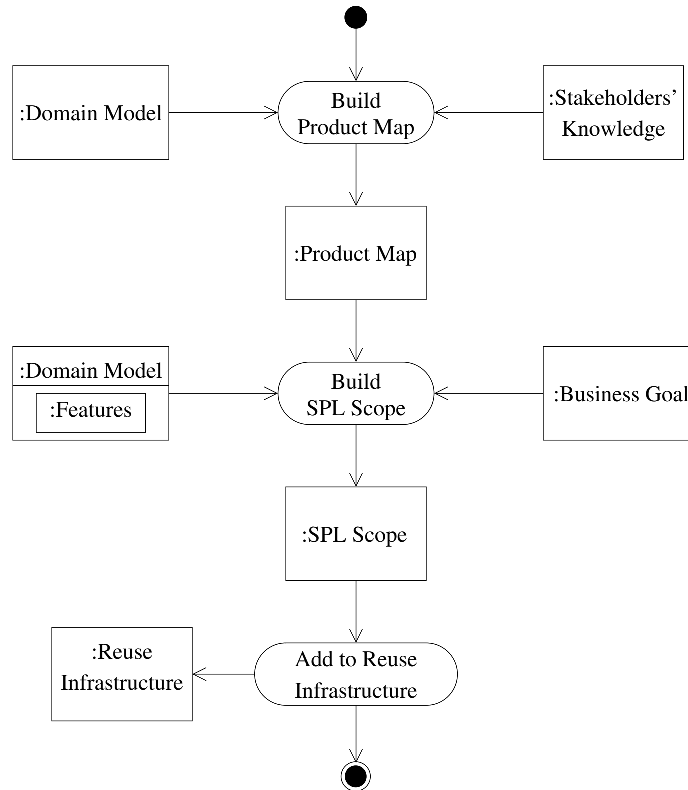


Figure 4.7: Scoping Construction Process

The first activity is building the product map; it is carried out taking into account the stakeholders' knowledge and the domain model artifacts produced during the process described in Section 4.2.1.3.

The main artifacts considered from the domain model are the feature model and the scenarios. The features are incorporated in the product map for each SPL product. Any given product is defined completely by its features. The scenarios illustrate the diverse relationships among features, and the global behavior of each product. Additionally, the goals contextualize the products.

Once the product map is finished, the scope is completely defined, because the other two artifacts that compose the scope, i.e., the feature model and the business goal, have also been already developed.

4.2.4 Related Work

This section discusses several techniques for domain analysis in general and meshing tool domains, as well as other approaches that have been used to build meshing tools.

Coplien et al. [57] propose SCV (Scope, Commonalities and Variabilities), a method for conceptually addressing domain analysis within SPL. FAST [246], FORM [116] and PuLSE [24] are notations and techniques proposed for the application of SCV to any application domain and generally cover the whole domain engineering stage. All these methods propose well-defined processes for building the domain model.

This thesis's approach goes a step further by formalizing the domain model definition and also precisely defining iteration/termination conditions for the proposed process. Furthermore, it does not need to tailor the approach for the specific application domain because it has been specifically created for the meshing tool domain.

Smith and Chen [212] have applied SCV to the meshing tool domain using FAST. Even though their approach is systematic, they do not take full advantage of meshing tool domain characteristics because they apply a general DA method for scientific computing software [211, 214]. In particular, binding time for variabilities in meshing tools is fixed: which features are included is always decided at product design time, and which particular implementation for each chosen feature is decided at compilation time. This default binding time leads to a simpler DA process and more compact documentation, and also allows making decisions at a higher level of abstraction, thus yielding simpler tools with the potential for improved performance.

Kim et al. [119] propose a DA method based on goals and scenarios. It involves four information levels: business, service, interaction and internal, each of them refining the previous one. Due to the number of information levels, this method is appropriate for characterizing domains where the domain expert has little experience in software development. Meshing tool experts are usually involved in software development even though they are not always knowledgeable in software engineering, so it is possible to simplify the domain model and the process for building it. The process proposed in this thesis includes the business goal as a driver for identifying domain model artifacts of the SPL and defines a single level where the complete domain model is defined. Park et al. [167] propose using features, scenarios and

goals for capturing the characteristics of the domain. In both cases, as their approach is general for any domain, their method involves three successive specification levels. Even though the methods of Kim et al. [119] and Park et al. [167] are iterative, they do not define a clear termination condition so as to determine whether the desired domain elements' quantity and quality have been reached.

Niemelä and Immonen [154] introduce the QRF (Quality Requirements of a software Family) method, which explicitly focuses on how quality requirements have to be defined, represented and transformed into architectural models. This method is appropriately structured and each step is clearly defined. Even though this method can be used to define the meshing tool domain model, it needs tailoring because it was proposed for a software distribution domain. Moreover, it requires the existence of concrete products, and in the case of the meshing tool domain this condition has been identified as optional. Finally, there is no clear distinction between the problem space and the solution space, due to its focus on transforming requirements to architectures. This can represent a risk for stakeholders lacking previous experience in the task of capturing quality requirements.

Douta et al. [79] present an approach to commonality and variability analysis called CompAS for the specific domain of computer-assisted orthopaedic surgery. This approach focuses in the analysis of the evolution of the domain to effectively determine which features should be included as common or variable. The method bases its source of information for building the domain model on publicly available literature (e.g. books, articles and standards). CompAS suggests that the domain analyst should regularly ask the domain expert for a correct, consistent and complete functional decomposition. The approach of this thesis assigns to the domain expert a relevant role as part of the process of building the domain model. However, it agrees with Douta et al. [79] on the convenience of counting on a domain specific process.

Smith and Yu [215] provide a document-driven approach for generating a family of parallel meshing tools. The information used is similar to that included in this thesis's model, but the process they present is mainly a waterfall. In contrast, the process presented in the thesis is iterative and the model is built incrementally so that feedback can be systematically incorporated.

Smith and Chen [213] researched meshing tool requirements with a SPL perspective, but no procedure is provided for using the products of this method for actually building meshing tools. Also, Bastarrica et al. [20] propose a product line architecture for the meshing tool domain, but do not focus on a systematic DA method.

4.3 Contributions of this Chapter

In this chapter, a complete process with activities, roles, input and output artifacts, and clear termination conditions was presented for domain analysis. The process was divided in two subprocess: one for building the domain model and an other for building the domain scope.

Both were modeled using UML activity diagrams.

The domain model was defined using not only feature models, the most widely accepted model for representing the common and variable aspects of the domain, but also considering the inclusion of other elements such as goals, scenarios, actions and a lexicon, which provide the support and rationale for discovering the features. Even though the inclusion of goals, scenarios and actions in the domain model and their relationships have been used in other domains, they were never used in the meshing tool domain. Furthermore, the inclusion of the lexicon in the domain model is a novelty.

Several formal languages have been used to represent feature models. However, the relationships among all elements of the domain model have rarely been formalized. In this work, these relationships are formalized using Z language. Thus, the domain analyst can iterate through several inconsistent domain models, using the formal consistency conditions, until he verifies that the domain model satisfies all consistency conditions. This iterative process can lead to domain model completeness if the captured domain model elements are sufficient for building all expected products.

On the other hand, the domain scope is built using the domain model produced in the previous process plus the identified business goal and the product map. Even though the business goal exists as an important concept in the domain, it is not always considered explicitly as a part of the scope. In the approach presented in this thesis, this goal is the first element that helps limit the domain.

The product map is built considering the feature model and the existing products or those to be produced. This is a big difference with respect to other approaches for scope definition, because in general they are *product oriented*, i.e., they need the existence of several products for defining the scope, instead of the approach presented here that is *feature oriented*, i.e., it needs the definition of features of the products, currently existing or not.

Finally, the product map considers optional extra-functional features such as cost, time and development resources, usually known but not generally explicitly used. These are typically found in the feature model instead of the product map. The change aims to maintain the documentation and the feature model manageable.

Chapter 5

Applying the Domain Analysis Method

In the following sections, an application example of the domain analysis construction process described in Chapter 4 is presented. All activities are covered and all outputs of each activity are produced. Specific artifacts of both domain model and scope are described, and a consistency and completeness analysis is presented. In particular, the completeness analysis considers two real meshing tools. Finally, the product map is filled in the scoping, taking into account some existing meshing tools.

5.1 Meshing Tool Domain Model

In this section, the proposed process for building the meshing tool domain model[1] is applied. First, the main domain concepts are introduced as part of the lexicon. This section focuses on the business goal, defining particular goals, scenarios and features. Several features are defined as terms of the lexicon. The relationships between goals, scenarios and features are established. Then, the process continues with a second iteration mainly because the feature model is found to be incomplete and inconsistent. Thus, the process improves the feature model, adding more features, defining them in the lexicon, and also relating them according to *Requires* and *Excludes*.

The relationships among goals, features and scenarios are stated in tables to aid consistency checks. Only when the domain experts and the domain analysts intuitively think that the model is ready, they proceed to check for termination conditions. Thus the domain analysts check for domain model consistency using the constraints established in the *ConsistentDomainModel* schema, and the domain experts check for completeness by determining if

[1]An early version of the domain model was validated in November 2007 by Todd L. Veldhuizen from the Electrical and Computer Engineering Department and R. Bruce Simpson from the Cheriton School of Computer Science, both from University of Waterloo.

the SPL's candidate tools can be built with the documented elements.

The domain model is restricted to meshing tools used in applications that require mainly unstructured meshes for representing the domain to be discretized. Unstructured meshes need very sophisticated algorithms and data structures for their generation and manipulation, and there are several generation methods. The modeling process will also assume that the geometry of the domain is already given as an input and that the output of the meshing tool is the discretization of the domain with the respective physical values, if they are given as part of the input. The problems involved in geometry modelers and numerical solvers that usually work in cooperation with the meshing tools are out of the scope of this thesis.

5.1.1 Lexicon

The lexicon includes the definition of the terms that are identified as essential within the application domain. These definitions allow the stakeholders to understand the basic concepts and share a common language while building and/or using a product.

This section describes part of the vocabulary used in the meshing tool domain. Even though it is not exhaustive, it makes it easier to understand the domain, and the knowledge it contains helps delimit the scope of the SPL.

Mesh: A mesh is a discretization of a domain geometry into simple cells. A mesh is composed of vertices (nodes) defined by points, edges defined by segments, faces (in 3D) defined by triangles or quadrilaterals, elements defined by triangles and/or quadrilaterals (2D elements) and tetrahedra and/or hexahedra (3D elements). An element represents the smallest discretization unit and it is of the same dimension as the mesh, i.e., either 2D or 3D. Meshes are usually classified into structured meshes and unstructured meshes. Structured meshes are composed by elements that are similar in shape and size, while unstructured meshes are composed by elements of different size and/or shape.

Meshing Tool: It is a piece of software for generating and managing meshes. In general, meshes are easy to generate but their quality strongly degrades as domain geometry complexity increases, as point density can vary in a very irregular way in different regions of the mesh. Automatically generating these meshes usually requires designing and implementing complex algorithms and data structures [78].

Geometry: It is the description of the domain shape. There are different methods for specifying the domain geometry: boundary representation (b-rep), constructive solid geometry (CSG) and a domain discretization representation (dd-rep). A b-rep describes the boundary of the domain, e.g., a closed set of contiguous segments (simple polygon in 2D), or a closed set of non-overlapping polygons (simple polyhedron in 3D). A CSG is defined by the composition of a set of predefined primitives, e.g., circles and rectangles in 2D, or spheres and hexahedra in 3D. A dd-rep represents the domain by using a set of simple non-overlapping elements. A mesh is a kind of dd-representation.

(Quality) Criterion : This concept represents how good the user expects the final mesh to be. For example, if the quality criterion is Minimum Angle equal to 30, all the elements of a mesh (triangles o quadrilaterals) should have a minimum angle greater or equal to 30 degrees.

Refinement criterion: These are criteria that control parameters such as the number of points inserted in a certain process, or the size of the mesh elements, such as Maximum Edge Length (all the edges must have a length less or equal than a threshold value) or Maximum Area (all the elements must have an area less or equal than a threshold value).

Improvement criterion: These are criteria that control the quality of the mesh elements (usually the more equilateral an element the better). Two example criteria are: Minimum Angle (the minimum angle of the whole mesh must be greater or equal to a threshold value), and Maximum Aspect Ratio (the ratio between the longest and shortest edges length of the element must be less than or equal to a threshold value).

Optimization criterion: This type of criterion is used by optimization algorithms. These algorithms only move the mesh points so that an optimized mesh can be achieved.

Derefinement criterion: This type of criterion allows the user to eliminate elements of the mesh when the mesh is too dense. For example, triangles whose area is too small can be eliminated by specifying a Minimum Area criterion.

Algorithm: The meshing tools users usually input a domain geometry, some physical values associated to this geometry, and some quality criteria, in order to get an appropriate mesh to simulate some phenomena that occurs in this domain as output. In order to generate an adequate mesh, almost all meshing tools require one or more of the following algorithms:

Generate initial mesh: The meshing tool takes a domain geometry as input and generates a discretization that represents the geometry as precisely as possible. It may have the same number of final points as the input geometry or a different one.

Refinement: This process divides coarse elements into smaller ones according to certain refinement criteria. For example, if the Maximum Edge Length criterion is specified, the chosen algorithm divides (refines) all the elements whose longest edge is larger than the specified Maximum Edge Length value.

Improvement: This process improves the quality of the mesh by applying a given algorithm, which in turn may require the insertion of additional points. For example, to improve the mesh in order to fulfill the Delaunay condition (Delaunay algorithm), the minimum angle of the mesh is increased without inserting new points. However if the Voronoi point insertion [190] algorithm is applied, new points are inserted.

Optimization: This process improves mesh quality without inserting new points, simply by moving existing mesh points in an adequate way so as to improve the quality of the resulting elements according to some optimization criteria.

Derefinement: This process guides the generation of a coarser mesh in regions with too many mesh points. For examples, if the Minimum Edge Length criterion is specified, all edges whose length is less than the Minimum Edge Length value are eliminated.

Move boundary: All the previous algorithms work over fixed geometries, i.e., the shape of the domain does not change after applying any of these algorithms. However, there are applications where the geometry may change; for example, while modeling a growing tree. Thus, this concept refers to algorithms that manage geometry changes.

Evaluate: Generates statistics of the mesh. For example number points, edges and faces, size of the edges, area of the triangles, and minimum angle of each triangle, among others.

Postprocess: Represents additional information that is generated depending, for example, on the numerical method that will be used. Also included here is the generation of a quadrilateral mesh from a triangulation and viceversa.

Visualize: Represents tools that allow to visualize a mesh. This feature can be integrated in the mesh generator.

Input: Represents the geometry and physical values of the domain to be discretized.

Output: Represents the mesh that the mesh generator has built according to the specified input and criteria specified by the user.

Region: Represents the part of the domain where an algorithm will be applied.

User interface: The interaction between a user and a software system is done by means of the user interface (UI). In this sense, the UI provides both input and output facilities. The first one permits entering information or manipulating the software system, and the second one permits obtaining results of those inputs. According to Sommerville [217], there are at least four primary styles of interaction with the UI: direct manipulation, menu selection, form fill-in and command language.

Command Language: The user puts a special command and associated parameters to indicate the system what to do.

Menu selection: The user selects a command from a list of possibilities. This list is commonly represented by a menu.

Direct manipulation: The user interacts directly with objects on the screen. This style usually involves a pointing device such as a mouse that indicates the object to be manipulated and the action to be done with that object.

Form fill-in: The user fills in the fields of a form.

5.1.2 Goals

In this thesis, the business goal and a set of particular goals for this domain are identified.

5.1.2.1 Business goal

The business goal, i.e., the goal for developing the meshing tool SPL, can be stated as follows:

- Developing new meshing tools for different applications with minimum effort.

This business goal recognizes the complexity of building meshing tools in general, and supports the application of the SPL approach for their construction.

5.1.2.2 Goals

Goals define what stakeholders want to achieve through the software product line. This work has identified several particular goals for the Meshing Tool domain. Here, six goals that implicitly describe both the context and the quality attributes of the software product line it would like to develop are presented.

G1 : Generation of meshes for diverse domain geometries that fulfill certain given criteria in specific regions of the domain depending on the particular application requirements.

G2 : Generation of meshes with the minimum amount of points that fulfill the application requirements.

G3 : Generation of meshes in a reasonable CPU time.

G4 : Generation of meshes using an efficient memory management.

G5 : Scalability in the number of mesh points.

G6 : Generation of meshes that fulfill the requirements of a particular numerical method.

5.1.3 Scenarios and actions

This section details a list of use scenarios for particular meshing tools as well as their corresponding sequences of actions. Notice that actions that take part in different scenarios have the same identification whenever they are identical. Even though a 2D domain is considered, the scenarios can also be applied to a 3D sub-domain, taking note of particular differences, for instance, types of algorithms that work in only one dimension but not in the other one.

S1 : Generate quality Delaunay meshes for piecewise linear complex (PLC) or planar straight line graph (PSLG) domains.

- **A1** : Apply an algorithm for reading the PLC or PSLG geometry in the corresponding format.
- **A2** : Apply the Delaunay algorithm in order to generate the initial mesh.
- **A3** : Select refinement and/or improvement criteria, and regions where they will be applied.
- **A4** : Apply the Voronoi Point Insertion algorithm using the specified quality criteria and regions.
- **A5** : If desired, evaluate the quality of the mesh elements.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S2 : Generate quality meshes with the minimum number of final mesh points.

- **A8** : Apply an algorithm for reading the geometry in the corresponding format.
- **A9** : Apply an algorithm to generate the initial mesh.
- **A10** : Select refinement, improvement and/or optimization criteria, and regions where they will be applied.
- **A11** : Apply the refinement, improvement and/or optimization algorithm that minimizes the number of inserted points using the specified quality criteria and regions.
- **A12** : If necessary, apply a derefinement algorithm using the specified derefinement criterion and region.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S3 : Generate meshes with approximated quality as fast as possible.

- **A8** : Apply an algorithm for reading the geometry in the corresponding format.
- **A9** : Apply an algorithm to generate the initial mesh.
- **A13** : Select improvement criteria and regions where they will be applied.
- **A14** : Apply the fastest improvement approximated algorithm using the specified quality criteria and regions.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S4 : Generate meshes with minimal quality that optimize the memory used.

- **A8** : Apply an algorithm for reading the geometry in the corresponding format.
- **A9** : Apply an algorithm to generate the initial mesh.
- **A3** : Select refinement and/or improvement criteria and regions where they will be applied.
- **A15** : Apply a memory efficient refinement and/or improvement algorithm using the specified quality criteria and regions.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S5 : Generate large meshes in a reasonable CPU time.

- **A8** : Apply an algorithm for reading the geometry in the corresponding format.
- **A9** : Apply an algorithm to generate the initial mesh.
- **A16** : Select refinement criteria and a region where they will be applied.
- **A17** : Apply a refinement algorithm that scales with the number of points using the specified criteria and regions.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S6 : Generate meshes for a numerical method that requires specific information (postprocess).

- **A18** : Read an already generated mesh.
- **A19** : Store the mesh in its internal representation.
- **A20** : Apply postprocessing to the mesh.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S7 : Evaluate meshes.

- **A18** : Read an already generated mesh.
- **A19** : Store the mesh in its internal representation.
- **A21** : Evaluate the quality of the mesh.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

S8 : Adapt and improve the quality of an already generated mesh so that its minimum angle is less than a certain value in the whole mesh.

- **A18** : Read an already generated mesh.
- **A19** : Store the mesh in its internal representation.
- **A22** : Select the minimum angle criterion and a region where it will be applied.
- **A23** : Apply the Lepp-Delaunay algorithm using the specified quality criterion and region.
- **A5** : If desired, evaluate the quality of the mesh elements.
- **A6** : Store the mesh in a specified output format.
- **A7** : Visualize the mesh.

5.1.4 Features

Features are obtained both from the concepts defined as part of the lexicon and from actions that are required for carrying out the identified scenarios. These features are organized according to a feature model, identifying commonalities and variabilities. In this process, new features may also be identified and added to the model. Once the feature model is built, the *Requires* and *Excludes* relationships between features are established.

5.1.4.1 Feature diagram

Figures 5.1, 5.2, 5.3 and 5.4 show the feature model. The feature constraints are detailed in Section 5.1.4.2. According to Figure 5.1, the features **User Interface**, **Input**, **Output** and **Mesh** are common to any meshing tool. **Region**, **Generate initial mesh**, **Algorithms**, **Evaluate**, **Criterion**, **Move boundary**, **Postprocess** and **Visualize** are optionals, i.e., they may or may not be present in a particular meshing tool. Moreover, any subset or all of the interaction styles for user interfaces **Command language**, **Menu selection**, **Direct manipulation** and **Form fill-in** may be present in a particular tool. Finally, a meshing tool must work either with meshes in two dimensions (**2D Mesh**) or three dimensions (**3D Mesh**).

Even though meshes of different dimensions are never mixed in a particular product, the feature model as shown captures the knowledge of the whole meshing tool domain, making it reusable for building meshing tools of different dimensions. Figures 5.2, 5.3 and 5.4 refine some of the features in Figure 5.1.

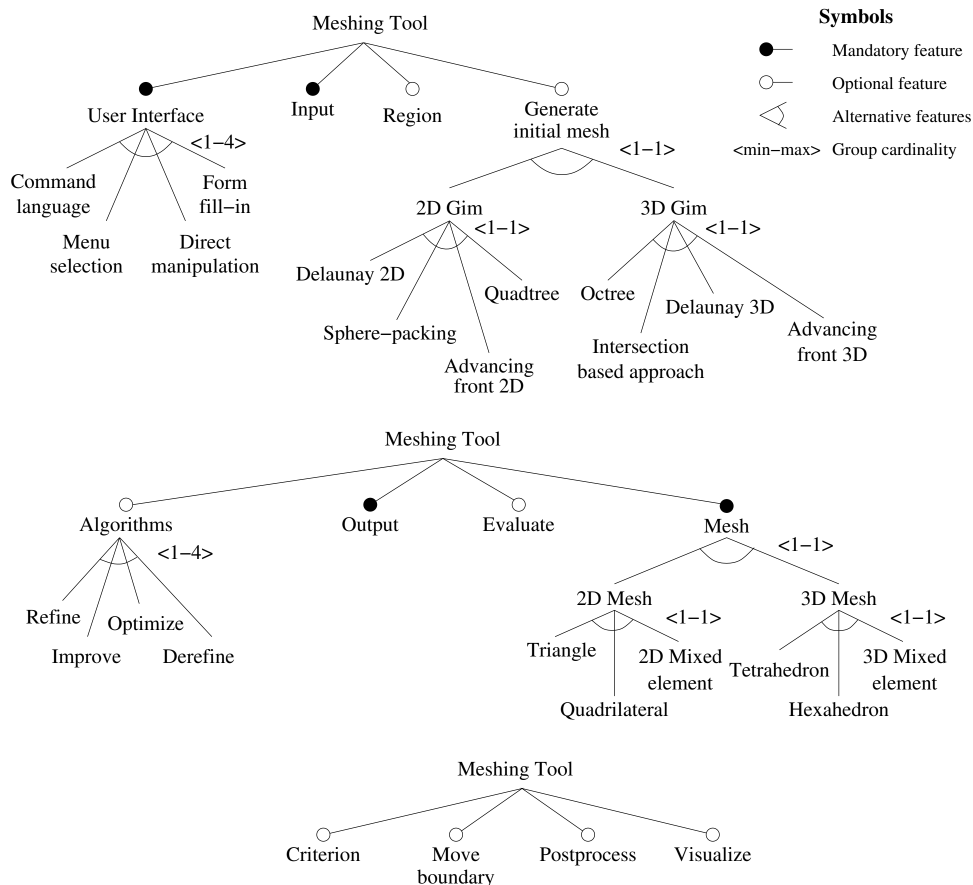


Figure 5.1: A Meshing Tool Domain Feature Diagram

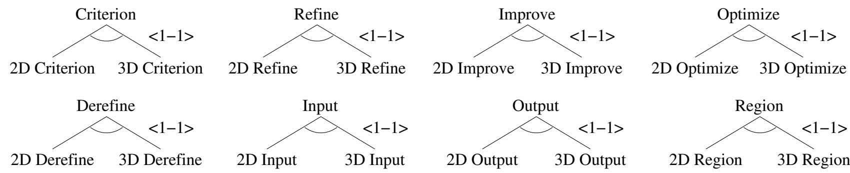


Figure 5.2: A Meshing Tool Domain Feature Diagram, continuation

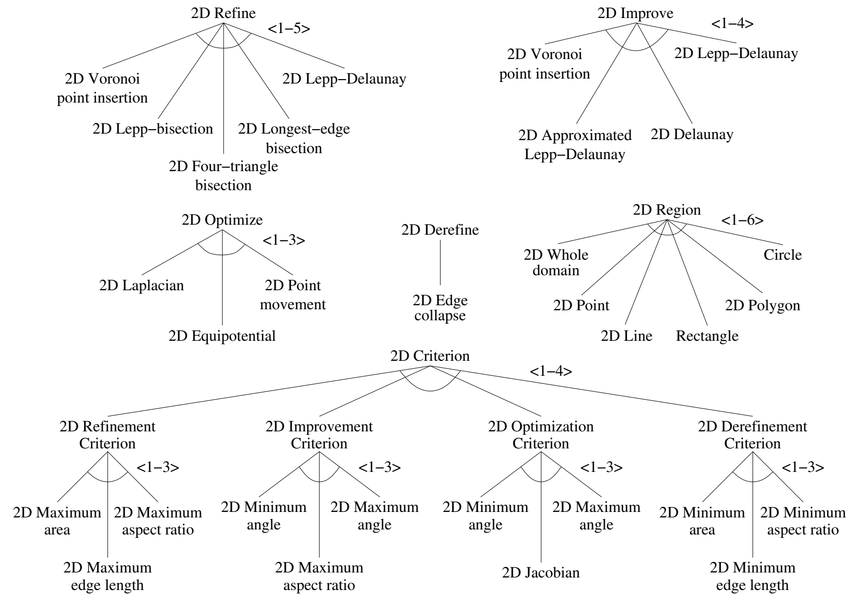


Figure 5.3: A Meshing Tool Domain Feature Diagram, continuation

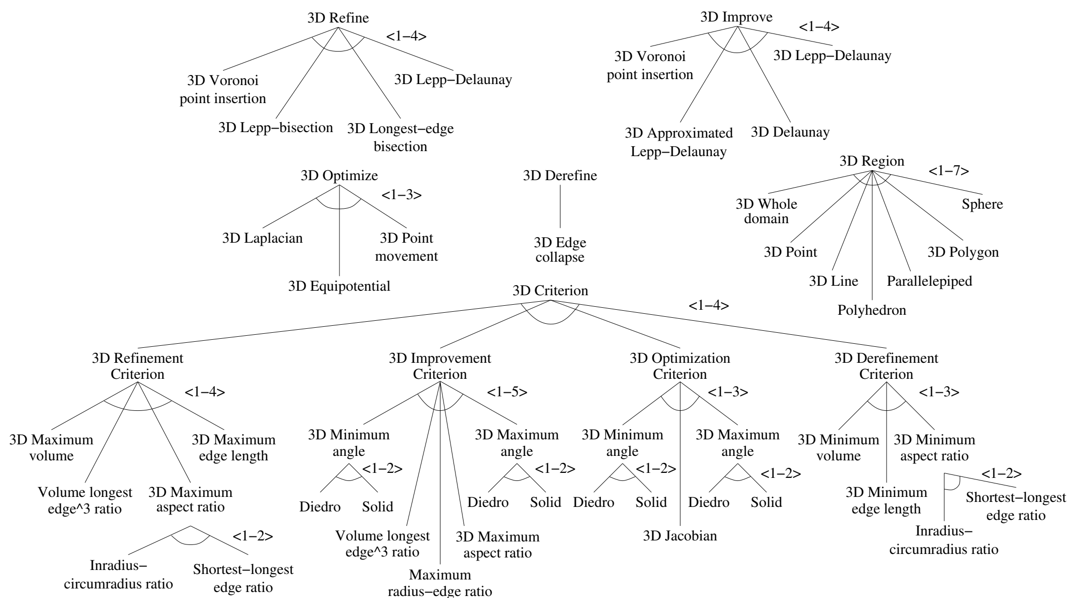


Figure 5.4: A Meshing Tool Domain Feature Diagram, continuation

5.1.4.2 Feature constraints

Feature models sometimes must contain additional constraints [60] or composition rules [115] that cannot be expressed only with solitary or grouped features in a plain feature diagram. These constrains are organized in Table 5.1. It shows some of the identified *Requires* and *Excludes* relationships. When managing **2D Refine** algorithms it is necessary to deal with **2D Mesh**es, and this is established with a *Requires* relationship. *Excludes* constraints are related to combinations that are meaningless, and so they must not be considered simultaneously for a particular product. For example, a refinement algorithm such as **2D Longest-edge bisection** that works in two dimensions needs a mesh of **Triangle**s. On the other hand, a refinement algorithm such as **2D Lepp-bisection** is excluded when dealing with regions that have **Rectangle**s.

Feature	Constraint	Feature
2D Mesh	REQUIRES	2D Gim
2D Refine	REQUIRES	2D Mesh
2D Derefine	REQUIRES	2D Region
2D Improve	REQUIRES	2D Criterion
2D Voronoi point insertion	REQUIRES	Triangle
2D Lepp-bisection	REQUIRES	Triangle
2D Four-triangle bisection	REQUIRES	Triangle
2D Longest-edge bisection	REQUIRES	Triangle
2D Lepp-Delaunay	REQUIRES	Triangle
3D Voronoi point insertion	REQUIRES	Tetrahedron
3D Lepp-bisection	REQUIRES	Tetrahedron
3D Longest-edge bisection	REQUIRES	Tetrahedron
3D Lepp-Delaunay	REQUIRES	Tetrahedron
2D Lepp-bisection	EXCLUDES	Rectangle
Move boundary	EXCLUDES	Criterion
Move boundary	EXCLUDES	Region
Postprocess	EXCLUDES	Criterion
Postprocess	EXCLUDES	Region
Quadtree	EXCLUDES	Triangle
Octree	EXCLUDES	Tetrahedron

Table 5.1: Constraints among features

Even though it is possible to find other constraints between features, such as "composed of", "implemented by" and "generalization/specialization", presented by Lee et al. [129], this thesis only uses *Requires* and *Excludes* [63, 115] because they are easily applied to the Meshing Tool domain, mainly because these products' features are functionalities, parameters, data storages and user interactions, and the mentioned relationships are established naturally.

5.1.5 Domain Model Consistency

Table 5.2 shows the relationships between goals and scenarios (*By-Scenario* relationship [a] in the *DomainModel* schema). The table shows that all goals are achieved by at least one scenario, and that all scenarios contribute to the achievement of at least one goal, fulfilling condition [q] in *ConsistentDomainModel*.

Scenario	Goals
S1	G1, G2, G6
S2	G1, G2, G5, G6
S3	G1, G2, G3, G6
S4	G1, G2, G4
S5	G1, G3, G4, G5, G6
S6	G6
S7	G1, G2, G6
S8	G1, G6

Table 5.2: Relationships between Scenarios and Goals

Table 5.3 (relationship *Attached* [o]) shows that all actions are satisfied by features in the feature model, fulfilling condition [s]. All features identified as required for fulfilling the actions are already part of the feature model (condition [t]). By construction, the set of actions is the union of all actions required for fulfilling the specified scenarios (condition [e]). Thus, the domain model is consistent.

For simplicity, Table 5.3 shows only 2D features. As the scenarios and actions in Section 5.1.3 show, 3D features can also be considered. If this is the case, Table 5.3 can be easily be modified to include these features.

Action	Features
A1	2D Input
A2	2D Delaunay, 2D Mesh
A3	2D Region, 2D Refinement criterion, 2D Improvement criterion
A4	2D Voronoi point insertion, 2D Refinement criterion, 2D Improvement criterion, 2D Region, 2D Mesh
A5	Evaluate, 2D Mesh
A6	2D Output, 2D Mesh
A7	Visualize, 2D Mesh
A8	2D Input
A9	2D Gim, 2D Mesh
A10	2D Region, 2D Refinement criterion, 2D Improvement criterion, 2D Optimization criterion
A11	2D Refine, 2D Improve, 2D Optimize, 2D Criterion, 2D Region, 2D Mesh
A12	2D Derefine, 2D Derefinement criterion, 2D Region, 2D Mesh
A13	2D Region, 2D Improvement criterion
A14	2D Approximate Lepp-Delaunay, 2D Improvement criterion, 2D Region, 2D Mesh
A15	2D Refine, 2D Improve, 2D Criterion, 2D Region, 2D Mesh
A16	2D Region, 2D Refinement criterion
A17	2D Refine, 2D Criterion, 2D Region, 2D Mesh
A18	2D Input, 2D Mesh
A19	2D Mesh
A20	Postprocess, 2D Mesh
A21	Evaluate, 2D Mesh
A22	2D Minimum angle, 2D Whole domain
A23	2D Lepp-Delaunay, 2D Minimum angle, 2D Whole domain, 2D Mesh

Table 5.3: Relationship between Actions and Features

5.1.6 Domain Model Completeness

In order to check for completeness, a domain expert checked if two meshing tools, *Triangle* [206] and *TetGen* [208], could be built with the elements specified in the domain model. It is assumed that these two tools are within the SPL scope, and as such it should be possible to build them with the elements identified as part of the domain model of this SPL.

This is a strong assumption, because a SPL exists in a specific application domain and all possible products to be built with the SPL must share preimplemented components (or source code), and such thing is not necessarily true for *Triangle* and *TetGen*. However, as an exercise and with the goal of evaluating the domain model completeness, it is possible to think that these applications are within the scope of the same SPL defined in this thesis. It must be noted that the number of applications needed to evaluate completeness is not related to the break-even point at which SPL is less expensive than single system development, i.e., approximately three systems (applications), which is shown in Figure 2.3. Completeness is checked by determining if the SPL's candidate tools can be built with the specified domain model elements. If there are several preexisting products in the domain of interest, the completeness check must consider all of them. On the other hand, if the stakeholders have already considered planned and potential products, they must also be taken in to account during the completeness checking activity. The approach presented in this thesis deals with both these cases and combinations thereof.

All features are highlighted and those that are not present in the feature diagram yet (Figures 5.1 to 5.4) also have an * attached.

5.1.6.1 Triangle

Triangle is a well known open source 2D mesh generator that allows the user to generate quality 2D triangulations. As input it can read either a **Geometry*** defined by a boundary representation (**b-rep***) or an already generated mesh (**dd-rep***). In order to **Generate initial mesh**, it provides two variations: the **Constrained Delaunay*** algorithm and the **Conforming Delaunay*** algorithm. The **Mesh** is formed by **Triangle**, **Edge*** and **Vertex***. As **Criterion**, it provides two alternatives: **Maximum area** and **Minimum angle**. As **Refine**, it provides the **Voronoi point insertion** algorithm to generate refined meshes. The same algorithm can be used as an implementation of **Improve**. The **Region** is always the **Whole domain**. There are two algorithms available for **Postprocess**: one for **Generating and storing the Voronoi diagram*** of the generated triangulation, and another for **Check mesh consistency***. As **Format*** of **Output**, it can be the **Triangle*** format (.node and .ele), or the **Object File Format***. The first one requires that the **Visualize** be **ShowMe*** and the second one requires **Geomview***.

The related scenarios are: S1, S3, S4, S5, S6, S8.

The goals are: G1, G2, G3, G4, G5, G6.

5.1.6.2 TetGen

TetGen is a 3D meshing tool developed by Hang Si, at the research group of Numerical Mathematics and Scientific Computing at Weierstrass Institute in Germany. This tool allows generating good quality and adaptive tetrahedral meshes suitable for numerical methods, such as finite element or finite volume methods. The tool has a **User Interface** driven by a **Command language** that permits doing all actions provided by the tool. The domain **Geometry*** can be specified by using a simple **b-rep*** or a piecewise linear complex **PLC***. The **Mesh** is formed by several **Tetrahedron**, **Triangle**, **Edge***, and/or **Vertex***. There are several available input and output **Format***s: the **Object File Format***, the **TetGen Format*** (.node, .face and .ele), and the **Plyhedral Format***, among others. For **Generate initial mesh**, there is a **Conforming Delaunay*** algorithm. To **Refine** and **Improve** the **Whole domain** of the **Mesh**, the **Voronoi point insertion** algorithm is implemented. As **Refinement Criterion**, **Maximum volume** is considered. Furthermore, as **Improvement Criterion**, **Maximum radius-edge ratio** is considered. The **Postprocess** is done through the **Generating and storing the Voronoi diagram*** algorithm. Finally, it is also possible to **Evaluate** and **Visualize** the mesh whenever the user requires it. Specifically for visualization, both **Geomview*** and **TetView*** are used.

The related scenarios are: S1, S5, S6, S7.

The related goals are: G1, G2, G5, G6.

5.1.6.3 Domain Model Completeness Evaluation

From the previous paragraphs, it can be seen that the domain model presented in Section 5.1.4 is incomplete, and missing all those features having an * attached. However, the iterative nature and structured notation of the proposed DA process allows the missing features to be easily added to the feature model, and the scenarios and goals can be adjusted accordingly.

Figure 5.5 shows the result of adding sub-features to the existing feature model as a result of this completeness evaluation, For example, three new subfeatures are added to the **Visualize** feature: **ShowMe**, **TetView** and **Geomview**. Similarly for 2D and 3D, the **Object File Format**, and **Triangle Format** subfeatures are added to the **Format** feature, and the **Plyhedral Format** subfeature is added only in the 3D case. Likewise, the corresponding subfeatures for the **Postprocess** feature are added as well. Other features, such as **Node**, **Vertex** and **Physical value** for the **Mesh** feature, are not present and can be incorporated in a next version of the feature model. This iterative process continues until the completeness condition is satisfied.

According to the process described, adding these new features forces a new iteration of the process. The inclusion of new elements in the domain model can affect the model's consistency, as these new features may need new scenarios and goals. As an example, for the *Triangle* tool, adding new subfeatures to the **Visualize** feature requires a variation **S1'**

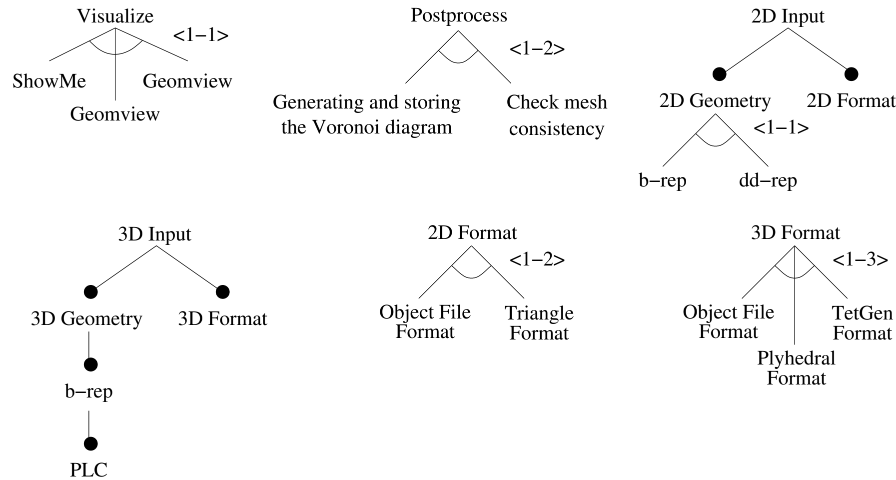


Figure 5.5: Added features according to completeness evaluation

of the scenario **S1** which in turn incorporates several other new actions (**A1'**, **A3'**, **A7'**), as shown below.

S1' : Generate quality Delaunay meshes for piecewise linear complex (PLC) domains.

- **A1'** : Apply an algorithm for reading the PLC geometry in the Triangle format.
- **A2** : Apply the Conforming Delaunay algorithm in order to generate the initial mesh.
- **A3'** : Select the Minimum angle criterion, and the Whole domain region where it will be applied.
- **A4** : Apply the Voronoi Point Insertion algorithm using the specified quality criteria and regions.
- **A5** : If desired, evaluate the quality of the mesh elements.
- **A6** : Store the mesh in a specified output format.
- **A7'** : Visualize the mesh using ShowMe.

Notice that the description for **S1'** is identical to that of **S1**, but they are still different since they have a different sequence of actions. In the case of *TetGen*, something similar occurs.

Even though reaching a domain model that is both complete and consistent may seem to be an endless task, the completeness and consistency conditions shown in Section 4.2.2 indicate where to concentrate the model construction efforts.

5.2 Meshing Tool Scope

For building the scope, it is necessary to count on some artifacts that form the domain model. According to Figure 4.7, a complete domain model is necessary for producing the product

map, and the feature model and the business goal are necessary for producing the scope. Furthermore, as it was described in Section 4.2.3.2, it is useful to consider the goals and scenarios. They contextualize and show the global behavior of each product.

As an example, tables 5.4 and 5.5 present the Meshing tool product map. These tables do not include the optional extra-functional features column for time and development cost present in Table 4.1. Priority is ranked from 1 (lowest priority) to 5 (highest priority).

Tables 5.4 and 5.5 present meshing tool products taken from the state of the practice in the meshing tool area, and also include some products that have not been created yet. In particular, products **P1** to **P6** were identified from Bastarrica and Hitschfeld-Kahler [18]. Products **P7** to **P12** were identified from Contreras [55] and Contreras et al. [56].

The product map's main assumptions are two: that the feature model presented in previous sections is complete, and that products **P1** to **P12** are in the SPL scope.

Considering the Tables 5.4 and 5.5 and other antecedents, it is possible to affirm:

- Such as it was mentioned in previous paragraphs, products P1 to P3 and P7 to P9 exist. In particular, products P1 and P7 correspond to characteristics of *Triangle* [206] and *TetGen* [208], respectively. The rest of the products have been developed by Hitschfeld-Kahler's research group and mentioned in Bastarrica and Hitschfeld-Kahler [18], but no open source code has been released and no tool was sold. They were all developed with academic purposes.

- For the 2D sub-domain, several features have only been considered for the **Planned** and **Potential** products in Table 5.4, such as *2D Optimization Criterion*, *2D Optimize* algorithms, and *2D Region* different from *2D Whole domain*. This means that those features are not "popular" among existing products but are necessary for planned and potential products.

- In general, mesh users can be divided in two big groups: those who use quadrilateral or hexahedron meshes, and those who use triangle or tetrahedron meshes. The products presented in both tables are focused on the second group. Furthermore, there are based on the open source code of the *Triangle* and *TetGen* mesh generators. The products specified in the tables can be used for modeling or simulation with finite element methods or finite volume methods, with the addition of some postprocessing[2].

- By taking into account the users' needs, several different meshing tools can be generated. It is not mandatory to consider all features in all products. For example, P2 is the product with the smallest number of features selected in the Table 5.4, and it is a totally operative product.

[2]Nancy Hitschfeld-Kahler. Personal communication, October 2012. On Mesh Elements.

5.3 Contributions of this Chapter

In this chapter, the processes for building both the domain model and the domain scope presented in Chapter 4 were applied completely and carefully. In this sense, every single element was built, including those in the domain model, such as feature model, goals, scenarios, actions, and lexicon, and those in the domain scope, such as business goal, feature model, and product map.

Thus, the practical applicability of the domain analysis process described in the previous chapter was shown. The formal modeling of the relationships among the different elements of the domain model allows verifying the consistency of the model. Furthermore, the completeness of the model can be checked taking existing meshing tools into account. The iterative nature of the domain analysis process permits easily adding additional elements to the domain model, in case of being necessary.

Finally, the rationale for building both the domain model and the domain scope is preserved explicitly, through the different domain elements, and the relationships shown in the previous chapter.

Sub-Domain	**Feature**	**Priority**	**Existing**			**Planned**	**Potential**	
			P1	P2	P3	P4	P5	P6
2D	User Interface	5	X	X	X	X	X	X
	Command Language	3	X		X	X		
	Menu Selection	5		X	X	X	X	X
	Direct Manipulation	1					X	X
	Form Fill-in	1					X	X
	2D Gim	3	X		X	X	X	X
	Delaunay 2D	3	X		X	X	X	X
	Algorithm	5			X	X	X	X
	2D Optimize	1					X	
	2D Laplacian	1					X	
	2D Refine	3	X		X	X		X
	2D Lepp-Delaunay	3			X	X		X
	2D Voronoi point insertion	3	X					
	2D Improve	3		X	X	X	X	X
	2D Lepp-Delaunay	3		X	X	X	X	X
	2D Voronoi point insertion	3	X			X		X
	2D Input	5	X	X	X	X	X	X
	2D Geometry	5	X	X	X	X	X	X
	b-rep	3	X		X			
	dd-rep	5	X	X	X	X	X	X
	2D Format	3		X	X	X	X	X
	Object File Format	3		X	X	X	X	X
	2D Output	5	X	X	X	X	X	X
	dd-rep	5	X	X	X	X	X	X
	2D Format	5	X	X	X	X	X	X
	Object File Format	5	X	X	X	X	X	X
	2D Criterion	3	X			X	X	X
	2D Optimization Criterion	1				X	X	X
	2D Minimum angle	1					X	X
	2D Maximum angle	1					X	X
	2D Refinement Criterion	3	X		X	X		X
	2D Maximum area	3	X					
	2D Maximum edge length	3			X	X		X
	2D Improvement Criterion	3		X	X	X	X	
	2D Minimum angle	3	X	X	X	X	X	
	2D Maximum angle	3		X	X	X	X	
	2D Region	3	X	X	X	X	X	X
	2D Whole domain	3	X	X	X	X	X	X
	2D Point	1				X		X
	2D Line	1				X		X
	Rectangle	1				X		X
	2D Circle	1				X		X
	Move Boundary	1					X	X
	Visualize	3	X		X	X	X	X
	Postprocess	1		X	X	X		X
	2D Mesh	5	X	X	X	X	X	X
	Triangle	5	X	X	X	X	X	X

Table 5.4: Product Map for 2D Meshing Tools

Sub-Domain	**Feature**	**Priority**	**Existing**			**Planned**	**Potential**	
			P7	P8	P9	P10	P11	P12
3D	User Interface	5	X	X	X	X	X	X
	Command Language	3	X	X	X			
	Menu Selection	3				X	X	X
	Direct Manipulation	1					X	X
	Form Fill-in	1						X
	3D Gim	1	X			X		X
	Delaunay 3D	1	X			X		X
	3D Algorithm	5	X	X		X	X	
	3D Refine	5	X	X		X	X	
	3D Voronoi point insertion	3	X	X			X	
	3D Lepp-bisection	3				X	X	
	3D Input	5	X	X	X	X	X	X
	3D Geometry	5	X	X	X	X	X	X
	b-rep	5	X		X	X		X
	dd-rep	5		X	X	X	X	X
	3D Format	5	X	X	X	X	X	X
	Object File Format	5	X	X	X	X	X	X
	3D Physical properties	3				X		X
	3D Output	5	X	X	X	X	X	X
	dd-rep	5	X	X	X	X	X	X
	3D Format	5	X	X	X	X	X	X
	Object File Format	5	X	X	X	X	X	X
	3D Physical properties	5				X		X
	3D Criterion	5	X	X	X	X	X	
	3D Refinement Criterion	3	X	X	X	X	X	
	3D Volume longest edgeˆ3 ratio	3			X		X	
	3D Radius-edge ratio	3	X	X	X	X	X	
	3D Region	5	X	X	X	X	X	X
	3D Whole domain	5	X	X	X	X	X	X
	Evaluate	1	X		X	X		
	Visualize	5	X	X	X	X	X	X
	3D Mesh	5	X	X	X	X	X	X
	Tetrahedron	5	X	X	X	X	X	X

Table 5.5: Product Map for 3D Meshing Tools

Chapter 6

Domain Design for the Meshing Tool Domain

Domain Design (DD) is the second stage in the domain engineering activity. The main purpose of DD is to develop an architecture for the SPL [58, 60]. To reach this objective, the Domain Design process takes its inputs from the Domain Analysis stage and the knowledge gained in that process.

Section 6.1 discusses the product architecture, both for single system and SPL.

Section 6.2 describes in more detail the domain design process and its activities.

6.1 Generalities

6.1.1 Product Architecture

There are several definitions for software architecture. For example, Medvidovic et al. [138] present an elaborate definition, which states that a software system's architecture is the set of principal design decisions about the system. These decisions encompass every aspect of the system under development, such as system structure, system functional and non-functional properties, and system interaction, as well as the system development process itself and the system's business position. For these authors, which design decisions must be included in the architecture is determined by a negotiation process among the system's stakeholders.

Software architecture

Bass et al. [17] define the software architecture of the program or computing system as the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. In this definition, the externally visible properties are those assumptions that other elements can make of an

element, such as its provided services, performance characteristics, fault handling, and shared resource usage, among others.

For the purpose of this thesis, Bass et al.'s definition is more appropriate as it takes into account the existence of multiple views that help understand and specify a system, and it also takes the existence of an software architecture for granted, even though it may not be designed explicitly.

Several authors [74, 77, 82, 203, 240] concur on the myriad benefits of having a correct and understandable architecture defined before the product development process. However, the architecture itself does not resolve all problems in the development life cycle [139]. For instance, the simple and explicit use of the architecture neither guarantees a quality software development process nor the quality of the final product.

Software architecture purpose

According to Niemelä and Immonen [154], the main goals of the software architecture are to provide an overview of the software structure and its components, to classify these components into generic and specific categories, to describe the responsibilities and contexts of components, and to safeguard the balance of business and technical issues.

For these authors, building the architecture is an essential activity: it provides a means to reason about and prioritize quality attributes, manage development team members, establish work divisions, map responsibilities to services/components and vice versa, map functional and quality requirements to components/services, and cluster the components to be developed.

The software architecture greatly influences the final quality of the system, because it can inhibit or enable its quality attributes [17, 82]. Thus, an important task when building the architecture is not only to design it so as to achieve the specified functionality, but also to design it so as to incorporate quality attributes into it.

6.1.1.1 Quality Attributes

A quality attribute is a property of a software product by which its quality will be judged by some stakeholder or stakeholders. Quality attribute requirements such as those for performance, security, modifiability, reliability, and usability have a significant influence on the software architecture of a system [201]. Then, it is important to understand how quality attributes interact with and constrain each other, and how they affect the achievement of other quality attributes in a particular software architecture [154].

Categories

Several authors [16, 76, 106] have defined and categorized quality attributes. For example, Bass et al. [16] categorize quality attributes into **observable via execution** or **operational** such as performance, security, availability, usability and functionality, and **not observable via execution** or **development attributes** such as modifiability, portability, reusability, integrability and testability. On the other hand, the ISO 9126-1 [106] standard defines a software quality model with six categories of characteristics: **functionality**, **reliability**,

usability, **efficiency**, **maintainability** and **portability**, where each one is also divided into subcategories.

Lists of quality attributes, with detailed descriptions of each one, can be found in Barbacci et al. [10], Bass et al. [17], Clements et al. [48], Dobrica and Niemelä [76], and ISO 9126-1 [106].

6.1.1.2 Viewtypes, styles and views

Depending on the complexity of the systems developed and the number of quality attributes to be considered, it can be necessary to use different software architecture views. Figure 6.1, structured from the classification of Clements et al. [48], shows the viewtypes and styles.

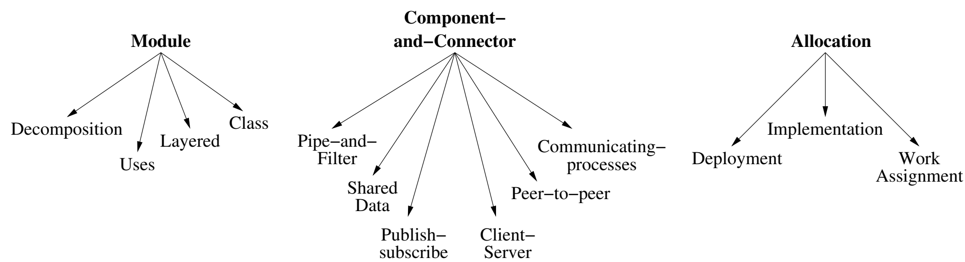


Figure 6.1: Viewtypes and Styles

The three viewtypes —module, component-and-connector, and allocation— correspond to three perspectives an architect must consider when designing a system. These viewtypes represent the three kinds of decisions involved in architectural design [48]. The decision to be taken can be formulated as a question, where the viewtype is the answer:

1. How is it structured as a set of implementation units? Module.
2. How is it structured as a set of elements that have runtime behavior and interactions? Component-and-connector.
3. How does it relate to non-software structures in its environment? Allocation.

A style or pattern is a specialization of a viewtype and shows an interaction pattern and recurrent structure, independent of any system. Each viewtype has many associated styles, as shown in Figure 6.1. A style defines a family of architectures that satisfy some constraints. Styles allow applying specialized design knowledge to a particular class of systems [48]. Furthermore, a system is not necessarily and/or exclusively built from a single style; the architecture of a system can be constituted by several styles.

Usually it is not possible to describe a system in a single way; it is necessary to have more than one view of the system. Views are a representation of a set of system elements and the relationships among them. Different views highlight different quality attributes: each

view emphasizes certain aspects of the system and minimizes or ignores others [48]. In any case, the views used to document an architecture must be only those necessary to properly understand and describe the system, i.e., if two views are enough to completely describe a system, then including additional views for documentation purposes only is unnecessary, because those additional views are not adding any more information that is important for the stakeholders[1].

The previous discussion is applicable to software architectures in general, focusing on the development of single products. In the following section, product line architectures (PLA) for software product lines are discussed.

6.1.2 Product Line Architecture

A PLA is a core asset for all the products in a SPL. It explicitly provides variation mechanisms that support diversity among the products in the software product line [202]. Furthermore, it defines the concepts and structure necessary to achieve variation in features of variant products while achieving maximum sharing parts in the implementation [108]. The PLA has to include and handle an explicit representation of the variability it covers, and it guides the developers in the product specialization work [60, 125].

Taylor et al. [226] define a PLA or reference architecture as

Product line architecture

> *"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."*

Design decisions cover every aspect of the system under development, including system structure, functional behavior, interaction, non-functional properties, and system implementation [226].

The most important reusable asset within a SPL is the product line architecture (PLA) since it is shared by all the products; it is developed as part of the domain engineering process and it is used as a roadmap in the application engineering for building each product supported by the SPL [82, 125, 157, 164].

The importance of the PLA lies in its ability to reduce the complexity and cost of developing and maintaining products of the SPL. Also, it facilitates the production of documentation, training materials, and product literature [74]. Moreover, the PLA involves not only the decisions about many of the quality attributes for the system, but also integrates them with the variations of the SPL [125]. This, in turn, can be a challenge because quality properties may vary for different products belonging to the SPL [154].

[1] Krzysztof Czarnecki. Personal communication, October 2007. On Redundant Views in Software Product Architecture.

A good PLA enables organizations to accelerate the introduction of new products and improve their quality, reengineer legacy systems, and manage and enhance the needs of product variations; however, technically excellent product line architectures generally can fail due to: they are not effectively used; some are developed but never used; others lose value as product teams stop sharing the common architecture; still others achieve initial success but fail to keep up with a rapidly growing product mix; sometimes the architecture deterioration is not noticed at first, hidden by what appears to be a productivity increase [74].

It is very common that the development of the SPL requires an architecture-centric approach in which the PLA captures both the commonalities and the variabilities present in the different products [164]. Taking into account this approach, it is possible to reach the trinity of software development [134]:

Software development trinity

1. Better: It promotes designs that improve requirements consistency and traceability.
2. Cheaper: It encourages strong interface definition and control, reducing the need for nonessential or redundant code, and it helps communication among stakeholders.
3. Faster: Once standard components are architecturally compliant, they can be constructed and reused easily.

The goal of the PLA is to incorporate variability within the architecture, defined as the ability of an artefact to be efficiently extended, changed, customized or configured for use in a particular context [224]. Bachmann and Bass [9] enumerate sources of variation, such as variations in function, data, control flow, technology, quality goals, and in the environment. These authors also presented a classification for variations:

- A variation can be optional.
- A variation can be an instance out of several alternatives.
- A variation can be a set of instances out of several alternatives.

These authors' view is similar to the feature model presented in Section 5.1.4. The following section presents several notations that can be used to describe the PLA and allow describing architectural variability.

6.1.2.1 Architecture Description Language (ADL)

An ADL is a formal notation with a well-defined semantics, whose primary purpose is to represent the architecture of software systems [8]. Medvidovic and Taylor [140] provide an alternative definition: an ADL is a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation.

ADL definition

Choosing or building an appropriate ADL for a particular domain is a tough task, requiring a good domain characterization. Consequently, many ADLs have been developed, each one focused on a different application domain and its characteristics. Medvidovic and Taylor [140] and Asikainen et al. [8] compare and characterize several ADLs, many of them not necessarily suitable for SPL. Some ADLs that are suitable for building the SPL architecture, both for general purpose and/or specific domains, are: Koala [241], Koalish [7], ADLARS [11], ALI [12, 13], and xADL [64].

Koala, described by van Ommering et al. [241], was developed by Philips Electronics to help them specify and manage their consumer electronics products that work with embedded software. This ADL is based mainly on components that have interfaces as their explicit connection points. Koala components are units of design, development and reuse. A component communicates with this environment through interfaces, which are a small set of semantically related functions.

A component description language is used to describe component boundaries. Both *provides* and *requires* interfaces are detailed. Furthermore, it is possible to indicate if a component *contains* other components inside it. Similarly, a simple interface definition language is used to describe the interfaces, which uses C-like function signatures.

Koala is graphic too. Koala's graphical notation permits product design, making components look like integrated circuit chips and configurations look like electronic circuits. Interfaces are represented as pins on the chip; the triangles designate the direction of function calls.

The two main characteristics of the Koala language are switches and optional interfaces. Both are important because the first ones are architectural constructs that represent variation points, and the second ones permit to have several components providing similar, but not identical, services. So, those special constructs support product line variability.

Koalish was developed by Asikainen et al. [7], and it is focused on configurable software product families (CSPF). This subclass of software product families has the property that members of the family can be deployed in such a way that there is no need for coding within components, and little need for adding glue code between components.

Koalish is based on Koala [241], and it adds new variation mechanisms to Koala, namely the possibility of selecting the number and type of parts of components, and writing constraints that must hold for each member of the CSPF. This ADL works mainly with the idea of configurable products, i.e., each individual product is adapted to the requirements of a particular customer order. The possibilities for adapting the configurable product are predefined in a configuration model that explicitly and declaratively describes the set of legal particular product members. The configurations intend to serve as descriptions of individual systems in CSPFs.

Some concepts from Koala were not considered in Koalish. One of the omitted concepts was modules. Unfortunately, the concept of module is relevant in the meshing tool domain, according to the identified features listed in Section 4.2.1.2: all data storage, parameters,

functionalities and user interactions, should be encapsulated into modules. Without the module concept it is very difficult to build architectures for meshing tools using this ADL.

ADLARS [11] is an ADL designed to support the description of families of software systems. Particularly, it was designed for defining the product line reference architecture. The emphasis of this ADL, according to its authors, is on capturing architectural relationships between product features and the software architecture. In this sense, the assumption made is that a feature model for the application domain will be available before the architecture design process. Thus, one of the main characteristics of this ADL is that it produces reference architectures from the feature model. In this way, it is natural to pass from the domain analysis to the domain design stage considering that features both in the domain model and the product line architecture are mandatory, optional or alternative, i.e, that features in both stages have a similar associated semantic.

ADLARS is both a graphical and textual ADL, which potentiality allows its use by stakeholders with different backgrounds. The textual notation combines both formal and informal elements. For example, it is possible to have components and subcomponents, and it provides explicit representation for mandatory, optional and alternative features in the architecture. On the other hand, the graphical notation mainly represents features associated with particular components, and these features are represented as color-coded ellipses with different colors for mandatory, optional or alternative features.

ALI [12, 13] is an extensible ADL for industrial applications based on ADLARS [11]. It comprises seven parts: meta types, which provide a notation for capturing meta-information; interface types, which provide a notation for creating types of interfaces; connector types, where architectural connectors are defined; component types, where architectural components are defined; pattern templates, where design patterns are defined; features, where the system features are catalogued; and system, where the system architecture is described.

According to the authors of ALI, the main characteristics of this ADL are flexible interface description, architectural pattern description, formal syntax for capturing meta information, and linking the feature and architecture spaces. Furthermore, it supports the relationship between components, connectors, and patterns in an architecture description and features in the feature model using first order logic. These relationships allow capturing complex situations that might arise between the two artifacts in real-life systems.

In contrast to ADLARS, ALI only has a formal textual notation. Even though this characteristic has several advantages, as it allows architectural analysis and potential automation, it is not easy for every stakeholder to use it. It is necessary to count with a proper CASE toolset to use it, but that tool does not currently exist.

Dashofy et al. [64] proposed xADL 2.0, which is an infrastructure for development of XML-based architecture description languages. In this sense, xADL provides facilities for defining customized ADLs, and it is useful for modeling software architectures in many domains.

xADL has three basic components for modeling architectures: components, connectors, interfaces, and links. Components may define a compositional structure and they are the

loci of computation in the architecture. Interfaces are connection points of components and connectors. Connectors are the loci of communication in the architecture, and allow communication between components. Finally, links are connections between elements that define the topology of the architecture.

xADL allows modeling variability through the optional elements, variant types, and optional variant elements. Optional elements may or may not be included in an individual system. Variant types are related to choosing one out of two or more elements. Optional variant elements are the combination of optional elements and variant types.

This ADL has tool support by means of ArchStudio [5]. ArchStudio is an environment of integrated tools for modeling, visualizing, analyzing and implementing software and system architectures, and it is based on the Eclipse open development platform.

Even though xADL has both textual and graphical notations, it is difficult to represent alternative features of a system as the feature model proposes. The graphical notation can be used with mandatory and optional features of the system.

Δ-MontiArc [92], based on a previous work of the same authors [93], is a delta-oriented variability modeling language explicitly designed to represent architectural variability. In delta modeling, a set of systems is described by a designated core system and a set of system modifications (deltas) of the core product to obtain other products. Δ-MontiArc contains operations specifically built to model architectural variability, such as add, modify, replace component, among others. Even though this approach has several good properties such as human readability and incremental model building, it does not permit structuring variabilities in the same way that feature models do. Indeed, a delta model lies in the solution space, since it describes the variability of the artifacts that are transformed by deltas, and a feature model lies in the problem space, where the variability of the requirements is formulated[2]. This could be considered as a disadvantage from a design point of view because it is harder to establish the correspondence between the source model (requirement) and the target model (design). Furthermore, the same authors established that “delta models...are independent of product features specified in feature models” and “a product configuration (i.e. particular product) is independent of product features”.

In [94], the authors extended their work by introducing hierarchical variability modeling concepts. Even though this work explicitly incorporates concepts of *commonality* and *variability*, it seems to be difficult to establish a complete system with those commonalities and variabilities, because the main idea of the approach is centered in components, with an implementation point of view.

6.1.2.2 PLA Assessment

Architecture evaluation should be a standard part of every architecture-based development methodology. The purpose of architecture assessment is to analyze the software architecture

[2] Ina Schaefer. Personal communication, October 2012. On Delta Model

in order to identify potential risks and verify that the quality requirements have been taken into account in the design [77].

Bass et al. [17] state that having an effective technique to assess a candidate architecture and accept it as good before the implementation stage is of great economic value. Moreover, the authors enumerate at least five benefits that can be derived from holding architectural inspections: *financial benefits*, *rationale capture*, *early detection of problems with the existing architecture*, *validation of requirements*, and *architecture improvement*.

Assessment benefits

The PLA is shared by all SPL products and is used as a roadmap in the application engineering for building each product. Thus, its quality must be assessed early in the development process.

There are several methods for assessing software architectures, for both single product architectures (SPA) and for product line architectures (PLA). Dobrica and Niemelä [77] survey eight of the most representative methods for assessing single product architectures, among them the popular SAAM and ATAM [49]. Unfortunately, these methods cannot be applied directly to PLA assessment: if a product line is developed using a single product architecture, it may happen that some specific quality attributes considered for a single product cannot be directly generalized for all family products. Moreover, the number of quality attribute scenarios in a PLA assessment is generally larger and more complex than in a SPA assessment [164].

PLA assessment is outside the scope of this thesis. However, interested parties can refer to Korhonen and Mikkonen [125], which present an approach for estimating the adaptability of a SPA to a PLA, Maccari [135], which presents a method mainly centered in requirement evolution in a product family, and Gannod and Lutz [89], which present an approach for architectural analysis of product lines. Of particular relevance is the method presented by Olumofin and Mišić [164], which is based on ATAM [17].

6.2 Domain Design Method in the Meshing Tool Domain

This thesis proposes two alternatives for domain design development, specifically centered in the domain architecture, which are shown in Figure 6.2.

The deductive process for building the reference architecture has two main activities. First, the domain model is built, with a special emphasis in the feature model. Then, a product line architecture is generated using this model as a basis and any particular single product architecture is derived from it, as needed. This approach is presented in Section 6.2.1.

The transformational process is an alternative approach which also requires the construction of the domain model and then generates a feature model configuration, specifying the features present in the feature model that are present in a particular single product archi-

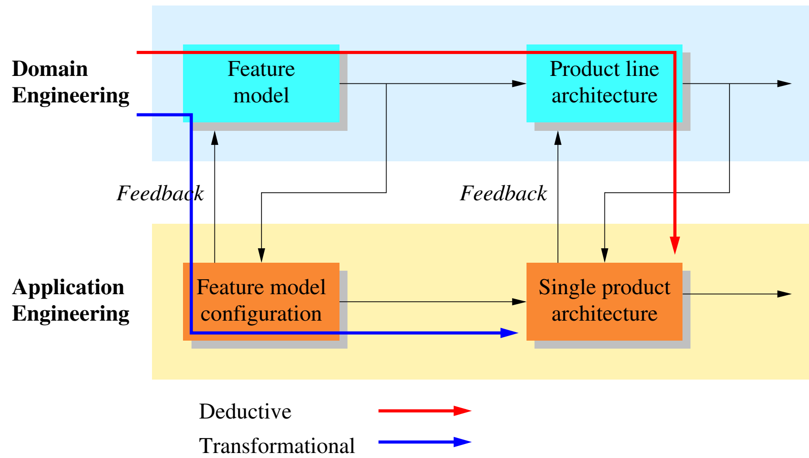


Figure 6.2: Overview of Domain Design Process

tecture. In contrast to the deductive process, a product line architecture is not generated explicitly, but rather it is implied by the superposition of all generated single product architectures. This approach is presented in Section 6.2.2.

6.2.1 Deductive Process

In this section, we present the overall deductive process for the domain design stage.

The domain design process is modeled as an activity diagram which has three main activities: determining the quality attributes, producing the architecture including both structural and a behavioral views, and assessing the architecture. An overview of this process is shown in Figure 6.3.

The activities and input artifacts of this process are subordinated to the existence of the artifacts produced during the domain analysis stage, and therefore, they are specific to the meshing tool domain. The first activity, called *Do Quality Analysis*, consists of deducing the quality attributes necessary for building the PLA using the goals identified in the domain analysis step as the starting point. The identified quality attributes are prioritized.

The second activity is *Generate Architectural Views*. This activity is composed internally by two macro activities. The first one generates a structural view of the PLA, and the second one generates a behavioral view by building the state machine. If necessary, other structural views can be developed during the building of the PLA.

Once the structural and the behavioral views are built and they satisfy both the functional and quality attributes required by stakeholders, it is necessary to do an architectural assessment in the third activity called *Assess PLA*, by using an assessment method suitable for PLAs.

In the following sections, the first two activities are described in detail.

Figure 6.3: Domain Design Process with Explicit Architecture

6.2.1.1 Quality Analysis

Similarly to Niemelä and Immonen [154], a *Quality Analysis* is performed to deduce the quality attributes using the goals from the domain model identified during the domain analysis stage. This in turn helps the stakeholders transition from the analysis level to the design level, by exposing them to the existing relationships among goals and other domain model elements such as features, scenarios and lexicon.

This analysis uses the quality attributes and definitions established by Bass et al. [17] and Dobrica and Niemelä [76]. These qualities attributes are: Performance, Security, Availability, Reliability, Usability (Learnability, Efficiency, Memorability, Error avoidance, Error handling, Satisfaction), Modifiability, Maintainability, Flexibility, Scalability, Portability, Reusability, Integrability, Interoperability and Testability.

Quality attributes

The product line architect must analyze every goal in the domain model independently. Each goal should address at least one quality attribute, and any quality attribute can be addressed by more than one goal. Even though this activity is done mainly by the domain architect, it could be worked on by some or all SPL stakeholders.

Typical issues that can arise when analyzing goals are when the same quality attribute is deduced from all goals, and when it is not possible to deduce quality attributes from the current goals. In both cases, it is necessary to review or rebuild the goals, as well as all artifacts from the domain analysis, such as the domain model and the SPL scope.

Once the quality attributes are identified, they must be validated and prioritized. Both these activities must involve all stakeholders, as the quality attributes drive the building of the PLA and thus must properly represent the tradeoffs made by the stakeholders. In this sense, all stakeholders must agree on the quality attributes deduced from the goals and their relative importance.

6.2.1.2 Architectural Views Generation

Building the PLA is an inherently creative process, highly dependent on the architects' experience. DeBaud et al. [68] state that it is not possible to operationalize the building of the PLA.

This work proposes the construction of both a structural view and a behavioral view of the architecture. A pre-requisite to the architectural views generation is understanding how each architectural style addresses the quality attributes [90]. Several structural views can be built and their styles can be chosen from among several existing styles. This thesis also defines the construction of an explicit behavioral view, also called control view. This view is represented using the Data Flow architectural style [204]. For simplicity, only the state machine is provided, following the examples of Clements et al. [48] and Muccini and Bucchiarone [149], which also document behavior by means of state machines.

Figure 6.4 presents in detail the *Generate Architectural Views* activity shown in Figure 6.3.

The left-hand branch of Figure 6.4 illustrates the construction of architectural structural views, which is briefly described in the following paragraphs.

In the *Select Architectural Styles* activity, the architects use those quality attributes identified during the *Do Quality Analysis* activity of Figure 6.3 to decide which styles are more suitable for representing the architecture. This is not a trivial task, and there are several

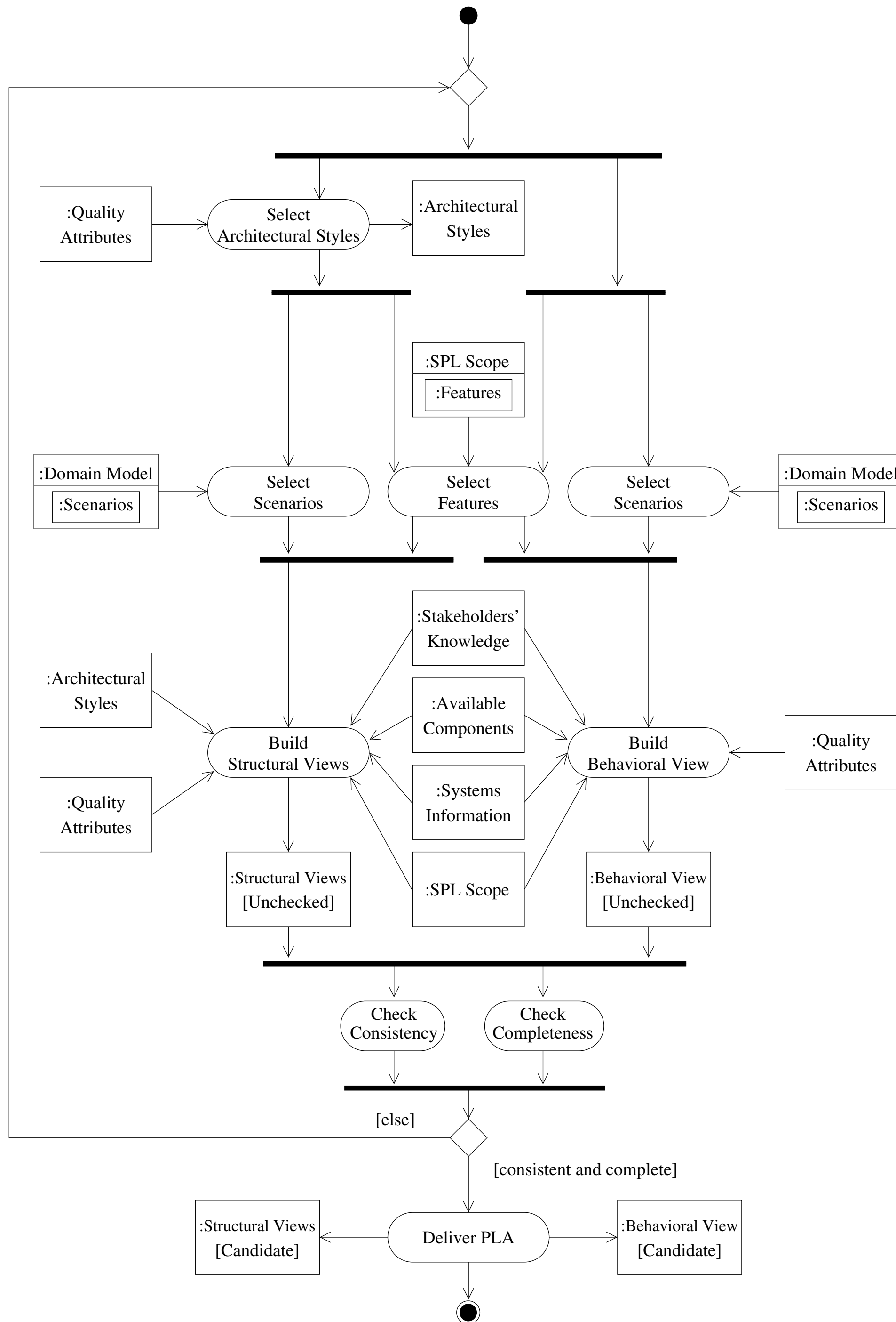


Figure 6.4: Generate Architectural Views activity

works that address this issue: Kim et al. [121] presents a classification that relates architectural styles and quality attributes; Clements et al. [48] analyze each style showing what quality attribute is better supported by which style; Niemelä et al. [155] relate architectural styles to quality attributes for systems in the wireless service domain; Gorton [90] describes how architectural styles address common quality attributes.

The architect must only choose among those styles related to the *Module* and *Component-*

and-connector viewtypes. The Allocation viewtype can be discarded, because the meshing tool domain usually involves relatively few stakeholder types and small to medium development teams, and its scope is not centered on distributed systems, i.e., they run in a single computer system[3]. Also, several final components have already been developed, and their assembly can be done automatically.

The *Select Scenarios* and *Select Features* activities are iterative processes, in which an initial group of scenarios and features are chosen by using the *By_Feature* relationships established in the domain model. Initially, a subset of all scenarios may be chosen, and expanded later in subsequent iterations, maybe by adding other scenarios and features not linked by the *By_Feature* relationship. Features and/or scenarios may be added by themselves, that is, not necessarily in pairs. Thus, the architect is free to emphasize certain aspects in each activity iteration but, in the end, all features and scenarios must be present in the architecture.

Once all scenarios and features are selected, the architect builds the structural views taking into account additional information such as the Stakeholders' Knowledge, Available Components, Systems Information, and SPL Scope, as shown in Figure 6.4. These views must now be checked for consistency and completeness.

The right-hand branch of Figure 6.4 illustrates the construction of the architectural behavioral view. The process is similar to the construction of the structural view.

Building an architectural behavioral view is critical to the thesis's approach: it helps understand and review how the meshing tools work, it is useful for building other architectural views, and it explicitly shows system behavior. Furthermore, the resulting behavioral view can be used in the Domain Implementation stage to develop an automated tool for building each particular SPL product.

A consistency check is necessary to guarantee that, for each scenario or feature selected, the corresponding and related features or scenarios have been considered in the architecture. A completeness check is necessary to guarantee that every feature, scenario, and additional information is considered in the architecture, and if those elements are enough for building all the expected products. Consequently both the software architect and the relevant stakeholders must participate in the *Check Consistency* and *Check Completeness* activities and agree with their results. If the results are unsatisfactory, the whole process is repeated iteratively.

Once a consistent and complete architecture is built, the complete PLA can be assessed.

6.2.2 Transformational Process

As described in the previous section, the software architect has a huge responsibility on the success of the SPL. Even if he or she has a well-documented domain analysis at hand, if the PLA is not appropriate, all resulting products will be flawed. This makes the PLA design

[3]Nancy Hitschfeld-Kahler. Personal communication, May 2008. On Viewtypes in Meshing Tool Domain.

task a difficult and critical one. What is even worse, if the SPL scope evolves over time, the PLA may become inadequate and, as the rationale of the PLA design is usually lost in the design process, a new PLA should be redesigned from scratch [74].

An alternative is to build only single product architectures, but this approach does not reuse software and, more importantly, does not rely on existing domain knowledge, forcing system developers to reinvent potentially reusable architectural decisions over and over again.

A third approach consists of capturing all relevant domain knowledge in a common feature model that captures commonalities and variabilities in the domain analysis stage, which is used as the basis for several single product architectures. In this approach, a PLA is not built explicitly. Rather, the superposition of all SPAs generated from the same feature model can be thought of as implicitly defining a PLA. This is valid in the meshing tool domain because the features represent functionality, user interactions, parameters or data storage, and these elements can be mapped easily to architectural elements.

This section, then, describes the application of model-driven engineering (MDE) techniques to systematize the Domain Engineering stage so as to enable the automation of the Application Engineering stage [199]. To this end, the features in the feature model are considered to represent functional areas, and architectural decisions are explicitly recorded as a set of model transformation rules attached to each feature or cohesive set of features. These transformation rules express how to build the fragment of the PLA that provides the functional areas that take care of the given features, and that organizes the identified components in a way that they address the quality attributes impacting the features, provided that certain quality attributes are specified in the form of goals. In particular, the transformation rules are expressed using ATL (ATLAS Transformation Language). According to Jouault et al. [113], ATL is a a domain-specific language for specifying model-to-model transformations buildt upon the OCL formalism. ATL is a hybrid language that provides a mixture of declarative and imperative constructs. On the one hand, the declarative style is based on specifying relations between source and target patterns and tends to be closer to the way the developers perceive a transformation. On the other hand, the imperative style is based on constructs for specifying sequences of instructions such as conditions, loops, assignments, among others.

ATL

Once the functionality of a new product is defined by a feature model configuration, the single product architecture can be automatically derived from it, and the product implementation supporting such an architecture can also be automatically generated.

The process is based on the following four main ideas [169]:

Features represent functionality: Feature models are constrained to include only those features representing functionality, user interactions, parameters or data storages. Feature models express variability only at the functional level, and not at the quality attribute level. Quality requirements are documented in a separate artifact.

Features lead component architecture construction: Each feature that may be selected as part of a product inspires a set of architectural decisions that guides the construction

of part of the architecture of a product that includes that feature. Decisions are made locally to each particular feature, only considering its direct member features.

Record the architecting activity, not the architecture: For each feature in the feature model, the set of decisions involved in providing this feature by the architecture is preserved. Such decisions are explicitly recorded as the set of actions that must be performed to build a SPA so as to support the feature. These actions are described in terms of model transformation rules that output a fragment of the SPA model when the particular feature is present in the product.

Incrementally develop the product line: The defined Domain Engineering artifacts are built incrementally. While a complete feature model is usually built during the Domain Analysis stage, other artifacts can be produced incrementally by addressing only those features that are required by each particular product under development.

In the following sections, each step of this approach is presented, for both the Domain Engineering and Application Engineering stages.

6.2.2.1 Domain Analysis

As mentioned above, the goal of this activity is to produce a feature model where features represent functionality and lead component architecture construction. This in turn leads to a feature model where leaf nodes must represent either a specific functionality provided by a product, a parametrization of such a functionality, user interaction, or access to data storage, and also can be encapsulated in a single coherent functional unit. Also, the model's internal features (i.e., those with subfeatures) represent functional areas of the SPL that can be provided by means of the interaction or combination of the functionality provided by the features they depend on, i.e., their children features.

A particular feature model complying with the previous restrictions can be built by using a metamodel. The metamodel is necessary due to the use of MDE techniques. Thus, it is important to count on a feature model metamodel for describing the elements that are part of it. Other stages can use the domain model as an input. In particular, the domain design stage uses the feature model metamodel for building the transformation rules that permit transforming feature models to the product line architecture. Moreover, the application requirements stage use the feature model metamodel for transforming the feature model into a feature model configuration, i.e., a particular instance or product of all possible products from the feature model.

In this thesis, a simplification of the metamodel proposed by Czarnecki et al. [62] is used. The simplified metamodel is illustrated in Figure 6.5.

In simple terms, the feature diagrams shown in Figures 5.1 to 5.4 conform to the feature model metamodel of Figure 6.5. Indeed, the metamodel presents the elements that are part of the feature model. The *root feature* is **Meshing Tool**. **Command language** is

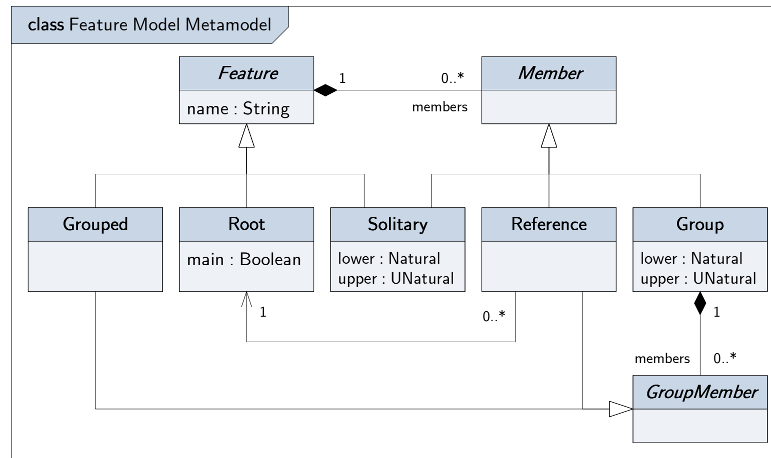


Figure 6.5: Feature Model Metamodel

a *grouped feature* in the feature *group* **User Interface**. Examples of *solitary features* are **Input**, **Region**, **Output** and **Evaluate**. Feature diagram *references* allow reusing or modularizing feature models, specially when it is necessary to modularize a large feature model over different diagrams. *References* are not present in the feature diagrams of Figures 5.1 to 5.4.

6.2.2.2 Domain Design

The goal of the Domain Design stage is to record all architectural decisions made to address the functionality and variability represented by every feature in the architecture. Figure 6.6 illustrates this stage. This figure is very similar to Figure 6.3, which presented the Domain Design process with explicit PLA. In the transformational process, however, no PLA is explicitly generated, but rather can be thought of as being implicit in the architecture transformation rules shown.

The top half of Figure 6.6 represents the inputs to the domain design process. Given that the features included in the feature model represent functional aspects, the tree-structure of the feature model is used as a guide to organize the decision making activity and modularize the architectural decisions. This thesis's approach is to explicitly record the architecting activity, not the architectural products. Quality attributes and existing implementation assets are also considered, mainly when recording design decisions associated to those features near the root of the feature model.

The bottom half of Figure 6.6 presents the *Build Model Transformations* activity, which iteratively produces a Feature-to-Architecture Transformation Rule artifact, and records the architecting activity. These transformation rules are expressed in terms of the metamodel shown in Figure 6.7.

In this metamodel, a PLA element is formed by a set of declarations and a top feature. Every declaration corresponds to a general declaration that can be used by the rules attached

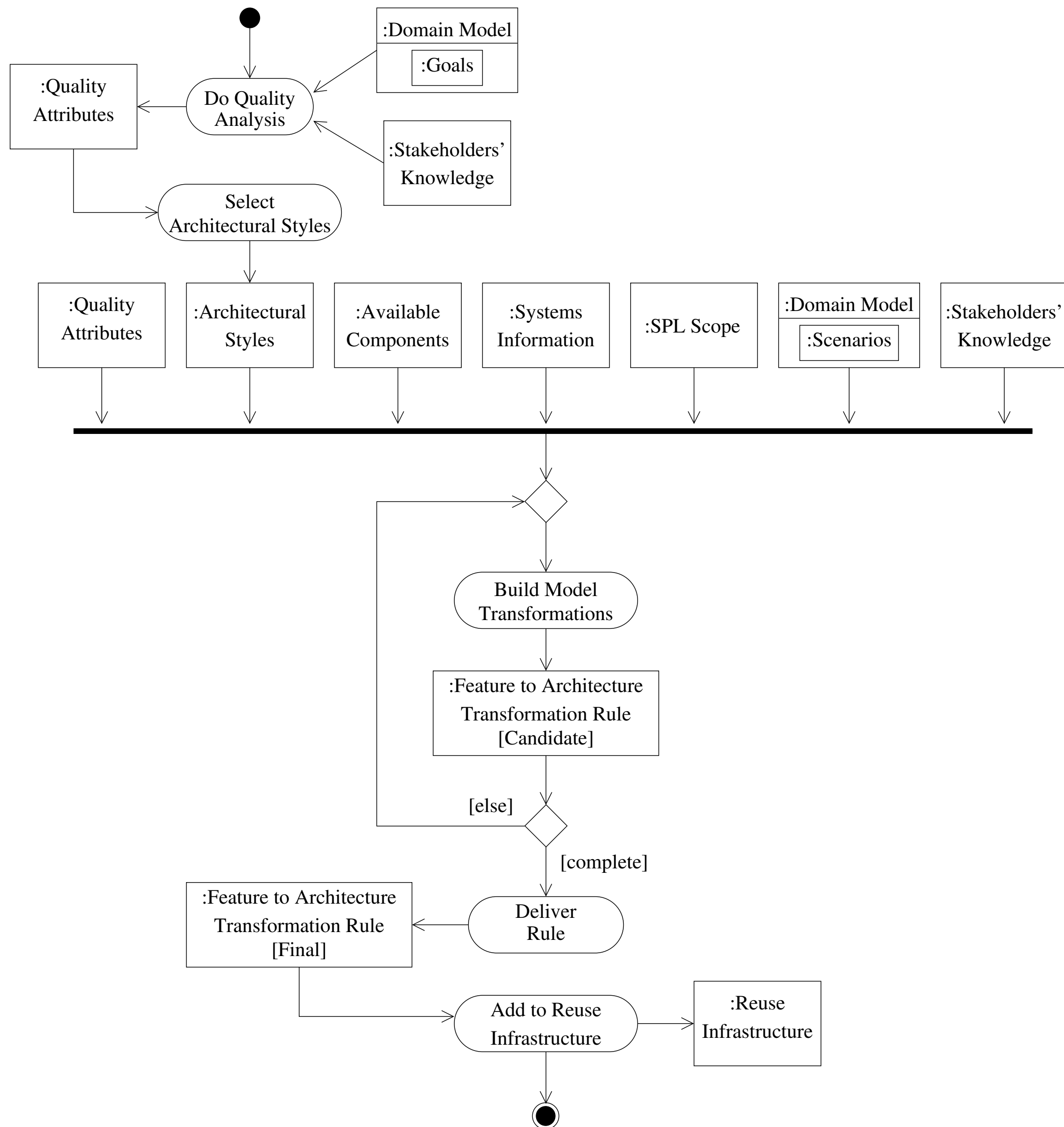


Figure 6.6: Domain Design Process with Implicit Architecture

to each feature. Every feature has a distinct name, when is used for matching purposes with the features in an input feature configuration model. Every feature also has an associated set of rules to indicate how it affects an output single product architecture (both structural and behavioral views) when the given feature is present in a feature configuration model. Declaration and rule metaclasses are abstract for portability purposes.

6.2.2.3 Application Requirements

The goal of application requirements is the selection of the desired features for a particular product. These features are selected from those provided by the SPL, considering variability constraints. Thus, a Feature Model Configuration defines which configuration of the Feature Model represents the product to be developed and consists of Features com-

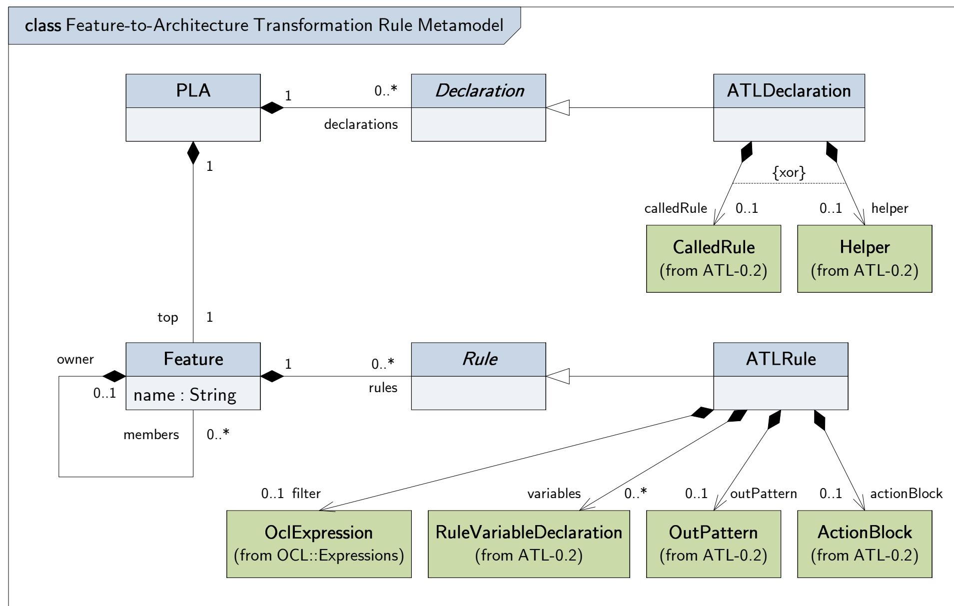


Figure 6.7: Feature to Architecture Transformation Rule Metamodel

posed by subfeatures. A Feature Model Configuration is an instance of the metamodel in Figure 6.8; the Feature Model constrains which Feature Model Configurations can be actually defined.

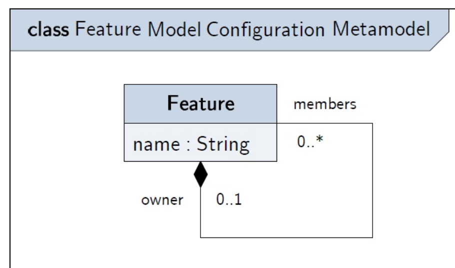


Figure 6.8: Feature Model Configuration Metamodel

6.2.2.4 Application Design

The goal of the Application Design stage is to define the single product architecture for the particular product being developed, considering its desired features as defined in a feature configuration model. The architectural decisions made during the Domain Design stage are used to produce a single product architecture. In particular, the architecture is derived using only those model transformation rules corresponding to the features included in the feature model configuration of the product under development.

To this end, a meta-transformation is defined, which takes a particular Feature-to-Architecture Transformation Rule artifact targeting a given Model-Driven Engineering (MDE) technology, e.g. ATL, and produces a Feature-to-Architecture Transformation for that MDE technology.

A specific meta-transformation is required for each MDE technology used to specialize a Feature-to-Architecture Transformation Rule.

Given the ATL specialization, the corresponding meta-transformation that transforms a Feature-to-Architecture Transformation Rule artifact to an ATL transformation is implemented. This derived transformation is then applied to the feature model configuration to obtain the particular single product architecture. By this means, the Application Design stage is fully automated. The resulting architecture has two views: a structural view and a behavioral view. The first one is an instance of the metamodel in Figure 6.9, and the second one is an instance of the metamodel in Figure 6.10.

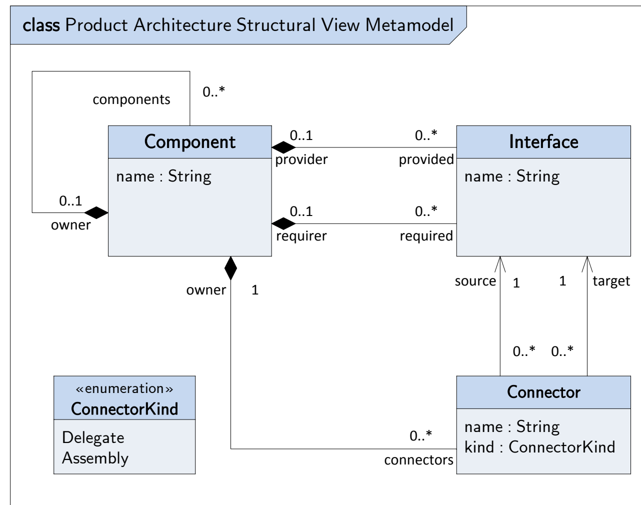


Figure 6.9: Single Product Architecture Structural View Metamodel

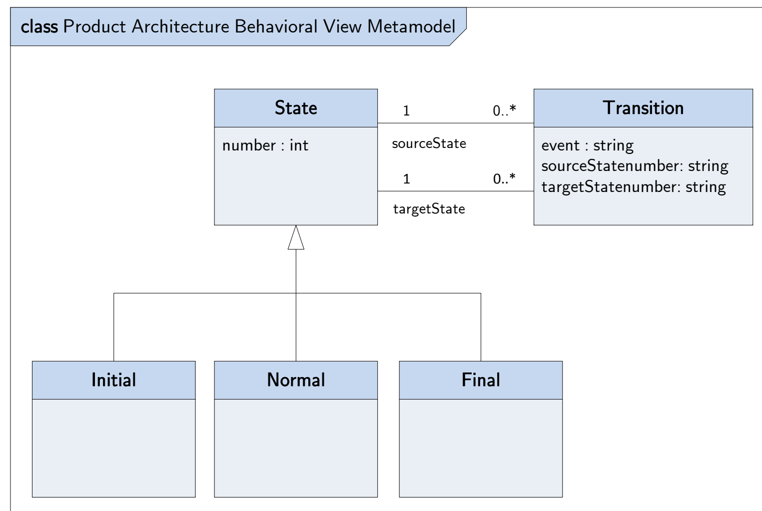


Figure 6.10: Single Product Architecture Behavioral View Metamodel

The single product architecture structural view metamodel presents the elements that are part of a product architecture structural view. It has standard elements such as *components*, *interfaces* and *connectors*, that are considered for the representation of the single product architecture structural view shown in the following chapter. Similarly, the single product

architecture behavioral view metamodel presents the elements that are part of a product architecture behavioral view. In this case, *states* and *transitions* are part of the representation of the single product architecture behavioral view shown in the following chapter.

6.3 Contributions of this Chapter

In this chapter, a complete process with activities, roles, input and output artifacts, and clear termination conditions was presented for the domain design stage. The process was divided into two subprocess: one for building an explicit product line architecture and another for building an implicit product line architecture. Both were modeled using UML activity diagrams.

Two views were considered for building the product line architecture in the domain design process: a structural view and a behavioral view. Even though in theory both views are recognized as essential, in practice only the structural one has that status. In this thesis both the structural and the behavioral view are mandatory. The behavioral view allows understanding how meshing tools work.

Also, several authors explain the benefits of considering different architectural styles into the architectural views, but only some of them detail or explain the necessary activities for developing each one. This thesis presents a detailed process which is a road map that considers the relevant quality attributes to be considered in the architecture of the SPL, starting from the goals identified in the domain analysis stage.

Another difference with respect to other approaches is that the quality attributes are incorporated in early stages of the software development life cycle, which permits a better structuring, understanding, modeling and integration with other requirements. This is because the quality attributes are deduced from the goals which are identified during the domain analysis stage.

Finally, quality attributes are considered in the feature model, but only for the transformational process for deducing the architecture.

Chapter 7

Applying the Domain Design Process

In this chapter, the application of the domain design process described in Chapter 6 is presented. First, Section 7.1 describes the application of the deductive process for obtaining a product line architecture. Then, Section 7.2 shows the application of the transformational process to obtain a FEATURE-TO-ARCHITECTURE TRANSFORMATION RULE artifact that is used to generate a specific single product architecture.

7.1 Deductive Process

7.1.1 Quality Attribute Deduction

The quality attributes are deduced from the goals defined in the domain model which was obtained in the domain analysis stage. Section 5.1.2 identifies the relevant goals, which were previously presented by Rossel et al. [188]. Table 7.1 shows the deduced quality attributes.

Goal	Quality Attribute(s)
G1	Satisfaction (usability), Reliability
G2	Satisfaction (usability), Reliability
G3	Performance, Efficiency (usability)
G4	Performance, Efficiency (usability)
G5	Scalability, Flexibility
G6	Satisfaction (usability), Reliability

Table 7.1: Correspondence between Goals and Quality Attributes

Every goal should have at least one quality attribute associated with it, but a goal may have have several different associated quality attributes.

These quality attributes are then validated by the domain stakeholders. Bastarrica and

Hitschfeld-Kahler [18] state that Performance is a priority attribute. Furthermore, Bastarrica et al. [20] establish Performance and Flexibility as priority attributes as well. Finally, Satisfaction can be deduced from several goals, implying that it is also an important quality attribute. Then, taking previous research and stakeholder-provided information into account, the quality attributes were prioritized from high to low priority order as: *Satisfaction*, *Reliability*, *Scalability*, *Performance*, *Efficiency*, and *Flexibility*.

7.1.2 Meshing Tools Structural View

The product line architecture[1] shown in this and the following section relies on the feature model already presented in Figure 5.1.

Figure 7.1 shows the structural view for the meshing tool domain. Figure 7.2 presents the User Interface component in detail, while Figure 7.3 presents a detailed view of the Algorithm component.

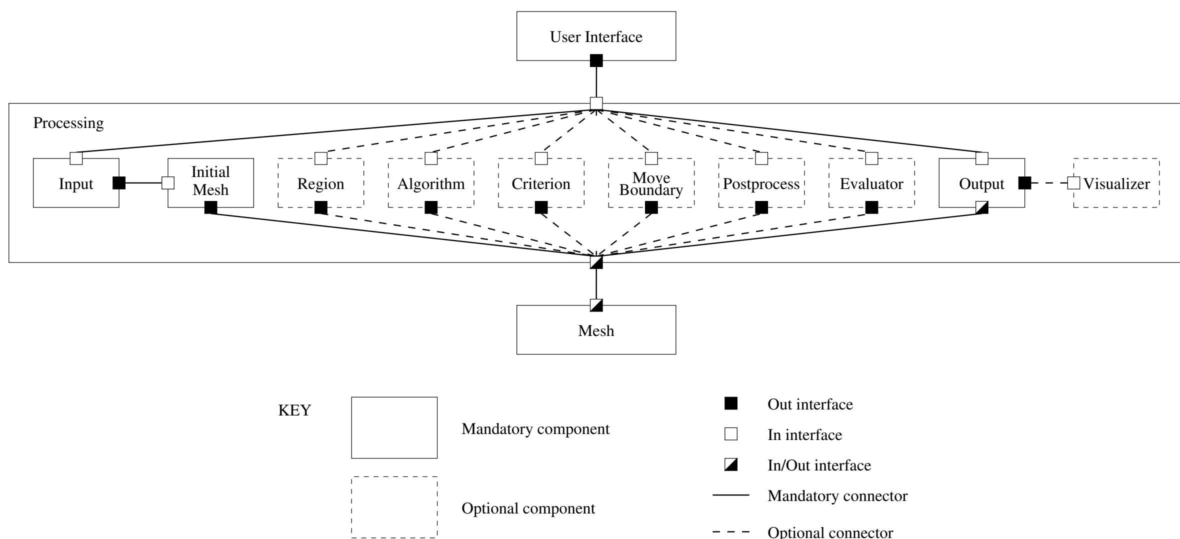


Figure 7.1: Structural View

The notation used for describing the product line architecture is inspired by the graphical notation of two ADLs presented in Section 6.1.2.1, according to their strengths: ADLARS [11] and xADL [64]. Furthermore, a new element is incorporated for alternative components: group cardinality, such as is presented in Figure 7.2 and 7.3 for `User Interface` component and `Algorithm` component respectively. This group cardinality works in a similar way to the group cardinality in a feature diagram.

The architectural style is defined through a tradeoff between stakeholders' needs and considering the quality attributes identified and presented in Section 7.1.1. As mentioned in

[1] An early version of the PLA structural view was validated in November 2007 by Todd L. Veldhuizen from the Electrical and Computer Engineering Department and R. Bruce Simpson from the Cheriton School of Computer Science, both from University of Waterloo.

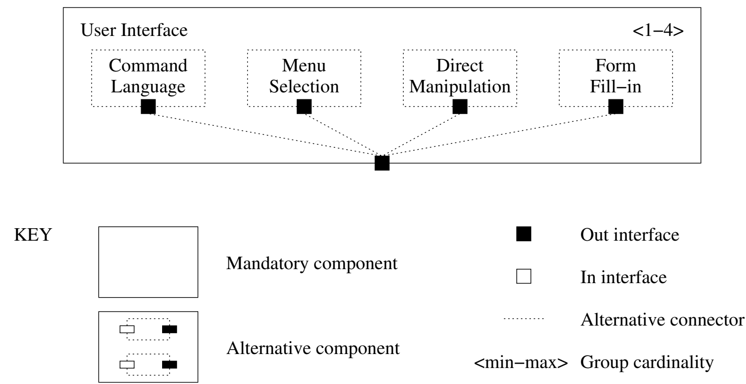


Figure 7.2: User Interface component

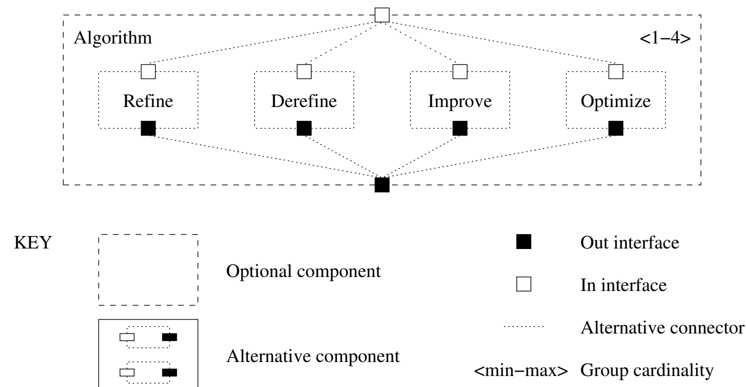


Figure 7.3: Algorithm component

Section 6.2.1.2, the selection of the most suitable architectural style for each architectural view is done considering four authors: Clements et al. [48], Gorton [90], Kim et al. [121] and Niemelä et al. [155]. Furthermore, the stakeholders' knowledge is important when all relevant information about the relationship between quality attributes and architectural styles is not available. In this case, the *layer* architectural style was chosen because it reflects *Satisfaction* and *Flexibility* well, it is neutral about *Scalability*, and according to Bastarrica et al. [20], the *layer* style does not degrade significantly the *Performance* when considering the meshing tool domain.

Even though *Reliability* is better reflected in the blackboard architectural style, and *Efficiency* is not promoted in the *layer* style, the final decision considers only one style, although that decision can be changed as a result of an architectural assessment.

The decision to use the *layer* style in the structural view is made considering the quality attributes, the prioritization that was done in the previous paragraphs, and the experience of the architect. There, *Satisfaction* is the quality attribute with the highest priority, and such as was mentioned, *layer* reflects well this quality attribute.

If only one quality attribute is considered, other styles can be chosen. For example, if only *Performance* is considered when selecting the architecture style, then, according to Clements et al. [48], one option is the *pipe-and-filter* style, even though Kim et al. [121] established

that *pipe-and-filter* does not reflect *Performance* well. Moreover, *pipe-and-filter* style must be avoided if interaction between components is required [226], which is the case of meshing tools. If either of the *Reliability*, *Satisfaction* or *Performance* quality attributes is considered, a *blackboard* style [48, 121] is a good option. If either *Reliability*, *Scalability* or *Performance* is chosen, a *N-tier* style [48, 90] is a good option. In particular, a *3-tier* style is recommended when it is necessary to process, store and retrieve significative amounts of data, and each *tier* is in charge of one of them [48, 226].

In the end, the architect is free to choose any of the previously mentioned styles, or even a hybrid style, according to which quality attribute(s) is (are) more important in each case.

For the structural view, the feature diagram presented in Figure 5.1 was considered, specifically the features belonging to the first level, i.e., those nearest to the **Meshing Tool** root. The features are distributed in three layers: User Interface, Processing and Mesh. The first and the last layer are directly related to the features of the same name. The remaining features correspond to the other components in the processing layer.

This correspondence is possible because the features represent functionality, user interactions, parameters or data storage (see Section 4.2.1.2), characterizations that are suitable and useful when defining architectural components. Furthermore, there is a correspondence between mandatory features, optional and grouped features, and mandatory, optional, and grouped components.

According to the previous paragraph, it is possible to have components inside other components, as is shown in Figure 7.2 and Figure 7.3 for the `User Interface` and `Algorithm` components, respectively. Sub-components conform to components in the same way that sub-features conform to features in the feature diagram. Constraints such as *Requires* and *Excludes* are considered in the behavioral view.

When a component is being considered for its inclusion in the PLA, the Available Components, the Systems Information, the SPL Scope, and the Stakeholders' Knowledge shown in the center of Figure 6.4 is also reviewed. Furthermore, the scenarios are also reviewed regarding their relationship with features when considering them for the construction of the structural views.

7.1.3 Meshing Tools Behavioral view

Figure 7.4 shows the behavioral view for the meshing tool domain as a state machine.

Similarly to the structural view, the behavioral view considers the feature diagram presented in Figure 5.1, specifically the features belonging to the first level, i.e., the nearest to the **Meshing Tool** root.

In the behavioral view case, scenarios identified during the domain analysis stage are very important, because they reflect the behavior of the different possible meshing tools. Further-

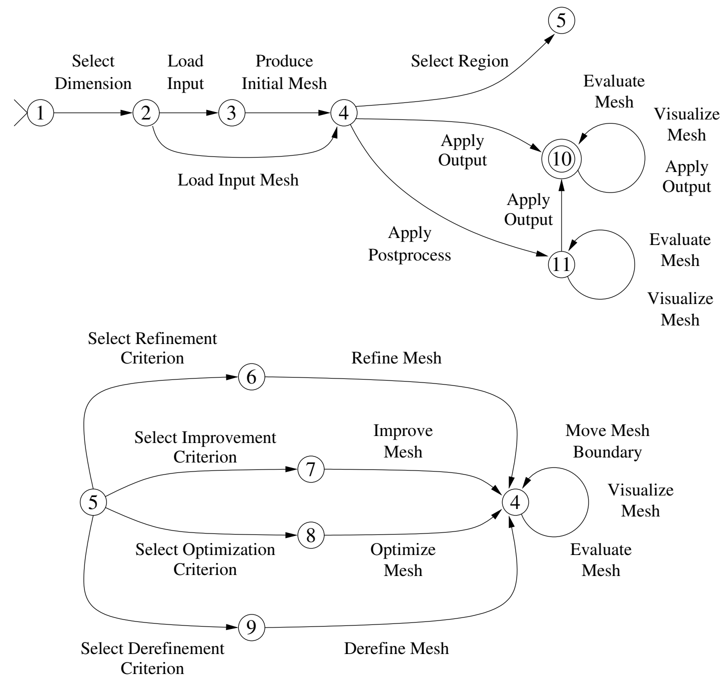


Figure 7.4: Behavioral View

more, the relationship *By_Feature*, defined in Section 4.2.2.1 and exemplified in Table 5.3, clearly establishes which features must be considered in the behavioral view, taking into account the selected scenarios.

This behavioral view reflects all potential interactions between the components identified in the structural view through the feature model. Furthermore, as a characteristic of the state machine, the different features can be mapped into this view, considering if they are mandatory, optional and/or alternative features. For example, the feature **Generate initial mesh** is optional because it is possible to load a generated mesh directly from the input in a suitable format, and **Input** is mandatory because always is necessary to load the geometry or the mesh that represents the domain itself. In the state machine, this situation is shown through the states ②, ③ and ④. If a meshing tool needs only to load the mesh and does not need to generate it, the state machine provides the transition from state ② to state ④. On the other hand, if a meshing tool needs to generate an initial mesh, then the state machine provides the transition from state ② to state ③ and the transition from state ③ to state ④. An analogous process is followed for the other features.

The constraints *Requires* and *Excludes* constrains the possible transitions between states. These are not shown in Figure 7.4 because they are not present at the first level in the feature diagram.

Even though the quality attributes also drive the construction of the behavioral view, according to Clements et al. [48] this view permits reasoning about the completeness, correctness and the quality attributes of the system, because it is possible to simulate the behavior of all products in the SPL through this view.

Finally, this behavioral view can change as a result of an architectural assessment.

7.2 Transformational Process

7.2.1 Domain Analysis

In the Domain Analysis stage, the *FeaturePlugin* tool developed by Antkiewicz and Czarnecki [2] was used to convert the feature model shown in Figure 5.1 to a XML representation. Also, a text-to-model transformation was developed [169] for transforming the XML file produced by the *FeaturePlugin* tool to the corresponding model instance of the metamodel shown in Figure 6.5.

7.2.2 Domain Design

In the Domain Design stage, a FEATURE-TO-ARCHITECTURE TRANSFORMATION RULE artifact is built iteratively. First, we use a transformation to create an initial version of this artifact from the feature model, which contains only the defined features and their member relationships. Then, the artifact is manually duplicated and augmented to include the required declarations, together with the rules for each feature. This artifact duplication is necessary because of the existence of two architectural views, and the declarations and the rules for each feature are different in each view, due to the nature of the structural and behavioral views. The FEATURE-TO-ARCHITECTURE TRANSFORMATION RULE artifact conforms to the ATL specialization of the metamodel illustrated in Figure 6.7. Figure 7.5 presents a fragment of the rule for the *Meshing Tool* feature, expressed in ATL, for the structural view.

The rule corresponds to the MESHING TOOL feature (line 1) in the case where the optional ALGORITHMS feature is selected (line 2); **f** represents the Feature element of the source FEATURE MODEL CONFIGURATION. This rule encodes which architectural patterns govern the overall structure of the product architectures; in this case study, a hybrid architectural style, based on the *3-tier* pattern where the two bottom-most tiers follow the *blackboard* pattern, is applied [34]. The architectural style was chosen considering the elements exposed in Section 7.1.2. In this case, it was considered appropriate to change the style from *layer* to the hybrid architectural style mentioned, because the *3-tier* and *blackboard* styles reflect well the quality attributes identified in Section 7.1.1 for this SPL. The rule requires a component **c** to be present in the target PRODUCT ARCHITECTURE STRUCTURAL VIEW model (line 5-8), with the same name as the feature and whose subcomponents are those generated by the rules corresponding to the subfeatures of **f** (line 6). The connectors for **c** are those defined in this rule. The figure includes two examples: a connector linking the USER INTERFACE and INPUT subcomponents (lines 9-13), and several connectors linking the USER INTERFACE to each provided interface of the ALGORITHMS component (lines 14-18).

```
1  ATLRule for 'Meshing Tool' {
2    filter f.members→select(fi |fi.name = 'Algorithms')→notEmpty();
3    variable inames : Sequence(String) = thisModule.getAlgorithmFeatures(f)→collect(fa |fa.name)→asSequence();
4    out {
5      c : PAMM!Component (
6        name ←f.name, components ←f.members,
7        connectors ←Set{xinput, xgenerate, xgeneratemesh, xoutput, xoutputmesh, xalg, xalgmesh}
8      ),
9      xinput : PAMM!Connector (
10       name ←'Input', kind ←#Assembly,
11       source ←c.components→any(ci |ci.name = 'User Interface').required→any(ii |ii.name = 'IInput'),
12       target ←c.components→any(ci |ci.name = 'Input').provided→first()
13     ),
14     xalg : distinct PAMM!Connector foreach(iname in inames) (
15       name ←iname, kind ←#Assembly,
16       source ←c.components→any(ci |ci.name = 'User Interface').required→any(ii |ii.name = 'I' + iname),
17       target ←c.components→any(ci |ci.name = 'Algorithms').provided→any(ii |ii.name = 'I' + iname)
18     ),
19     ...
20   }
21 }
```

Figure 7.5: Feature-to-Architecture Transformation Rule for the Meshing Tool Feature, Structural View Part

Similarly, Figure 7.6 presents a fragment of the rules artifact for the *Meshing Tool* and *Output* features for the behavioral view.

```
1  ATLRule for 'Meshing Tool' {
2    out {
3      s1 : PABvMM!Initial (number ←'1'),
4      s2 : PABvMM!Normal (number ←'2'),
5      t1 : PABvMM!Transition (
6        sourceStatenumber ←s1.number, event ←'select dimension', targetStatenumber ←s2.number
7      )
8    }
9  }
10
11 ATLRule for 'Output' {
12   out {
13     s1 : PABvMM!Normal (number ←'4'),
14     s2 : PABvMM!Final (number ←'5'),
15     t1 : PABvMM!Transition (
16       sourceStatenumber ←s1.number, event ←'apply output', targetStatenumber ←s2.number
17     ),
18     s3 : PABvMM!Final (number ←'5'),
19     s4 : PABvMM!Final (number ←'5'),
20     t2 : PABvMM!Transition (
21       sourceStatenumber ←s3.number, event ←'apply output', targetStatenumber ←s4.number
22     )
23   }
24 }
```

Figure 7.6: Feature-to-Architecture Transformation Rules, Behavioral View Part

The rules artifact corresponds to the MESHING TOOL feature (line 1) and the OUTPUT feature (line 11). In this case, both features were selected in the source FEATURE MODEL CONFIGURATION, because they are mandatory. This artifact has the necessary rules for building the corresponding product architecture behavioral view such as a state machine. In the case of the MESHING TOOL feature, the rule requires two states **s1** and **s2** to be present in the target PRODUCT ARCHITECTURE BEHAVIORAL VIEW model (lines 3-4). The transition between **s1** and **s2** states is defined in this rule (line 5). In the case of OUTPUT feature, the rule requires four states and two transitions. The transition **t1** (line 15) with its corresponding states **s1** and **s2** (lines 13-14) and the transition **t2** (line 20) with its corresponding states

s3 and **s4** (lines 18-19) are triggered by the same APPLY OUTPUT event. Those transitions are necessary because the APPLY OUTPUT event aims to a Final state, and once this Final state is reached, the automaton can remain in this state. More ATL rules can be found in the Appendix C.

7.2.3 Application Requirements

In the Application Requirements stage, the *FeaturePlugin* tool is used to create the feature model configuration defining the desired features for the new product being built; Figure 7.7 illustrates an example of the selected features. A text-to-model transformation is used to obtain this model as an instance of the metamodel shown in Figure 6.8.

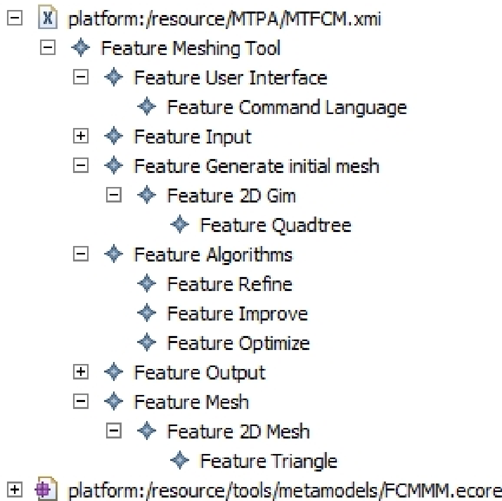


Figure 7.7: Feature Model Configuration for a Meshing Tool

7.2.4 Application Design

In the Application Design stage, the meta-transformation is used to generate the FEATURE-TO-ARCHITECTURE TRANSFORMATION artifact from the FEATURE-TO-ARCHITECTURE TRANSFORMATION RULE, both for the structural and behavioral view. These transformations are then applied to the feature model configuration to automatically generate the product architecture. Figure 7.8 illustrates a fragment of the resulting product architecture structural view generated by the rule shown in Figure 7.5, applied to the feature model configuration shown in Figure 7.7. The meshing tool component is composed by the subcomponents generated by the rules attached to the subfeatures of the meshing tool feature.

Similarly, Figure 7.9 illustrates the resulting product architecture behavioral view generated by the rule shown in part by Figure 7.6, applied to the feature model configuration shown in Figure 7.7. Not all possible transitions are present in the figure because the feature model configuration does not consider all features present in Figure 5.1.

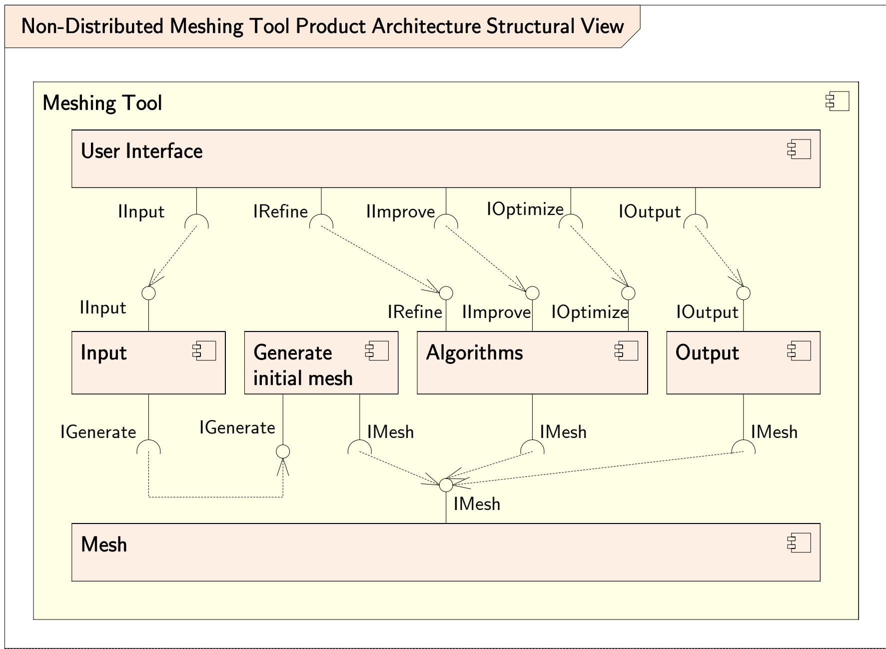


Figure 7.8: Product Architecture Structural View - Fragment for Meshing Tool Feature

7.2.5 Adding Quality Attributes to the Feature Model

The approach described in the previous sections automates the application requirements and application design stages, in the context of software product lines. This approach only considers variability at the functional level. Thus, the resulting single product architectures are guided by a single architectural style shared among all products of the SPL. It is commonly agreed that the quality of a software application not only depends on its functionality but also on other relevant quality attributes such as maintainability, performance, modularity, portability and usability, among others [177, 217, 226]. This section extends the previous work by considering variability at the quality attribute level. In this case, different architectural styles may be used to guide the design structure of different products, making apparent the need to structure and document the architectural knowledge appropriately so that software reuse is feasible [189].

In order to consider variability at the quality attribute level, the functional feature model of section 4.2.1.2 shown in Figure 5.1 is augmented by adding two quality attributes requirements: *Mesh Processing Response Time* and *Mesh Processing Distribution*. The former is mandatory because all products must comply with a specified mesh processing response time, and the latter implies a mandatory choice between *Distributed* and *Non-Distributed* processing, i.e., which represents a variability at the quality level. Again, the *FeaturePlugin* tool is used to define this feature model and a text-to-model transformation was developed to transform the XML file produced by the *FeaturePlugin* tool to the corresponding model of the metamodel in 6.5. Figure 7.10 shows the modified feature model.

Regarding the Feature-to-Architecture Transformation Rule artifact, specifically the rule for the *Non-Distributed* feature, it organizes the components according to the

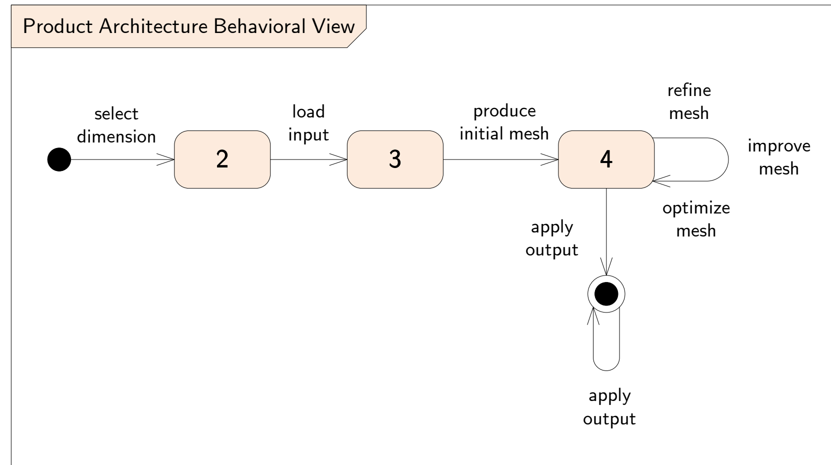


Figure 7.9: Product Architecture Behavioral View

3-tiers and *blackboard* patterns by connecting the `User Interface` component with the interfaces in the second tier (the knowledge sources), and also connects the second tier with the `Mesh` component, which plays the role of a blackboard. The resulting product architecture is shown in Figure 7.8, which is similar to the product architecture that can be derived from the product line architecture presented in Bastarrica et al. [20].

If the *Distributed* feature is chosen instead, almost all rules may be reused, and only the rule covering the distribution itself would be different. In this case, the product architecture is organized following a relaxed 4-tier architecture. A master-slave pattern [34] is used in the second tier. The `Mesh` component in the fourth tier acts as a blackboard and is connected with knowledge sources in both the second and third tiers. So, the rule adds a `Master Tool` component that is in charge of dividing the mesh, distributing it among the `Slave Tools`, and combining the results. The transformation rule also adds the `Slave Tool` component that is in charge of applying the algorithms contained in the `Algorithms` component to a part of the `Mesh`. There may be several of these components at runtime, and as such there may be several of these processes in a process view. The rule also connects the `User Interface` component with the non-distributed knowledge sources and with the `Master Tool`, and connects the `Master Tool` to the `Slave Tool` and vice versa, and connects both these components to the `Algorithms` component. Finally, the rule also connects the knowledge sources to the `Mesh` component, applying the blackboard pattern. The resulting product architecture is depicted in Figure 7.11.

Adding a quality attribute to the feature model at this level is a hard task. If the impact of the quality attribute to be added is felt only by certain components, then it may be simple to construct a transformation rule artifact that takes the new quality attribute into account. In the previous example, adding the *Mesh Processing Distribution* quality attribute affects the architecture mainly in the `Algorithms` component (see Figure 7.8), by adding two components, i.e., the `Master Tool` and `Slave Tool` components (see Figure 7.11). Other quality attributes may have a more transversal impact on almost all software architecture components. In this case, constructing the transformation rule artifacts may be considerably harder.

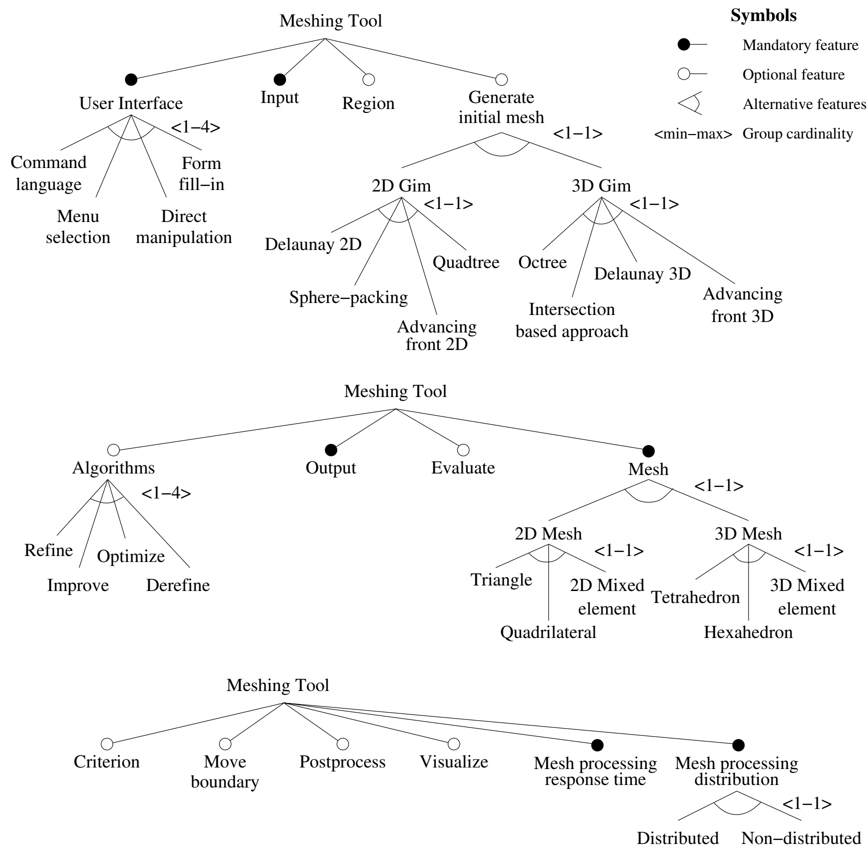


Figure 7.10: Meshing Tool Feature Diagram Simplification

Finally, if other structural and/or behavioral views must be added, they must be specified in the transformation rules. In particular for this example, the product architecture behavioral view is the same shown in Figure 7.9, essentially because the behavior of this product is the same.

7.3 Contributions of this Chapter

In this chapter, the process for building the product line architecture presented in Chapter 6 was applied completely and carefully for a meshing tool product line. In this sense, both the structural and behavioral views were built. Moreover, the two processes described for building the product line architecture –deductive and transformational– were applied. With these results, it is possible to establish for the meshing tool domain an effective process that permits building the domain design from artifacts produced during the domain analysis process.

The quality attributes, important elements of any product or product line architecture, are deduced from goals shown in Section 5.1.2.2. Even though doing a *quality analysis* from goals is not new, the novelty in this case is given by the relationships established among the different domain model elements in Section 4.2.2.1 and how the domain model elements are

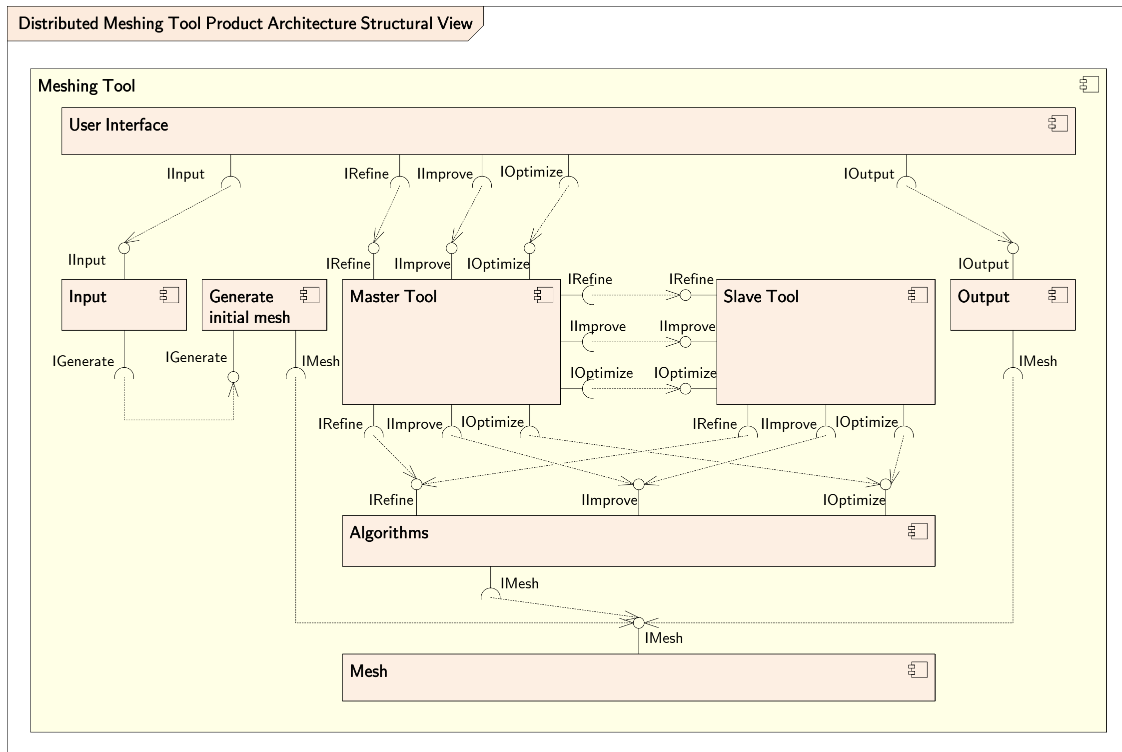


Figure 7.11: Distributed Product Architecture Structural View

mapped to the architecture.

At the same time, the transformational process for building the product line architecture was specified using ATL rules. These rules allow building several single product architectures, according to the needs of the stakeholders and to the set of features they select for producing the feature model configuration. Furthermore, these rules are useful for the creation of several more rules, which in turn permit instantiating other more sophisticated product architectures, including both product architecture structural and behavioral views.

Finally, a feature model with quality attributes was produced. Even though in the literature it is a known fact that it is possible to do so, it is not a common practice. Furthermore, in this thesis, a product architecture was derived considering quality attributes, and it is shown how to derive different architectures depending on which quality attribute was chosen.

Chapter 8

Conclusions and Future Work

8.1 Results

In the previous chapters, a software product line process specially suited to the meshing tool domain has been presented, and its characteristics have been specified using a model based on features, scenarios, goals and lexicons. The domain engineering phase described in this thesis covers mainly the first two stages of a SPL process, that is, the domain analysis and domain design stages, in accordance to the objectives defined in Section 1.2.2. Nevertheless, Appendix A gives an overview of the complete SPL process for the meshing tool domain, covering all stages in both the domain engineering and application engineering phases through activity diagrams.

`Work summary`

8.1.1 Objectives

The objectives defined in Section 1.2.2 were fulfilled completely. In the following paragraphs, the chapter and/or section of the thesis where the objectives were fulfilled are indicated.

Objective 1 Identify the characteristics of meshing tools and of the way they are built: Section 1.1.1, Chapter 3.

Objective 2 Develop a DA process, describing the sequence of activities, artifacts and roles involved: Chapter 4.

Objective 3 Identify architectural needs of meshing tools, considering the shared quality attributes: Section 5.1.2.2, Section 7.1.1.

Objective 4 Develop a DD process, describing the sequence of activities, artifacts and roles involved, in a manner consistent with the DA process: Chapter 6.

Objective 5 Validate the proposed DA and DD processes: Chapter 5, Chapter 7.

8.1.2 Publications

The following list enumerates the publications produced as a result of this research, and indicates the corresponding thesis sections, where applicable.

- Pedro O. Rossel, Daniel Perovich, and María Cecilia Bastarrica. Reuse of Architectural Knowledge in SPL Development. In *Proceedings of the 11th International Conference on Software Reuse (ICSR 2009)*, volume 5791 of Lecture Notes in Computer Science, pages 191-200. Springer, September 2009.

  This paper corresponds to Section *6.2.2 Transformational Process*, and Section *7.2.5 Adding Quality Attributes to the Feature Model*.

- Daniel Perovich, Pedro O. Rossel, and María Cecilia Bastarrica. Feature Model to Product Architectures: Applying MDE to Software Product Lines. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (WICSA/ECSA 2009)*, pages 201-210. IEEE, September 2009.

  This paper corresponds to Section *6.2.2 Transformational Process*, and Sections *7.2.1 Domain Analysis* to *7.2.4 Application Design*.

- Pedro O. Rossel, María Cecilia Bastarrica, and Nancy Hitschfeld-Kahler. A Systematic Process for Defining Meshing Tool Software Product Line Domain Model. In *Proceedings of the 12th Workshop on Requirements Engineering (WER'09)*, pages 103-114, July 2009.

  This paper corresponds to Section *1.1.1 Meshing Tools*, *4.2.1 Domain Model Construction Process*, *4.2.2 Domain Model Definition*, *4.2.4 Related Work*, and Sections *5.1.1 Lexicon* to *5.1.5 Domain Model Consistency*.

- María Cecilia Bastarrica, Nancy Hitschfeld-Kahler, Pedro O. Rossel, and César Castro. Rapidly Generating Different Meshing Tools. In *Proceedings of the 10th US International Congress on Computational Mechanics. 2.18.3 Trends in Unstructured Mesh Generation*. July 2009.

- Cecilia Bastarrica, Sebastián Rivas, and Pedro O. Rossel. From a Single Product Architecture to a Product line Architecture. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, pages 115-122. IEEE Computer Society, November 2007.

  This paper corresponds to Section *6.1.2.2 PLA Assessment*.

- María Cecilia Bastarrica, Sebastián Rivas, and Pedro O. Rossel. Designing and Implementing a Product Family of Model Consistency Checkers. In *Proceedings of the Quality in Modeling Workshop at MoDELS 2007, ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pages 36-49, October 2007.

This paper corresponds to Section *6.1.2.2 PLA Assessment*.

- María Cecilia Bastarrica, Nancy Hitschfeld-Kahler, and Pedro O. Rossel. Architecting a Family of Meshing Tools. In *Second Chilean Workshop on Numerical Analysis of Partial Differential Equations (WONAPDE 2007)*, pages 94-95. January 2007.

  This paper corresponds to Section *1.1.1 Meshing Tools*, *6.1.2 Product Line Architecture*, and *7.1.2 Meshing Tools Structural View*.

- María Cecilia Bastarrica, Nancy Hitschfeld-Kahler, and Pedro O. Rossel. A Meshing Tool Product Line Architecture. In *Proceedings of the IFIP 19th World Computer Congress. First International Workshop on Advanced Software Engineering (IWASE 2006)*, volume 219 of IFIP International Federation for Information Processing, pages 1-15. Springer, August 2006.

  This paper corresponds to Section *1.1.1 Meshing Tools*, *3.9 Developing Meshing Tools*, *6.1.2.1 Architecture Description Language (ADL)* and *7.1.2 Meshing Tools Structural View*.

- María Cecilia Bastarrica, Nancy Hitschfeld-Kahler, and Pedro O. Rossel. Product Line Architecture for a Family of Meshing Tools. In *Proceedings of the 9th International Conference on Software Reuse (ICSR 2006)*, volume 4039 of Lecture Notes in Computer Science, pages 403-406. Springer, June 2006.

  This paper corresponds to Section *1.1.1 Meshing Tools*, *3.9 Developing Meshing Tools*, and *7.1.2 Meshing Tools Structural View*.

8.2 Contributions of this thesis

Domain analysis stage

In the domain analysis stage, a rigorous process is presented, complete with activities, roles, clear termination conditions and a formalization of the model specified in the formal language Z. The model ensures the consistency among domain elements, and completeness with respect to the products that are intended to be developed. The process can be customized so as to avoid those activities that are not relevant to the particular domain, such as determining the binding time of the identified variabilities.

The process of building a domain model must necessarily address both model completeness and consistency. Deciding when requirements are complete is generally a difficult issue. The termination conditions provided in the domain analysis process presented in this thesis give a systematic means for verifying if the elements included in the domain model allow building all the products within the SPL scope.

Defining a domain model that is complete for all potential meshing tools is impractical. The objective, then, is to build a domain model that is complete enough to allow building a series of planned products. The domain model and the scope shown meet this goal, and

the iterative nature of the proposed process makes the domain model easily extendable when needed, by adding particular scenarios, features, goals and lexicons, as was shown in Section 5.1.6.3. It should be possible to build the planned products with the specified elements.

The domain model construction process must also ensure the consistency among all requirement elements. The proposed formalization of the domain model allows iterating through several inconsistent domain models, using the formal consistency conditions until a consistent domain model is established. Allowing the temporal existence of inconsistent domain models gives flexibility to the modeling process and permits model improvement while the modelers gain knowledge about the domain.

Building a good feature model for complex domains is a difficult task. Moreover, domain experts tend to think in terms of goals and scenarios. The approach presented in this thesis provides a framework and a process to guide the knowledge acquisition and recording in a natural way. Currently, there are only a handful of methods that consider scenarios and goals as part of the domain model, taking advantage of the relationships among scenarios, goals and features. The formalization of these relationships, their definition and the rationale about the domain that they support are crucial for building the domain model. This formalization is one of the main contributions of this thesis.

The SPL scope is built using artifacts produced for the domain model and an additional artifact, the product map. This is a relatively standard artifact, but in this work, it incorporates other characteristics such as cost and time of development of core assets, which are generally known but not usually used explicitly for building the scope. In our case, these two characteristics are very important when determining the best order for developing the core assets.

Domain design stage

In the domain design stage, most traditional approaches to SPLs build the product line architecture using the feature model as a basis, where features are usually associated to concrete software modules. This thesis follows a similar approach, and also considers the inclusion of quality attributes as variation points. Quality attributes were not considered initially in the feature model, but rather affected the product through their presence in the goals. Further experience with the domain guided their inclusion in the feature model so they can directly affect the PLA.

Even though there is plenty of knowledge about the meshing tool domain, it is not easy to know the necessary steps for building a PLA. To aid in this task, this thesis provides activity diagrams which are based on experience in building meshing tools SPAs and the existing relationships between domain model elements and the architectural styles, quality attributes and views.

Designing a PLA that considers functional variabilities of all potential products in the SPL is a complex task, and considering variabilities at the quality level is even more complex. This thesis provides a systematic approach to defining an explicit PLA that considers functional variabilities and an implicit PLA that considers both functional and quality variabilities, and that also enables automatic product architecture generation.

Both structural and behavioral views for the explicit architecture are provided. Even though the literature broadly explains the benefits of having both kinds of views, in practice it is common to find architectures with only structural views. The mandatory behavioral view is important to the thesis's approach, because it helps understand and review how the meshing tools work, it is useful for building other architectural views, and it explicitly shows system behavior.

The implicit PLA is built incrementally. Only those features that are present in already existing products need to have their associated rules. Thus, the SPL scope can evolve without losing the design effort already invested: as new features are added or modified, only their rules need to be added or updated, respectively. However, in certain cases, these changes may force a review of the rules affecting other features, but the divide-and-conquer strategy used for the design phase makes this task easier. In most cases, the architectural knowledge associated to each feature can be directly reused.

The systematic building of the software architecture leads to consistency between requirements specified in the form of the feature model. Also, the sequence of model transformations applied in the design process enables traceability, another desirable quality.

Finally, some objections that may be raised regarding the process described in this thesis are described and addressed in the following paragraphs.

A few comments regarding the DA stage include that, from a documentation perspective, the quantity of elements of the domain model may be considered excessive by some stakeholders, that consistency and completeness verification is a rather tedious activity, and that managing the different domain model elements may be error prone, given the current shortage of automated tools. A possible solution is the automatization of all or part of the DA process.

In the DD stage, the relationships of the different elements that participate in the PLA must be formalized, and the termination conditions for the completeness and consistency of the architecture must be established. Even though the different elements used to build the architecture are the same for building the domain model and their relationships have already been documented, other architectural particular elements may arise, for example, issues of architectural style.

As SPL approaches are centered in architecture, it is highly recommended to assess the PLA. The transformational approach described does not provide an explicit PLA, but rather an implicit PLA is defined instead by the Feature-to-Architecture Transformation Rule artifact. Therefore, this implicit PLA cannot be assessed with traditional methods. The transformational process generates explicit single product architectures that can be assessed using traditional methods, but this can be expensive if the number of products in the SPL is large. Nevertheless, representative product architectures can be automatically generated so as to perform architecture assessment.

The definition of the transformation rules is not a trivial task, and the definition of transformation rules for several tools can be considered a hard and creative activity. Therefore,

it is difficult to express some of the activities in the the UML activity diagram depicted in Figure 6.6, specifically the *Build Model Transformations* activity, in more detail, as this activity is of an unstructured nature.

The complete process for developing a meshing tool SPL works well because this domain is stable and known. If these conditions are modified because of environmental variables, then the construction of the domain model, scope, and architecture of the SPL can change. This would, in turn, imply changes to the UML activity diagrams of both the DA and DD stages presented in the Chapter 4.2 and 6.2 respectively, and to the activity diagram of the derivation process presented in Section A.2.

Finally, a Meshing Tool Generator application was developed by Díaz [73] following the domain model structured in this thesis and its related works. This generator allows stakeholders to automatically create meshing tool applications. The generator can produce two kinds of mesh generators (Tree Growth Simulator and Generic Mesh Generator), it can use several different input/output files for the mesh, and it can apply different algorithms to the mesh (refine, derefine, among others), considering different quality mesh criteria. In particular, Díaz's work compares six products, describing two of them in detail. In this comparison only final product size and compilation time were considered. Execution time was omitted because it was similar among the different applications.

`Meshing tool generator`

This Meshing Tool Generator validates the domain model built in this thesis, even though the complete process presented in this thesis was not applied to construct the generator. Further work is needed to apply the complete process described in this thesis, strongly considering the product line architecture.

8.3 Knowledge Transfer to Other Domains

Knowledge developed or obtained during the thesis's process is amenable for transfer to other domains. However, it is difficult to provide enough evidence to validate effective knowledge transfer from the meshing tool domain to other domains, because basically only the meshing tool domain has been explored. In this sense, a lot of work is required, and this issue is outside the scope of the thesis.

8.3.1 Domain Analysis

The domain model construction process, sketched in Figure 4.6, is a structured and detailed process that can be used in other domains, as the activities considering actors and elements can be present in any domain: stakeholders, developed systems, and components as a product of the development process, among others. Furthermore, the domain model itself contains artifacts such as goals, features, scenarios, actions and a lexicon, that can be and have been used in several different domains [167, 187, 226, 250]. Finally, consistency and completeness

checking are activities highly required by any model or requirements document for any kind of system. The domain model is not the exception. In particular, the domain model formal definition, shown in Section 4.2.2.1 helps in the consistency and completeness checking, and it is sufficiently general for any domain.

To be able to apply this process to other domains, it must be noted that data storage, parameters, functionalities and user interactions were considered valid abstractions for the feature model in the meshing tool domain, as was mentioned in Section 4.2.1.2. Therefore, in the new domain, the existence of these abstractions is required.

The domain scope definition process shown in Figure 4.7 is applicable to other domains too. Its elements (the business goal, the feature model and the product map) are widely used separately in many domains for SPL developments, with some minor differences [51, 195]. The idea of integrating these three elements into the scope is novel, and this integration leads to a clearer understanding of the domain and all products supported by the SPL.

8.3.2 Domain Design

Similarly to the domain analysis stage, the process shown in Figure 6.3 for building both structural and behavioral view is transferrable to other domains. This process considers actors and elements that can be present in any domain: architects, domain experts, quality attributes, developed systems, and components as a product of the development process, among others. The deduction of quality attributes from goals, as explained in Section 6.2.1.1, is not a standard form to obtain them, but a similar process has been applied in another domain [154]. The selection of scenarios and features from the domain model, and other elements such as domain scope and developed systems, is an important guideline that facilitates building the product line architecture, because it specifies explicitly which elements must be considered.

On the other hand, the transformational process for building the product architecture can be transferrable to other domains. Furthermore, the metamodels can be used almost without changes. However, it is necessary to consider that the features used in the feature model, input element for this process, must be data storage, parameters, functionalities and user interactions. Finally, all transformation rules must be built from scratch.

8.4 Future Work

Regarding future work, there are several development avenues to be pursued both in the immediate future and in forthcoming work. The following paragraphs describe some of these ideas.

This thesis focuses its endeavor in the domain analysis and domain design stages. The

next step is to proceed to the domain implementation stage. Here there are two possible lines of inquiry. First, a domain-specific language (DSL) can be used to take advantage of the abstractions found in a specific domain, and to produce particular products for that domain. Losilla et al. [132] and Vicente-Chicote et al. [243] present examples of this approach. In regard to the thesis, there has been initial work toward building a DSL for the meshing tool domain using the feature model and the behavioral view as a basis, and the Graphical Modeling Framework from Eclipse (GMF, http://www.eclipse.org/gmf/) as a DSL development tool, with promising results. In this sense, a domain model and the product line architecture are the basis for building any DSL for the meshing tool domain. Without these artifacts, DSL construction is a very hard task. Secondly, building meshing tool domain applications following the approach presented in this thesis necessarily requires a repository of pre-built and pre-tested software components. These components can be obtained from previous system developments or can be built from scratch.

Regarding the domain analysis stage, both goals and scenarios have been written using natural language, without any predefined structure. This, in turn, does not permit their automatized validation. This can be improved by imposing a structure upon these goals and scenarios, thus also making it possible to advance in their formalization through Z schemas. This issue has been explored by Basili et al. [15] and Rolland et al. [187].

In a similar manner, there are no explicit relationships between the business goals and any element of the domain model. It seems that relationships can be naturally established between the business goals and the domain model goals. These relationships can be captured and formalized as Z schemas.

In the domain design stage, there are two interesting areas worth exploring. The transformational process shown in Section 7.2.5 includes quality attributes in the feature model and shows how to use them to deduce reasonable architectures. The next step is to change the deductive process so that it too can include quality attributes in the feature model. This, in turn, will also require changes to the domain model definition and the domain analysis process.

This work is oriented to the definition of processes aimed at generating reasonable product line architectures, and no formal assessment of the resulting architectures is done. Therefore, the application of a formal architectural assessment method such as the one presented by Olumofin and Mišić [164] is needed.

The formalization of the relationships among domain design artifacts is a pending task. Even though there are several explicit relationships among some elements that participate in the product line architecture, as stated in Section 6.2 for goals and quality attributes, quality attributes and architectural styles, features and components in a structural view, and features and transitions in a behavioral view, it is not clear at all how to organize them in a coherent Z schema, in order to provide conditions for consistency and completeness checking of the architectural views.

Considering the statements in previous paragraphs related to the formalization of several artifacts and relationships, it is possible to apply formal methods to the SPL. This idea is

not new [107] and has gained strength in recent years [192]. In that sense, HATS (Highly Adaptable and Trustworthy Software Using Formal Models) [44, 45] is a methodology that integrates established formal methods such as TVL (Textual Variability Language) [46] and ABS (Abstract Behavioral Specification language) [112] in almost all stages of the SPL development life-cycle. This is several steps ahead of the formalization proposed in this thesis. However, if the HATS approach is to be applied to the meshing tool domain model, there are some issues to be decided: how well HATS adapts to the domain, its abstractions and characteristics; what domain model artifacts besides the feature model (goals, scenarios, actions and lexicon) used in this thesis can be modeled and formalized in HATS; and what relationships among domain model artifacts can and must be modeled and formalized in HATS.

Finally, the methodology applied in this thesis to the meshing tool domain can be extended to other domains, as mentioned in Section 8.3. For example, there is work in progress on modifying the activity diagrams of the domain analysis stage presented in this thesis to aid their application to software process lines. Additionally, there is interest in applying this methodology to the mobile collaborative applications domain, work which is currently at the domain analysis stage.

References

[1] Ali, M. S., Babar, M. A., and Schmid, K. 2009. A Comparative Survey of Economic Models for Software Product Lines. *In* Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009), pp. 275–278. IEEE Computer Society.

[2] Antkiewicz, M. and Czarnecki, K. 2004. FeaturePlugin: Feature Modeling Plug-in for Eclipse. *In* Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange (eclipse '04), pp. 67–72. ACM.

[3] Apel, S. and Kästner, C. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8:49–84.

[4] Arango, G. 1994. A Brief Introduction to Domain Analysis. *In* Proceedings of the 1994 ACM Symposium on Applied Computing (SAC '94), pp. 42–46. ACM Press.

[5] ArchStudio 4 2006. ArchStudio 4. Software and Systems Architecture Development Environment. Institute for Software Research, University of California, Irvine. http://www.isr.uci.edu/projects/archstudio/.

[6] Ardis, M., Daley, N., Hoffman, D., Siy, H., and Weiss, D. 2000. Software product lines: a case study. *Software: Practice and Experience* 30:825–847.

[7] Asikainen, T., Soininen, T., and Männistö, T. 2003a. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. *In* F. van der Linden (ed.), 5th International Workshop on Software Product-Family Engineering (PFE 2003), volume 3014 of *Lecture Notes in Computer Science*, pp. 225–249. Springer.

[8] Asikainen, T., Soininen, T., and Männistö, T. 2003b. Towards Managing Variability Using Software Product Family Architecture Models and Product Configurators. *In* J. van Gurp and J. Bosch (eds.), Proceedings of Software Variability Management Workshop, pp. 84–93.

[9] Bachmann, F. and Bass, L. 2001. Managing Variability in Software Architectures. *In* Proceedings of the Symposium on Software Reusability (SSR '01), pp. 126–132. ACM.

[10] Barbacci, M., Klein, M. H., , Longstaff, T. A., and Weinstock, C. B. 1995. Quality Attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute.

[11] Bashroush, R., Brown, T. J., Spence, I. T. A., and Kilpatrick, P. 2005. ADLARS: An Architecture Description Language for Software Product Lines. *In* Proceedings of 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29 2005), pp. 163–173. IEEE Computer Society.

[12] Bashroush, R., Spence, I., Kilpatrick, P., Brown, T. J., Gilani, W., and Fritzsche, M. 2008. ALI: An Extensible Architecture Description Language for Industrial Applications. *In* 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008), pp. 297–304.

[13] Bashroush, R., Spence, I. T. A., Kilpatrick, P., and Brown, T. J. 2006. Towards More Flexible Architecture Description Languages for Industrial Applications. *In* V. Gruhn and F. Oquendo (eds.), Proceedings of the 3rd European Conference on Software Architecture, volume 4344 of *Lecture Notes in Computer Science*, pp. 212–219. Springer.

[14] Basili, V. R., Bailey, J., Joo, B. G., and Rombach, H. D. 1987. Software Reuse: A Framework for Research. *In* Proceedings of the 10th Minnowbrook Workshop on Software Performance Evaluation.

[15] Basili, V. R., Caldiera, G., and Rombach, H. D. 1994. Goal Question Metric Paradigm, pp. 528–532. *In* Encyclopedia of Software Engineering. John Wiley & Sons.

[16] Bass, L., Clements, P., and Kazman, R. 1998. Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley.

[17] Bass, L., Clements, P., and Kazman, R. 2003. Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley, 2nd edition.

[18] Bastarrica, M. C. and Hitschfeld-Kahler, N. 2006. Designing a product family of meshing tools. *Advances in Engineering Software* 37:1–10.

[19] Bastarrica, M. C., Hitschfeld-Kahler, N., and Rossel, P. O. 2006a. A Meshing Tool Product Line Architecture. In [161], pp. 1–15.

[20] Bastarrica, M. C., Hitschfeld-Kahler, N., and Rossel, P. O. 2006b. Product Line Architecture for a Family of Meshing Tools. In [148], pp. 403–406.

[21] Batory, D. S., Sarvela, J. N., and Rauschmayer, A. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30:355–371.

[22] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M. 1999a. PuLSE: A Methodology to Develop Software Product Lines. *In* Proceedings of the Fifth Symposium on Software Reusability (SSR'99), pp. 122–131, Los Angeles, CA, USA. ACM Press.

[23] Bayer, J., Gacek, C., Muthig, D., and Widen, T. 2000. PuLSE-I: Deriving Instances from a Product Line Infrastructure. *In* Proceedings of the 7th IEEE International Symposium on Engineering of Computer-Based Systems (ECBS 2000), pp. 237–245. IEEE Computer Society.

[24] Bayer, J., Muthig, D., and Widen, T. 1999b. Customizable Domain Analysis. *In* Proceedings of the 1st International Symposium Generative and Component-Based Software Engineering (GCSE'99), volume 1799 of *Lecture Notes in Computer Science*, pp. 178–194. Springer.

[25] Bern, M. and Eppstein, D. 1995. Mesh Generation and Optimal Triangulation, pp. 47–123. *In* D.-Z. Du and F. K.-M. Hwang (eds.), Computing in Euclidean Geometry, number 4 in Lecture Notes Series on Computing. World Scientific, 2nd edition.

[26] Bern, M. and Plassmann, P. 1999. Mesh Generation, pp. 291–332. *In* J.-R. Sack and J. Urrutia (eds.), Handbook of Computational Geometry, chapter 6. North-Holland Publishing Co.

[27] Bernatz, R. A. 2010. Fourier Series and Numerical Methods for Partial Differential Equations. John Wiley & Sons.

[28] Berti, G. 2000. Generic Software Components for Scientific Computing. PhD thesis, BTU Cottbus, Germany.

[29] Berti, G. 2001. A Generic Toolbox for the Grid Craftsman. *In* Proceedings of the 17th GaMM-Seminar Leipzig on Construction of Grid Generation Algorithms, pp. 1–28.

[30] Böckle, G., Clements, P., McGregor, J. D., Muthig, D., and Schmid, K. 2004. Calculating ROI for Software Product Lines. *IEEE Software* 21:23–31.

[31] Bosch, J. 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In [42], pp. 257–271.

[32] Bruaset, A. M. and Langtangen, H. P. 1997. A Comprehensive Set of Tools for Solving Partial Differential Equations; Diffpack, pp. 61–90. *In* Numerical Methods and Software Tools in Industrial Mathematics. Birkhäuser.

[33] Bryant, B. R., Gray, J., and Mernik, M. 2010. Domain-Specific Software Engineering. *In* G.-C. Roman and K. J. Sullivan (eds.), Proceedings of the Workshop on Future of Software Engineering Research (FoSER 2010), pp. 65–68. ACM.

[34] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons.

[35] Campos, L. and Zorzo, S. D. 2007. A Domain Analysis Approach for Engineering RFID Systems in Supply Chain Management. *In* Proceedings of the IEEE International Conference on System of Systems Engineering (SoSE '07), pp. 1–6.

[36] Canann, S. A., Muthukrishnan, S. N., and Phillips, R. K. 1996. Topological Refinement Procedures for Triangular Finite Element Meshes. *Engineering with Computers* 12:243–255.

[37] Canann, S. A., Muthukrishnan, S. N., and Phillips, R. K. 1998. Topological Improvement Procedures for Quadrilateral Finite Element Meshes. *Engineering with Computers* 14:168–177.

[38] Cánovas, J. and Cabot, J. 2012. Creación Colaborativa de Lenguajes Específicos del Dominio. *In* Jornadas de Ingeniería del Software y Bases de Datos.

[39] Capilla, R. and Dueñas, J. C. 2003. Light-Weight Product-Lines for Evolution and Maintenance of Web Site. *In* Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), pp. 53–62. IEEE Computer Society.

[40] Catal, C. 2009. Barriers to the Adoption of Software Product Line Engineering. *ACM SIGSOFT Software Engineering Notes* 34:1–4.

[41] Chastek, G. and McGregor, J. D. 2002. Guidelines for Developing a Product Line Production Plan. Technical Report CMU/SEI-2002-TR-006, Software Engineering Institute.

[42] Chastek, G. (ed.) 2002. Proceedings of the 2nd Software Product Lines Conference (SPLC 2), volume 2379 of *Lecture Notes in Computer Science*. Springer.

[43] Chui, C.-K., Wang, Z., Zhang, J., Ong, J. S.-K., Bian, L., Teo, J. C.-M., Yan, C. H., Ong, S. H., Wang, S.-C., Wong, H.-K., and Teoh, S.-H. 2009. A component-oriented software toolkit for patient-specific finite element model generation. *Advances in Engineering Software* 40:184–192.

[44] Clarke, D., Diakov, N., Hähnle, R., Johnsen, E. B., Puebla, G., Weitzel, B., and Wong, P. Y. H. 2010. HATS - A Formal Software Product Line Engineering Methodology. *In* G. Botterweck, S. Jarzabek, T. Kishi, J. Lee, and S. Livengood (eds.), Workshop Proceedings of the 14th International Software Product Lines Conference (SPLC 2010), pp. 121–128. Lancaster University.

[45] Clarke, D., Diakov, N., Hähnle, R., Johnsen, E. B., Schaefer, I., Schäfer, J., Schlatte, R., and Wong, P. Y. H. 2011. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. *In* M. Bernardo and V. Issarny (eds.), Proceedings of the 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011), volume 6659 of *Lecture Notes in Computer Science*, pp. 417–457. Springer.

[46] Classen, A., Boucher, Q., and Heymans, P. 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming* 76:1130–1143.

[47] Clements, P. 2003. What's the Difference Between Product Line Scope and Product Line Requirements? news@sei.

[48] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. 2002a. Documenting Software Architectures. Views and Beyond. SEI Series in Software Engineering. Addison-Wesley.

[49] Clements, P., Kazman, R., and Klein, M. 2002b. Evaluating Software Architectures. Methods and Case Studies. SEI Series in Software Engineering. Addison-Wesley.

[50] Clements, P. and Northrop, L. 2001. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

[51] Clements, P. C. 2001. On the Importance of Product Line Scope. *In* F. van der Linden (ed.), 4th International Workshop on Software Product-Family Engineering (PFE 2001), volume 2290 of *Lecture Notes in Computer Science*, pp. 70–78. Springer.

[52] Clements, P. C., Jones, L. G., McGregor, J. D., and Northrop, L. M. 2006. Getting There from Here: A Roadmap for Software Product Line Adoption. *Communications of the ACM* 49:33–36.

[53] Cohen, J. D. 1998. Appearance-Preserving Simplification of Polygonal Models. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill.

[54] Cohen, S. G. 2002. Product Line State of the Practice Report. Technical Note CMU/SEI-2002-TN-017, Software Engineering Institute.

[55] Contreras, F. 2007. Adaptación de una Herramienta de Generación de Mallas Geométricas 3D a una Nueva Arquitectura. Computer Science Engineering Thesis. Departamento de Ciencias de la Computación, Universidad de Chile. In Spanish.

[56] Contreras, F., Hitschfeld-Kahler, N., Bastarrica, M. C., and Lillo, C. 2010. Balancing flexibility and performance in three dimensional meshing tools. *Advances in Engineering Software* 41:471–479.

[57] Coplien, J., Hoffman, D., and Weiss, D. M. 1998. Commonality and Variability in Software Engineering. *IEEE Software* 15:37–45.

[58] Czarnecki, K. 2005. Overview of Generative Software Development. *In* J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel (eds.), Unconventional Programming Paradigms, International Workshop (UPP 2004), volume 3566 of *Lecture Notes in Computer Science*, pp. 326–341. Springer.

[59] Czarnecki, K. and Eisenecker, U. W. 1999. Components and Generative Programming. *In* O. Nierstrasz and M. Lemoine (eds.), Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'99), volume 1687 of *Lecture Notes in Computer Science*, pp. 2–19. Springer.

[60] Czarnecki, K. and Eisenecker, U. W. 2000. Generative Programming. Methods, Tools, and Applications. Addison Wesley.

[61] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and Wasowski, A. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. *In* U. W. Eisenecker, S. Apel, and S. Gnesi (eds.), Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012), pp. 173–182. ACM.

[62] Czarnecki, K., Helsen, S., and Eisenecker, U. W. 2004. Staged Configuration Using Feature Models. In [156], pp. 266–283.

[63] Czarnecki, K., Helsen, S., and Eisenecker, U. W. 2005. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice* 10:7–29.

[64] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. 2005. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology* 14:199–245.

[65] de Almeida, E. S. 2007. RiDE: The RiSE Process for Domain Engineering. PhD thesis, Centro de Informática, Federal University of Pernambuco, Brazil.

[66] de Almeida, E. S., Mascena, J. C. C. P., Cavalcanti, A. P. C., Alvaro, A., Garcia, V. C., de Lemos Meira, S. R., and Lucrédio, D. 2006. The Domain Analysis Concept Revisited: A Practical Approach. In [148], pp. 43–57.

[67] de Almeida, E. S., Santos, E. C. R., Alvaro, A., Garcia, V. C., de Lemos Meira, S. R., Lucrédio, D., and de Mattos Fortes, R. P. 2008. Domain Implementation in Software Product Lines Using OSGi. *In* Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008), pp. 72–81. IEEE Computer Society.

[68] DeBaud, J.-M., Flege, O., and Knauber, P. 1998. PuLSE-DSSA–A Method for the Development of Software Reference Architectures. *In* Proceedings of the 3rd International Workshop on Software Architecture (ISAW '98), pp. 25–28. ACM.

[69] DeBaud, J.-M. and Schmid, K. 1999. A Systematic Approach to Derive the Scope of Software Product Lines. *In* Proceedings of the 21st International Conference on Software Engineering (ICSE '99), pp. 34–43. IEEE Computer Society Press.

[70] Deelstra, S., Sinnema, M., and Bosch, J. 2005. Product derivation in software product families: a case study. *Journal of Systems and Software* 74:173–194.

[71] Dhungana, D., Grünbacher, P., Rabiser, R., and Neumayer, T. 2010. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software* 83:1108–1122.

[72] Dhungana, D., Neumayer, T., Grünbacher, P., and Rabiser, R. 2008. Supporting the Evolution of Product Line Architectures with Variability Model Fragments. *In* Proceedings of the 7th Working IEEE / IFIP Conference on Software Architecture (WICSA 2008), pp. 327–330. IEEE.

[73] Díaz, V. 2013. Desarrollo de una Línea de Productos de Software de Generación de Mallas Geométricas. Computer Science Engineering Thesis. Departamento de Ciencias de la Computación, Universidad de Chile. In Spanish.

[74] Dikel, D., Kane, D., Ornburn, S., Loftus, W., and Wilson, J. 1997. Applying Software Product-Line Architecture. *IEEE Computer* 30:49–55.

[75] Djidjev, H. N. 2000. Force-Directed Methods for Smoothing Unstructured Triangular and Tetrahedral Meshes. *In* Proceedings of the 9th International Meshing Roundtable (IMR 2000), pp. 395–406.

[76] Dobrica, L. and Niemelä, E. 2000. A strategy for analysing product line software architectures. VTT Publications 427, Espoo, Technical Research Center of Finland.

[77] Dobrica, L. and Niemelä, E. 2002. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering* 28:638–653.

[78] Douglass, R. W., Carey, G. F., White, D. R., Hansen, G. A., Kallinderis, Y., and Weatherill, N. P. 2002. Current views on grid generation: summaries of a panel discussion. *Numerical Heat Transfer, Part B: Fundamentals* 41:211–237.

[79] Douta, G., Talib, H., Nierstrasz, O., and Langlotz, F. 2009. CompAS: A new approach to commonality and variability analysis with applications in computer assisted orthopaedic surgery. *Information and Software Technology* 51:448–459.

[80] DSM Forum 2006. Why DSM? http://www.dsmforum.org/why.html.

[81] ElSheikh, A. H., Smith, S., and Chidiac, S. E. 2004. Semi-formal design of reliable mesh generation systems. *Advances in Engineering Software* 35:827–841.

[82] Etxeberria, L. and Mendieta, G. S. 2005. Product-Line Architecture: New Issues for Evaluation. In [160], pp. 174–185.

[83] Fabri, A. 2001. CGAL - The Computational Geometry Algorithm Library. *In* Proceedings of the 10th International Meshing Roundtable, pp. 137–142.

[84] Fortune, J., Valerdi, R., Boehm, B. W., and Settles, F. S. 2009. Estimating Systems Engineering Reuse. *In* Proceedings of the 7th Annual Conference on Systems Engineering Research 2009 (CSER 2009).

[85] Frakes, W. B. and Kang, K. 2005. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering* 31:529–536.

[86] Fraser, S. and Mancl, D. 2008. No Silver Bullet: Software Engineering Reloaded. *IEEE Software* 25:91–94.

[87] Frey, P. J. 2001. MEDIT. An interactive mesh visualization software. Technical Report 0253, INRIA.

[88] Frey, P. J. and George, P.-L. 2008. Mesh Generation: Application to Finite Elements. ISTE - John Wiley & Sons, 2nd edition.

[89] Gannod, G. C. and Lutz, R. R. 2000. An Approach to Architectural Analysis of Product Lines. *In* Proceedings of the 22nd International Conference on Software Engineering (ICSE '00), pp. 548–557. ACM Press.

[90] Gorton, I. 2006. Essential Software Architecture. Springer.

[91] Gray, J. and Karsai, G. 2003. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In [98], p. 325a.

[92] Haber, A., Kutz, T., Rendel, H., Rumpe, B., and Schaefer, I. 2011a. Delta-oriented Architectural Variability Using MontiCore. *In* W. Hasselbring and V. Gruhn (eds.), 5th European Conference on Software Architecture (ECSA 2011), Companion Volume, ACM International Conference Proceeding Series, p. 6. ACM.

[93] Haber, A., Rendel, H., Rumpe, B., and Schaefer, I. 2011b. Delta Modeling for Software Architectures. *In* H. Giese, M. Huhn, J. Phillips, and B. Schätz (eds.), Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, pp. 1–10. fortiss GmbH, München.

[94] Haber, A., Rendel, H., Rumpe, B., Schaefer, I., and van der Linden, F. 2011c. Hierarchical Variability Modeling for Software Architectures. *In* E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid (eds.), Proceedings of the 15th International Software Product Line Conference (SPLC 2011), pp. 150–159. IEEE.

[95] Harel, D. and Rumpe, B. 2004. Meaningful Modeling: What's the Semantics of "Semantic"? *Computer* 37:64–72.

[96] Heering, J., Mernik, M., and Sloane, A. M. 2003. Domain-Specific Languages. In [98], p. 323.

[97] Herskovic, V., Ochoa, S. F., Pino, J. A., and Neyem, A. 2011. The Iceberg Effect: Behind the User Interface of Mobile Collaborative Systems. *Journal of Universal Computer Science* 17:183–202.

[98] HICSS 2003. Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03). IEEE Computer Society.

[99] Hitschfeld, N., Lillo, C., Cáceres, A., Bastarrica, M. C., and Rivara, M. C. 2006. Building a 3D Meshing Framework Using Good Software Engineering Practices. In [161], pp. 162–170.

[100] Hitschfeld, N., Villablanca, L., Krause, J., and Rivara, M. C. 2003. Improving the quality of meshes for the simulation of semiconductor devices using Lepp-based algorithms. *International Journal for Numerical Methods in Engineering* 58:333–347.

[101] Hitschfeld-Kahler, N. 2005. Generation of 3D mixed element meshes using a flexible refinement approach. *Engineering with Computers* 21:101–114.

[102] Hunt, J. M. 2006. Organizing the Asset Base for Product Derivation. In [218], pp. 65–74.

[103] IMR 1996. Proceedings of the 5th International Meshing Roundtable (IMR 1996). Sandia National Lab.

[104] International Meshing Roundtable 2011. International Meshing Roundtable. http://www.imr.sandia.gov/. Accessed on March.

[105] International Society of Grid Generation 2011. Mesh Generation Definition. http://morden.csee.usf.edu/dragon/kpalbrec/isgg.html. Accessed on June.

[106] ISO 9126-1 1998. FCD 9126-1.2 Information Technology - Software product quality - Part 1: Quality model.

[107] Janota, M., Kiniry, J., and Botterweck, G. 2008. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero.

[108] Jazayeri, M., Ran, A., , and van der Linden, F. 2000. Software Architecture for Product Families: Principles and Practice. Addison-Wesley.

[109] John, I. and Eisenbarth, M. 2009. A Decade of Scoping - A Survey. In [150], pp. 31–40.

[110] John, I., Knodel, J., Lehner, T., and Muthig, D. 2006. A Practical Guide to Product Line Scoping. In [218], pp. 3–12.

[111] John, I., Muthig, D., Sody, P., and Tolzmann, E. 2002. Efficient and Systematic Software Evolution Through Domain Analysis. *In* Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE'02), p. 237. IEEE Press.

[112] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. 2010. ABS: A Core Language for Abstract Behavioral Specification. *In* B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue (eds.), Proceedings of the 9th International Symposium for Formal Methods for Components and Objects (FMCO 2010), volume 6957 of *Lecture Notes in Computer Science*, pp. 142–164. Springer.

[113] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. 2008. Atl: A model transformation tool. *Science of Computer Programming* 72:31–39.

[114] Kaindl, H. 2000. A Design Process Based on a Model Combining Scenarios with Goals and Functions. *IEEE Transactions on Systems, Man, and Cybernetics–Part A: Systems and Humans* 30:537–551.

[115] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. 1990. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute.

[116] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering* 5:143–168.

[117] Kasunic, M. 2008. A Data Specification for Software Project Performance Measures: Results of a Collaboration on Performance Measurement. Technical Report CMU/SEI-2008-TR-012, Software Engineering Institute.

[118] Kelly, D., Smith, S., and Meng, N. 2011. Software Engineering for Scientists. *Computing in Science and Engineering* 13:7–11.

[119] Kim, J., Kim, M., and Park, S. 2006a. Goal and scenario based domain requirements analysis environment. *Journal of Systems and Software* 79:926–938.

[120] Kim, J., Park, S., and Sugumaran, V. 2006b. Improving use case driven analysis using goal and scenario authoring: A linguistics-based approach. *Data & Knowledge Engineering* 58:21–46.

[121] Kim, J., Park, S., and Sugumaran, V. 2008. DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines. *Journal of Systems and Software* 81:37–55.

[122] Kinney, P. 1997. Clean-Up: Improving Quadrilateral Finite Element Meshes. *In* Proceedings of the 6th International Meshing Roundtable (IMR 1997), pp. 437–447.

[123] Knauber, P., Muthig, D., Schmid, K., and Widen, T. 2000. Applying Product Line Concepts in Small and Medium-Sized Companies. *IEEE Software* 17:88–95.

[124] Knupp, P. M. 1999. Winslow Smoothing on Two-Dimensional Unstructured Meshes. *Engineering with Computers* 15:263–268.

[125] Korhonen, M. and Mikkonen, T. 2004. Assessing systems adaptability to a product family. *Journal of Systems Architecture* 50:383–392.

[126] Krueger, C. 2002. Eliminating the Adoption Barrier. *IEEE Software* 19:29–31.

[127] Krueger, C. W. 2006. New Methods in Software Product Line Development. In [218], pp. 95–99.

[128] Krueger, C. W. 2007. The 3-Tiered Methodology: Pragmatic Insights from New Generation Software Product Lines. In [219], pp. 97–106.

[129] Lee, K., Kang, K. C., and Lee, J. 2002. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. *In* C. Gacek (ed.), Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools (ICSR-7), volume 2319 of *Lecture Notes in Computer Science*, pp. 62–77. Springer.

[130] Livengood, S. 2011. Issues in Software Product Line Evolution: Complex Changes in Variability Models. *In* Proceedings of the 2nd International Workshop on Product LinE Approaches in Software Engineering (PLEASE 2011), pp. 6–9. ACM.

[131] Lobos, C., Bucki, M., Hitschfeld-Kahler, N., and Payan, Y. 2007. Mixed-element Mesh for an Intra-operative Modeling of the Brain Tumor Extraction. *In* M. L. Brewer and D. L. Marcum (eds.), Proceedings of the 16th International Meshing Roundtable (IMR 2007), pp. 387–404. Springer.

[132] Losilla, F., Vicente-Chicote, C., Álvarez, B., Iborra, A., and Sánchez, P. 2007. Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach. *In* F. Oquendo (ed.), Proceedings of the First European Conference on Software Architecture (ECSA 2007), volume 4758 of *Lecture Notes in Computer Science*, pp. 179–194. Springer.

[133] Luoma, J., Kelly, S., and Tolvanen, J.-P. 2004. Defining Domain-Specific Modeling Languages: Collected Experiences. *In* J.-P. Tolvanen, J. Sprinkle, and M. Rossi (eds.), Proceedings of the 4th OOPSLA Workshop on Domain-specific Modeling (DSM' 04), pp. 1–10.

[134] Macala, R. R., Stuckey Jr., L. D., and Gross, D. C. 1996. Managing Domain-Specific, Product-Line Development. *IEEE Software* 13:57–67.

[135] Maccari, A. 2002. Experiences in assessing product family software architecture for evolution. *In* Proceedings of the 24th International Conference on Software Engineering (ICSE '02), pp. 585–592. ACM Press.

[136] McGregor, J. D., Northrop, L. M., Jarrad, S., and Pohl, K. 2002. Guest Editors' Introduction: Initiating Software Product Lines. *IEEE Software* 19:24–27.

[137] Medina, R., Hitschfeld-Kahler, N., Forest, L., Padilla, F., Martínez, S., and Martín, J. S. 2006. Geometric Modelling of Tree Stem Deformation. *In* Abstract Proceedings of the 5th Symposium on Trends in Unstructured Mesh Generation.

[138] Medvidovic, N., Dashofy, E. M., and Taylor, R. N.. 2007. Moving architectural description from under the technology lamppost. *Information and Software Technology* 49:12–31.

[139] Medvidovic, N. and Taylor, R. 1998. Separating Fact from Fiction in Software Architecture. *In* Proceedings of the Third International Workshop on Software Architecture, ISAW '98, pp. 105–108. ACM Press.

[140] Medvidovic, N. and Taylor, R. N.. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26:70–93.

[141] Melo, C. 2008. Desarrollo de una Herramienta que Genera Mallas de Superficie Compuestas de Cuadriláteros para Modelar el Crecimiento de Arboles. Computer Science Engineering Thesis. Departamento de Ciencias de la Computación, Universidad de Chile. In Spanish.

[142] Mernik, M., Heering, J., and Sloane, A. M. 2005. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37:316–344.

[143] Michalik, B., Weyns, D., and Betsbrugge, W. V. 2011. On the Problems with Evolving Egemin's Software Product Line. *In* Proceedings of the 2nd International Workshop on Product LinE Approaches in Software Engineering (PLEASE 2011), pp. 15–19. ACM.

[144] Miller, G. L., Talmor, D., and Teng, S.-H. 1997. Optimal Good-Aspect-Ratio Coarsening for Unstructured Meshes. *In* Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97), pp. 538–547. Society for Industrial and Applied Mathematics.

[145] Miller, G. L., Talmor, D., and Teng, S.-H. 1999. Optimal Coarsening of Unstructured Meshes. *Journal of Algorithms* 31:29–65.

[146] Miller, G. L., Talmor, D., Teng, S.-H., Walkington, N., and Wang, H. 1996. Control Volume Meshes using Sphere Packing: Generation, Refinement and Coarsening. In [103], pp. 47–61.

[147] Mobley, A. V., Tristano, J. R., and Hawkings, C. M. 2001. An Object-Oriented Design for Mesh Generation and Operation Algorithms. *In* Proceedings of the 10th International Meshing Roundtable, pp. 179–183.

[148] Morisio, M. (ed.) 2006. 9th International Conference on Software Reuse: Reuse of Off-the-Shelf Components (ICSR 2006), volume 4039 of *Lecture Notes in Computer Science*. Springer.

[149] Muccini, H. and Bucchiarone, A. 2004. Formal Behavioral Specification of a Product Line Architecture. Technical Report TRCS 014/2004, University of L'Aquila.

[150] Muthig, D. and McGregor, J. D. (eds.) 2009. Proceedings of the 13th International Software Product Line Conference (SPLC 2009), volume 446 of *ACM International Conference Proceeding Series*. ACM.

[151] Nedstam, J. and Staples, M. 2007. Evolving Strategies for Software Architecture and Reuse. *Software Process: Improvement and Practice* 12:295 – 309.

[152] Neighbors, J. M. 1981. Software Construction Using Components. PhD thesis, Department of Information and Computer Science, University of California, Irvine.

[153] Nestor, D., Thiel, S., Botterweck, G., Cawley, C., and Healy, P. 2008. Applying visualisation techniques in software product lines. *In* R. Koschke, C. D. Hundhausen, and A. Telea (eds.), Proceedings of the ACM 2008 Symposium on Software Visualization (SOFTVIS 2008), pp. 175–184. ACM.

[154] Niemelä, E. and Immonen, A. 2007. Capturing quality requirements of product family architecture. *Information and Software Technology* 49:1107–1120.

[155] Niemelä, E., Kalaoja, J., and Lago, P. 2005. Toward an Architectural Knowledge Base for Wireless Service Engineering. *IEEE Transactions on Software Engineering* 31:361–379.

[156] Nord, R. L. (ed.) 2004. Proceedings of the 3rd International Software Product Lines Conference (SPLC 2004), volume 3154 of *Lecture Notes in Computer Science*. Springer.

[157] Northrop, L. and Clements, P. 2007. A Framework for Software Product Line Practice. Version 5.0. http://www.sei.cmu.edu/productlines/framework.html.

[158] Northrop, L. M. 2004. Software Product Line Adoption Roadmap. Technical Report CMU/SEI-2004-TR-022, Software Engineering Institute.

[159] Nunes, I., de Lucena, C. J. P., Kulesza, U., and Nunes, C. 2009. On the Development of Multi-agent Systems Product Lines: A Domain Engineering Process. *In* J. J. Gomez-Sanz and M.-P. Gleizes (eds.), Proceedings of the 10th International Workshop on Agent-Oriented Software Engineering (AOSE 2009), pp. 109–120.

[160] Obbink, J. H. and Pohl, K. (eds.) 2005. Proceedings of the 9th International Software Product Line Conference (SPLC 2005), volume 3714 of *Lecture Notes in Computer Science*. Springer.

[161] Ochoa, S. F. and Roman, G.-C. (eds.) 2006. IFIP 19th World Computer Congress, First International Workshop on Advanced Software Engineering, Expanding the Frontiers of Software Technology, volume 219 of *IFIP International Federation for Information Processing*, Santiago, Chile. Springer Boston.

[162] O'Leary, P. 2010. Towards a Product Derivation Process Reference Model for Software Product Line Organisations. PhD thesis, Faculty of Science and Engineering, University of Limerick, Ireland.

[163] O'Leary, P., Rabiser, R., Richardson, I., and Thiel, S. 2009. Important Issues and Key Activities in Product Derivation: Experiences from Two Independent Research Projects. In [150], pp. 121–130.

[164] Olumofin, F. G. and Mišić, V. B. 2007. A holistic architecture assessment method for software product lines. *Information and Software Technology* 49:309–323.

[165] Owen, S. J. 1998. A Survey of Unstructured Mesh Generation Technology. *In* Proceedings of the 7th International Meshing Roundtable (IMR 1998), pp. 239–267. Sandia National Lab.

[166] Panthaki, M., Sahu, R., and Gerstle, W. 1997. An Object-Oriented Virtual Geometry Interface. *In* Proceedings of the 6th Annual International Meshing Roundtable, pp. 67–81.

[167] Park, S., Kim, M., and Sugumaran, V. 2004. A scenario, goal and feature-oriented domain analysis approach for developing software product lines. *Industrial Management & Data Systems* 104:296–308.

[168] Parnas, D. L. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* 2:1–9.

[169] Perovich, D., Rossel, P. O., and Bastarrica, M. C. 2009. Feature Model to Product Architectures: Applying MDE to Software Product Lines. *In* Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (WICSA/ECSA 2009), pp. 201–210. IEEE.

[170] Phillips, M., Levy, S., and Munzner, T. 2007. Geomview Manual.

[171] Plaza, A. and Carey, G. F. 2000. Local refinement of simplicial grids based on the skeleton. *Applied Numerical Mathematics* 32:195–218.

[172] Plaza, A., Márquez, A., Moreno-González, A., and Suárez, J. P. 2009. Local refinement based on the 7-triangle longest-edge partition. *Mathematics and Computers in Simulation* 79:2444–2457.

[173] Plaza, A. and Rivara, M. C. 2002. On the adjacencies of triangular meshes based on skeleton-regular partitions. *Journal of Computational and Applied Mathematics* 140:673–693.

[174] Pohl, K., Böckle, G., and van der Linden, F. 2005. Software Product Line Engineering. Foundations, Principles, and Techniques. Springer.

[175] Pohl, K. and Metzger, A. 2006. Software Product Line Testing. *Communications of the ACM* 49:78–81.

[176] Prehofer, C. 2001. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience* 13:465–501.

[177] Pressman, R. S. 2009. Software Engineering: A Practitioner's Approach. McGraw-Hill, 7th edition.

[178] Prieto-Díaz, R. 1990. Domain Analysis: An Introduction. *SIGSOFT Software Engineering Notes* 15:47–54.

[179] Rabiser, R. and Dhungana, D. 2007. Integrated Support for Product Configuration and Requirements Engineering in Product Derivation. *In* Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2007), pp. 219–228. IEEE Computer Society.

[180] Rabiser, R., Grünbacher, P., and Dhungana, D. 2007. Supporting Product Derivation by Adapting and Augmenting Variability Models. In [219], pp. 141–150.

[181] Remacle, J.-F. and Shephard, M. S. 2003. An Algorithm Oriented Mesh Database. *International Journal for Numerical Methods in Engineering* 58:349–374.

[182] Riebisch, M., Streitferdt, D., and Philippow, I. 2001. Feature Scoping for Product Lines. *In* K. Schmid and B. Geppert (eds.), Proceedings of the International Workshop on Product Line Engineering (PLEES'01). Fraunhofer IESE.

[183] Rivara, M. C. 1996. New Mathematical Tools and Techniques for the Refinementand/or Improvement of Unstructured Triangulations. In [103], pp. 77–86.

[184] Rivara, M. C. 1997. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering* 40:3313–3324.

[185] Rivara, M.-C. 2009. Lepp-bisection algorithms, applications and mathematical properties. *Applied Numerical Mathematics* 59:2218–2235.

[186] Rivara, M.-C. and Hitschfeld, N. 1999. LEPP - Delaunay Algorithm: a Robust Tool for Producing Size-Optimal Quality Triangulations. *In* Proceedings of the 8th International Meshing Roundtable (IMR 1999), pp. 205–220.

[187] Rolland, C., Souveyet, C., and Achour, C. B. 1998. Guiding Goal Modeling Using Scenarios. *IEEE Transactions on Software Engineering* 24:1055–1071.

[188] Rossel, P. O., Bastarrica, M. C., and Hitschfeld-Kahler, N. 2009a. A Systematic Process for Defining Meshing Tool Software Product Line Domain Model. *In* C. Ayala, C. Silva, and H. Astudillo (eds.), Proceedings of the 12th Workshop on Requirements Engineering (WER'09), pp. 103–114.

[189] Rossel, P. O., Perovich, D., and Bastarrica, M. C. 2009b. Reuse of architectural Knowledge in SPL Development. *In* S. H. Edwards and G. Kulczycki (eds.), Proceedings of the 11th International Conference on Software Reuse (ICSR 2009), volume 5791 of *Lecture Notes in Computer Science*, pp. 191–200. Springer.

[190] Ruppert, J. 1995. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *Journal of Algorithms* 18:548–585.

[191] Russell, S. and Norvig, P. 2009. Artificial Intelligence: A Modern Approach. Prentice Hall, 3rd edition.

[192] Schaefer, I. and Hähnle, R. 2011. Formal Methods in Software Product Line Engineering. *IEEE Computer* 44:82–85.

[193] Schmid, K. 1999. An Economic Perspective on Product Line Software Development. *In* First Workshop on Economics-Driven Software Engineering Research (EDSER-1).

[194] Schmid, K. 2000. Scoping Software Product Lines. *In* P. Donohoe (ed.), Proceedings of the 1st International Software Product Lines Conference (SPLC 1), pp. 513–532. Kluwer.

[195] Schmid, K. 2002. A Comprehensive Product Line Scoping Approach and its Validation. *In* Proceedings of the 24th International Conference on Software Engineering (ICSE '02), pp. 593–603. ACM.

[196] Schmid, K., Thiel, S., Bosch, J., Johnsson, S., Jaring, M., Thomé, B., and Trosch, S. 2001. Scoping. ESAPS Public Results. http://www.esi.es/esaps/public-pdf/CWD124-08-06-01.pdf.

[197] Schmid, K. and Verlage, M. 2002. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software* 19:50–57.

[198] Schmid, K. and Widen, T. 2000. Customizing the PuLSE™ Product Line Approach to the Demands of an Organization. *In* R. Conradi (ed.), Proceedings of the 7th European Workshop on Software Process Technology (EWSPT 2000), volume 1780 of *Lecture Notes in Computer Science*, pp. 221–238. Springer.

[199] Schmidt, D. C. 2006. Model-Driven Engineering. *Computer* 39:25–31.

[200] Schneiders, R. 2011. Mesh Generation and Grid Generation: Software. http://www.robertschneiders.de/meshgeneration/software.html. Accessed on June.

[201] SEI 2011a. Software Architecture Glossary. http://www.sei.cmu.edu/architecture/-start/glossary. Accessed on March.

[202] SEI 2011b. Software Product Lines Glossary. http://www.sei.cmu.edu/productlines/-start/glossary. Accessed on March.

[203] Shaw, M. and Clements, P. C. 2006. The Golden Age of Software Architecture. *IEEE Software* 23:31–39.

[204] Shaw, M. and Garlan, D. 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc.

[205] Shepherd, J. F., Dewey, M. W., Woodbury, A. C., Benzley, S. E., Staten, M. L., and Owen, S. J. 2010. Adaptive mesh coarsening for quadrilateral and hexahedral meshes. *Finite Elements in Analysis and Design* 46:17–32.

[206] Shewchuk, J. R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. *In* Proceedings of the 1st ACM Workshop on Applied Computational Geometry, volume 1148 of *Lecture Notes in Computer Science*, pp. 203–222. Springer.

[207] Shewchuk, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry* 22:21–74.

[208] Si, H. 2011a. TetGen. A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator. http://tetgen.org. Accessed on December.

[209] Si, H. 2011b. TetView. A Tetrahedral Mesh and Piecewise Linear Complex Viewer. http://tetgen.org/tetview.html. Accessed on December.

[210] Silva, N. 2007. Modelamiento del Crecimiento de Arboles Usando Mallas de Superficie. Computer Science Engineering Thesis. Departamento de Ciencias de la Computación, Universidad de Chile. In Spanish.

[211] Smith, S. 2006. Systematic Development of Requirements Documentation for General Purpose Scientific Computing Software. *In* Proceedings of the 14th IEEE International Conference on Requirements Engineering (RE 2006), pp. 205–215. IEEE Computer Society.

[212] Smith, S. and Chen, C.-H. 2004a. Commonality Analysis for Mesh Generating Systems. Technical Report CAS-04-10-SS, Department of Computing and Software, McMaster University, Canada.

[213] Smith, S. and Chen, C.-H. 2004b. Commonality and Requirements Analysis for Mesh Generating Software. *In* Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), pp. 384–387.

[214] Smith, S., McCutchan, J., and Cao, F. 2007. Program Families in Scientific Computing. *In* J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen (eds.), Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), pp. 39–47.

[215] Smith, S. and Yu, W. 2009. A document driven methodology for developing a high quality Parallel Mesh Generation Toolbox. *Advances in Engineering Software* 40:1155–1167.

[216] Smith, W. S., Carette, J., and McCutchan, J. 2008. Commonality Analysis of Families of Physical Models for use in Scientific Computing. *In* Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE08).

[217] Sommerville, I. 2010. Software Engineering. Addison Wesley, 9th edition.

[218] SPLC 2006. Proceedings of the 10th International Software Product Line Conference (SPLC 2006). IEEE Computer Society.

[219] SPLC 2007. Proceedings of the 11th International Software Product Lines Conference (SPLC 2007). IEEE Computer Society.

[220] Sprinkle, J., Mernik, M., Tolvanen, J.-P., and Spinellis, D. 2009. Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software* 26:15–18.

[221] Staten, M. L. and Canann, S. A. 1997. Post Refinement Element Shape Improvement for Quadrilateral Meshes. *In* S. Canann and S. Saigal (eds.), Proceedings of the Symposium on Trends in Unstructured Mesh Generation (AMD-Vol 220), pp. 9–16.

[222] Strembeck, M. and Zdun, U. 2009. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience* 39:1253–1292.

[223] Suárez, J. P., Plaza, A., and Carey, G. F. 2003. Estructuras de datos geométricas para algoritmos de refinamiento basados en el esqueleto. *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería* 19:89–109.

[224] Svahnberg, M., van Gurp, J., and Bosch, J. 2005. A taxonomy of variability realization techniques. *Software: Practice and Experience* 35:705–754.

[225] Tautges, T. J. 2000. The Common Geometry Module (CGM): A Generic, Extensible, Geometry Interface. *In* Proceedings of the 9th International Meshing Roundtable, pp. 337–347.

[226] Taylor, R. N., Dashofy, E. M., and Medvidovic, N. 2009. Software Architecture. Foundations, Theory, and Practice. John Wiley & Sons.

[227] Teng, S.-H. 1998. Provably Good Partitioning and Load Balancing Algorithms for Parallel Adaptive N-Body Simulation. *SIAM Journal on Scientific Computing* 19:635–656.

[228] Teng, S.-H. and Wong, C. W. 2000. Unstructured Mesh Generation: Theory, Practice, and Perspectives. *International Journal of Computational Geometry and Applications* 10:227–266.

[229] Thao, C. 2012. Managing Evolution of Software Product Line. *In* M. Glinz, G. C. Murphy, and M. Pezzè (eds.), Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), pp. 1619–1621. IEEE.

[230] Thiel, S., Ferber, S., Fischer, T., Hein, A., and Schlick, M. 2001. A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems. *In* Proceedings of In-Vehicle Software 2001 (SP-1587), pp. 43–55.

[231] Tolvanen, J.-P. and Kelly, S. 2005. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In [160], pp. 198–209.

[232] Trigaux, J.-C. and Heymans, P. 2003. Software Product Lines: State of the art. Technical Report EPH3310300R0462/215315, FUNDP Insitut d'Informatique Namur.

[233] Trinidad, P., Benavides, D., and Ruiz-Cortés, A. 2004. Improving Decision Making in Software Product Lines Product Plan Management. *In* Proceedings of the V ADIS 2004 Workshop on Decision Support in Software Engineering.

[234] van der Linden, F. 2002. Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software* 19:41–49.

[235] van der Linden, F., Bosch, J., Kamsties, E., Känsälä, K., and Obbink, J. H. 2004. Software Product Family Evaluation. In [156], pp. 110–129.

[236] van der Linden, F., Schmid, K., and Rommes, E. 2007. Software Product Lines in Action. Springer.

[237] van Deursen, A. and Klint, P. 1998. Little Languages: Little Maintenance? *Journal of Software Maintenance* 10:75–92.

[238] van Deursen, A. and Klint, P. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology* 10:1–17.

[239] van Deursen, A., Klint, P., and Visser, J. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35:26–36.

[240] van Ommering, R. C. and Bosch, J. 2002. Widening the Scope of Software Product Lines - From Variation to Composition. In [42], pp. 328–347.

[241] van Ommering, R. C., van der Linden, F., Kramer, J., and Magee, J. 2000. The Koala Component Model for Consumer Electronics Software. *IEEE Computer* 33:78–85.

[242] Čeh, I., Črepinšek, M., Kosar, T., and Mernik, M. 2011. Ontology driven development of domain-specific languages. *Computer Science and Information Systems* 8:317–342.

[243] Vicente-Chicote, C., Moros, B., and Toval, A. 2007. REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting. *Journal of Object Technology* 6:437–454.

[244] Visser, E. 2008. WebDSL: A Case Study in Domain-Specific Language Engineering. *In* R. Lämmel, J. Visser, and J. ao Saraiva (eds.), Generative and Transformational Techniques in Software Engineering II (GTTSE 2007), volume 5235 of *Lecture Notes in Computer Science*, pp. 291–373. Springer.

[245] Vollmer, J., Mencl, R., and Müller, H. 1999. Improved laplacian smoothing of noisy surface meshes. *Computer Graphics Forum* 18:131–138.

[246] Weiss, D. M. 1998. Commonality Analysis: A Systematic Process for Defining Families. *In* F. van der Linden (ed.), Proceedings of the 2nd International ESPRIT ARES Workshop, volume 1429 of *Lecture Notes in Computer Science*, pp. 214–222. Springer.

[247] Weiss, D. M. and Lai, C. T. R. 1999. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley.

[248] Yang, Y.-J., Yong, J.-H., and Sun, J.-G. 2005. An algorithm for tetrahedral mesh generation based on conforming constrained Delaunay tetrahedralization. *Computers & Graphics* 29:606–615.

[249] Yu, W. and Smith, S. 2009. Reusability of FEA Software: A Program Family Approach. *In* Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE '09), pp. 43–50. IEEE Computer Society.

[250] Yu, Y., do Prado Leite, J. C. S., Lapouchnian, A., and Mylopoulos, J. 2008. Configuring Features with Stakeholder Goals. *In* R. L. Wainwright and H. Haddad (eds.), Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), pp. 645–649. ACM.

[251] Zhang, H. and Jarzabek, S. 2004. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming* 53:381–407.

Appendices

Appendix A

Meshing Tool Derivation Process

The product derivation is an important process of any SPL development process, because it permits obtaining the particular products of the line. In this sense, the specific steps to be followed by the stakeholders must be clearly specified, as well as the relationships among the processes of the SPL development process, such as domain analysis, domain design, among others.

In the following sections, a product derivation process suitable for meshing tool domain is presented, considering products that are inside and outside the SPL scope.

A.1 Introduction

A specific domain has several sub-domains inside of it, and each of them has some associated functionalities. Additionally, it is relatively easy to see that in a domain there are several software products that usually do not cover all the functionalities present in each sub-domain. Figure A.1 shows the existing relationship between sub-domains and products in a particular domain [194] .

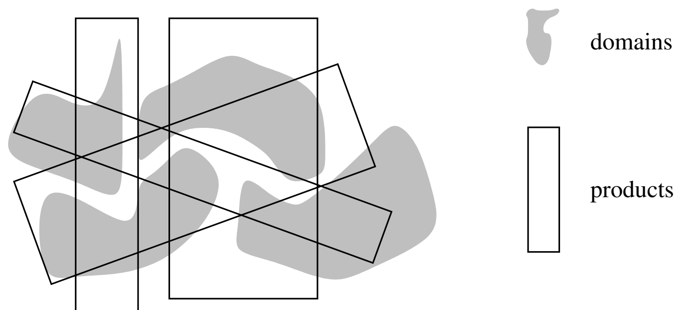


Figure A.1: Relationship between Domains and Products

The product line is composed of the combination of all products of the domain, and

several sub-domains are necessary for having the functionalities that compose the products.

With the above in mind, it is easy to understand the difficulties that specialists and stakeholders have for producing software products. Indeed, as presented in Chapter 2, SPL developments have big challenges, and to achieve the goals imposed by those challenges it is necessary to count on a suitable infrastructure. Furthermore, an appropriate model supporting the reuse infrastructure is needed.

In the context of SPL, when a new product is developed (or derived), all stages of the application engineering process are used. Therefore, all reuse infrastructure, core assets and underlying process model are ready for carrying out this activity. Product derivation is the complete process of constructing a product from product family software assets [70], and its goal is to build the product by reusing the reuse infrastructure as much as possible and minimizing the amount of product-specific development required [163].

`Product derivation purpose`

In general, a product derivation (or instantiation) process could be in a range from manual to automatic: all intermediate values are allowed. In that sense, Figure 2.6 (in Section 2.5) is valid. It is important to put attention in the derivation process, and consider that a tool may help increase efficiency and deal with product derivation complexity [163]. With an automatic o semiautomatic derivation process, it is more likely to have components integration activity or configuration scripts than a complete development process.

`Product derivation process`

In general, any derivation process begins with a customer request that can be satisfied by the SPL. This is the case when the requested product is in the scope of the SPL, i.e the SPL is able to produce the required product. The specification for the requested product is built. In some developments that use feature models, a feature model configuration is derived. Then, the product architecture is instantiated according to the reference architecture, product specification and feature configuration. Finally, from the product architecture and core assets produced in the domain implementation, the product is assembled. Generally, an integration test and acceptance test is carried out before the product is delivered to users. Figure A.2 summarizes this [180]. Irrelevant parts of the original variability model must be pruned, and additional project-specific knowledge needs to be incorporated to the model, adapting and augmenting it.

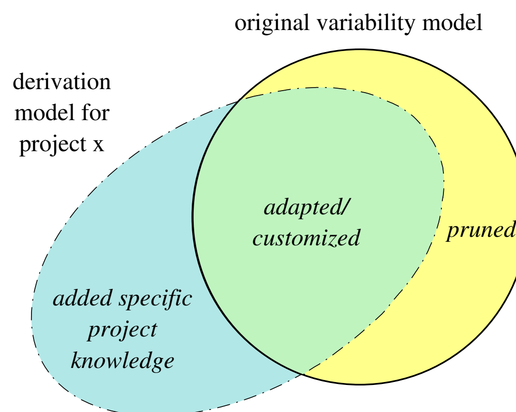


Figure A.2: Project-specific derivation models

According to O'Leary et al. [163] there are several key activities to be considered in a product derivation process:

Product derivation activities

- Preparing for Derivation: Due to the need of collaboration between customers and developers, customer requirements need to be translated into internal organizational languages. Furthermore, it is necessary to define roles, e.g. who is in charge of product derivation to fulfill product requirements. Finally, there must be some guidance for decision-makers.

- Product Derivation/Configuration: Product derivation is an iterative process that starts with the selection and/or customization of the core assets. Sometimes it is impossible to resolve all variabilities in one single step, and it is necessary to build a configuration that partially implements the software product. Then, additional development will be required.

- Additional Development/Testing: When a component is developed or adapted due to new requirements, then rigorous testing is necessary. The component must be integrated with the rest of the components of the reuse infrastructure.

Some authors [70, 153, 163] agree with the idea of having several activities inside of the product derivation process. Deelstra et al. [70] established a generic process for derivation that consists of two phases: an initial and an iterative one. Sometimes the initial phase is enough to produce a particular product. However, in many cases one or more cycles of the iterative phase are required.

Product derivation phases

1. Initial phase: The input to this phase is a set of customer requirements. With these, a first derivation is produced, considering three alternatives: assembly, which involves the assembly of the subset of shared core assets to the initial software product configuration; configuration selection, which involves the selection of a closest matching existing configuration; and hybrid, which is a mix of the previous two alternatives. The initial phase concludes with an initial validation of the built product for determining if it adheres to the requirements.

2. Iterative phase: Many times, the initial derivation obtained from the previous phase does not implement the desired product. Then, several cycles through the iterative phase are needed to produce the correct product. This may happen due to the following reasons: requirements may change during derivation; the configuration may not completely provide the required functionality or some of the selected components simply can not work together at all; or the core assets used to derive the configuration may have changed during product derivation. During the iterative phase, the product configuration is modified and validated until the product is ready.

The product derivation process presented by Deelstra et al. [70] is generic and applicable to any domain. However, there are few approaches and tools available for product derivation [162, 179]. It is possible to find some approaches that are particular to a SPL approach.

Some of them are PuLSE-I [23], and DOPLER^{UCon} [180, 179]. A good review was conducted by O'Leary [162] and more approaches can be found there. Moreover, some tools have been developed by Nestor et al. [153] and Rabiser et al. [180] for helping in the production of the software products. Finally, O'Leary [162] presents a generic approach for product derivation.

Because product derivation is a process prone to errors, it is convenient to have tools that support the product derivation process. The tools must deal with [153]:

- The complexity of the SPL in terms of variation points, variants and dependencies;
- The large number of implicit properties or dependencies associated with variation points and variants.

Furthermore, as Hunt [102] established, it is necessary to determine the set of assets that will be used to build specific product. If the size and complexity of the SPL grow and with it the reuse infrastructure, then it will be difficult to derive products from the infrastructure. It is necessary, as Hunt manifests, to organize the core assets according to some criteria:

- Natural division.
- Easy to find components.
- Generally applicable.
- Reasonably sized groups.
- Similarly sized groups.

In summary, a product derivation process must be specific enough to be useful but not too specific, but finding the right balance between specific and generic is a hard task [163]. This thesis is in the same line with this idea and proposes a product derivation process considering both domain engineering and application engineering phases. The process is showed in the following Section.

A.2 Products Derivation Process

A.2.1 Considerations

The product derivation process that will be presented in the following sections is not a complete or standard derivation process. This is because a complete derivation process is not part of this thesis (see Section 1.2 and Figure 1.3). However, the diagrams are shown

for completeness, i.e., it is better to show the complete process to understand its stages and their interactions, what is the real sequence for instantiating a product, and for future works.

The derivation process that will be presented in the following section corresponds to the “Preparing for Derivation” activity, and half of the “Product Derivation/Configuration” activity proposed by O’Leary et al. [163], and presented in Section A.1. The third activity “Additional Development/Testing” is not considered because the SPL model of this thesis does not produce final software products.

A.2.2 Activity Diagram

The product derivation process is depicted in Figure A.3. This diagram includes both the domain engineering and application engineering phases and their stages. This process is very similar to any other process of product derivation. In this sense, the presented process is general enough to use it as a guide for derivation in other domains but specific to the meshing tool domain, since it considers the necessary inputs and outputs for it at each stage.

Chapters 4 and 6 presented in more detail each one of the domain engineering stages. Application engineering stages will be commented in the following sections.

A.2.3 Applications in the SPL Scope

The product derivation process begins when a customer requests a product that can be built by the SPL, that is, the required product is within the scope of the SPL.

The derivation process starts with activity ❶ of Figure A.3: “Check Requirements in relation to SPL Scope”. This consists of five other activities presented in Figure A.4.

The product analysts review wether both the business goal and the feature model cover the current needs of the product. Because the product required is within the scope, the product map is verified and updated with the product needs, and a feature model configuration is established according to customer needs. These artifacts were detailed in Chapter 4.

The feature configuration model is used as input for the application design stage. In this stage, the product designer needs to check if all domain design artifacts, i.e. product line architecture, state machine, production plan and transformation rules, were built during the domain design stage. This check occurs in activity ❷: “Check Domain Design Artifacts Built”. It is highly probable that all these artifact exist in the Reuse Infrastructure artifacts since the product is in the scope.

Then, the product designer proceeds with the “Application Design Instantiation” activity (❸), which is shown in Figure A.5.

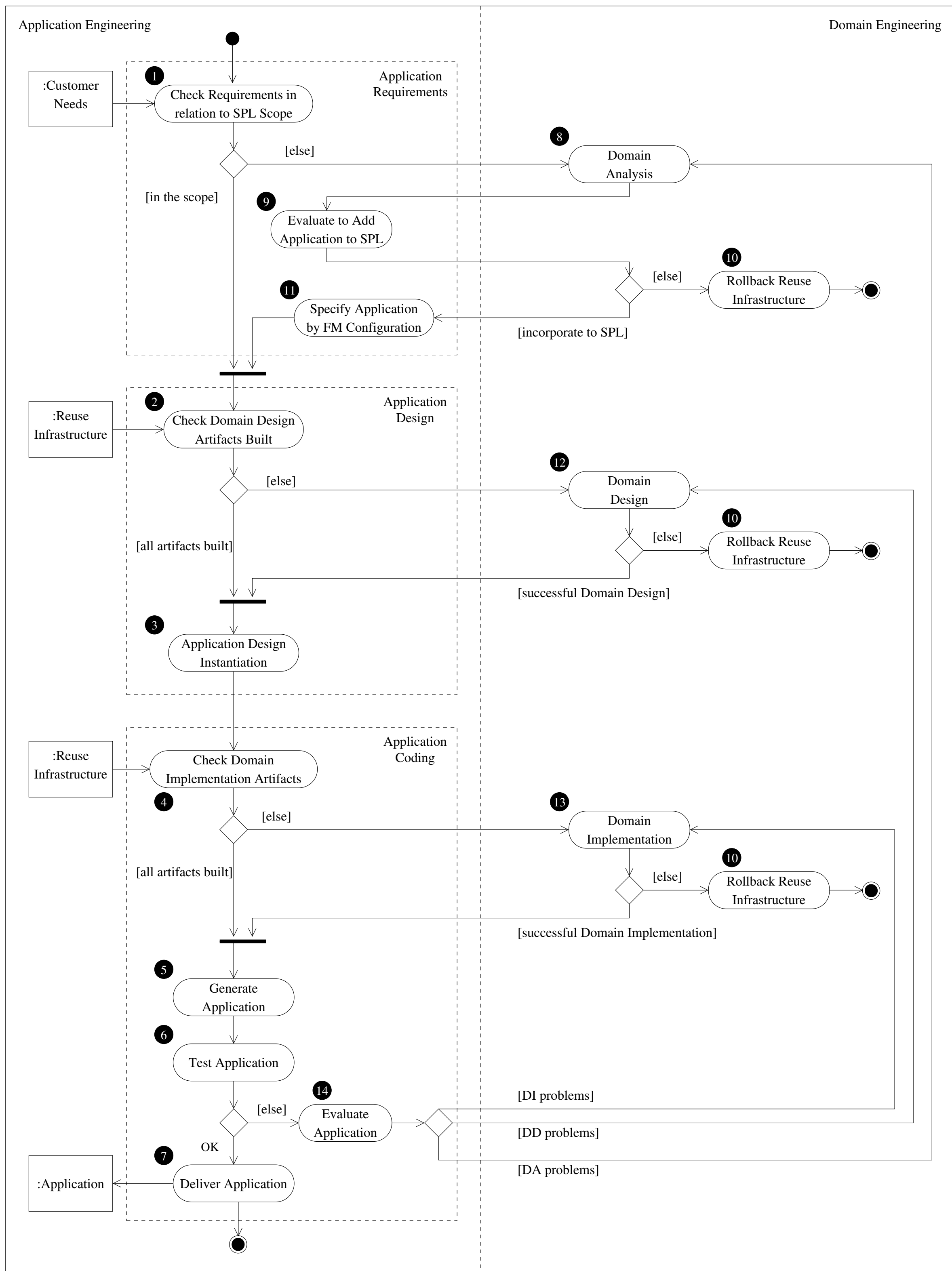


Figure A.3: Product Derivation Process

This activity can be carried out in two ways: through the use of the product line architecture, or through a model transformation. With the first alternative, several artifacts are necessary: the feature model configuration, the architectural structural views, and the be-

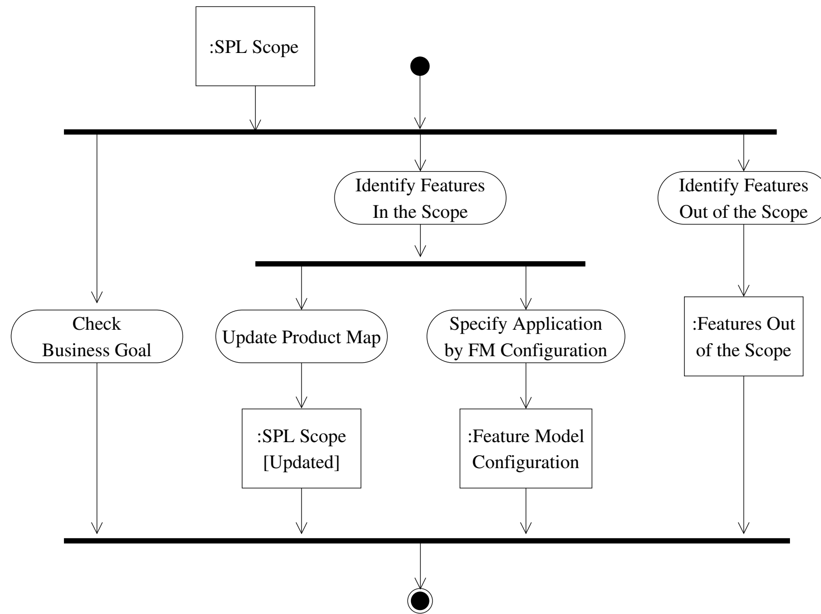


Figure A.4: Check Requirements in relation to SPL Scope

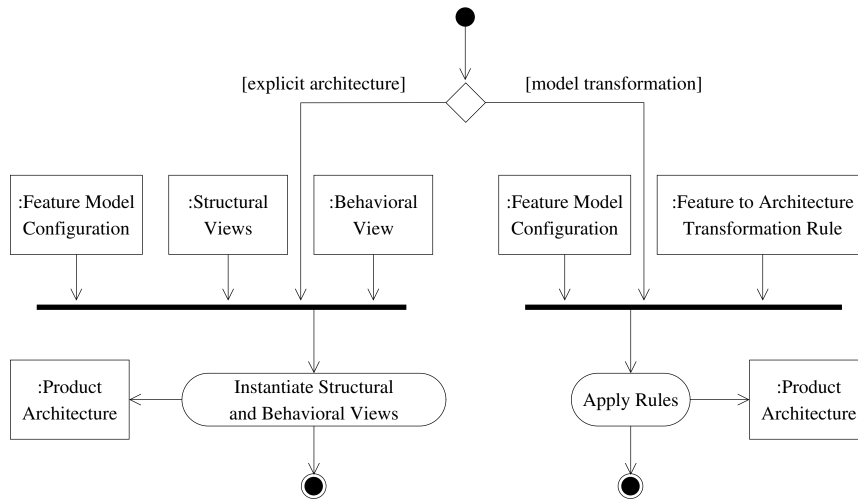


Figure A.5: Application Design Instantiation

havioral view. The product designer takes all the artifacts and proceeds to instantiate them, producing a particular product architecture. On the other hand, if the second alternative is chosen, a model transformation is applied over the feature model configuration and a unique artifact is built: the product architecture. The choice of one or the other alternative in the application design instantiation is related to which kind of Domain Design activity (⓬) was conducted. The two options were exposed in detail in Chapter 6.

The artifacts produced in activity ❸ are input to the application coding stage. The product implementer needs to check first if all domain implementation artifacts were developed. This is carried out during activity ❹: "Check Domain Implementation Artifacts". The artifacts to be checked are preimplemented components, generic code generators, and specific product generators. Similar to activity ❷, since the product is within the scope, it is highly

probable that all those artifacts exist in the Reuse Infrastructure artifact, because they were built during the domain implementation stage. The product implementer proceeds with ❺: the "Generate Application" activity shown in Figure A.6.

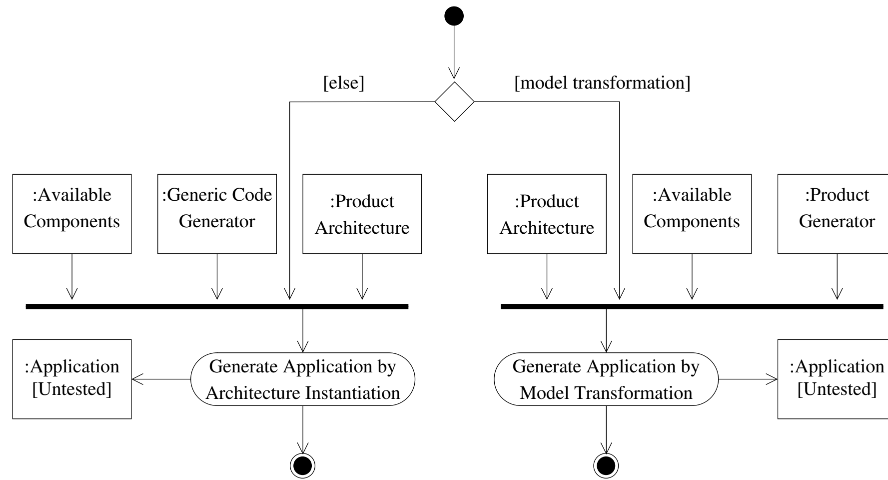


Figure A.6: Generate Application

Again, this activity may be carried out in two ways: through the use of a generic code generator, or through a specific product generator built by model transformation. The choice of one or the other alternative for generating the application is related to which kind of Domain Implementation activity (⓭) was conducted. With the first alternative several artifacts are needed: the generic code generator, the components for building the application, and the architecture of the product. The product implementer takes all these artifacts and with the generic code generator produces an untested application. If the other option is chosen, only the components, the architecture of the product and the specific product generator are needed to produce an untested application.

The resulting application is tested in activity ❻: "Test Application". This activity is not documented in this thesis. It is known that there are a variety of product tests [217]. This activity could consider at least two of those tests:

- System testing: Test the application as a whole. The focus is on testing component interactions.
- Acceptance testing: Test the application to decide whether or not it is ready to be accepted by the system developers and deployed in the customer environment.

Moreover, in the "Testing" practice area Northrop and Clements [157] discuss other suitable tests for SPL.

Finally, if the application overcomes the test activity, and therefore is ready to be used, the application is delivered to the customer (❼). On the other hand, if the application presents some problems during testing, it is necessary to evaluate those problems with activity ⓮.

These problems could be produced because something was missing or was wrong in the domain analysis, domain design or domain implementation stages, and therefore it is necessary to restart all product derivation processes, or go back to where the problem was produced, i.e, domain analysis activity (❽), domain design activity (⓬) or domain implementation activity (⓭).

A.2.4 Applications out of the SPL Scope

The derivation process starts with activity ❶ (Figure A.3) "Check Requirements in relation to SPL Scope". If the requested product can not be built by the SPL, it because the product is not aligned to the business goal defined for the SPL or it has some characteristics that are out of the scope of the SPL. This is shown in Figure A.4. In both cases is necessary that the domain analysts review the domain analysis activity, considering the features that are out of the scope, or modifying the business goal. The domain analysis activity (❽), which was described in Chapter 4, would permit incorporating new customer needs to the SPL, therefore expanding the SPL scope. Once domain analysis is finished and both domain model and SPL scope have been incorporated to the reuse infrastructure, it is necessary to evaluate if all requirements are included in these artifacts, because some features may not included due to consistency of the domain model, problems in the business goal, or the cost of some of the components in the product map. The evaluation occurs in activity ❾, and if it is successful, the product is added to the SPL scope, and a feature configuration model is built in activity ⓫. But, if something was wrong in the domain analysis activity and the application will be not considered in the scope of the SPL, it is necessary to return the reuse infrastructure to the previous state, i.e., before of the domain analysis for new customer needs. This is carried out in activity ❿ by the domain analysts.

A.2.5 Domain Design Artifacts Missing

Once the application requirements stage has finished successfully, the application design stage begins with activity ❷. If this activity, done by the product designers, does not find all necessary artifacts for an adequate application design instantiation activity (❸), then the domain design activity (⓬), described in Chapter 6, must be carried out by the domain designer.

A domain design activity must be conducted with the goal of incorporating all aspects of the application. It has many associated activities, dealing with the creation of the product line architecture, the production plan, the state machine and model transformation rules. When the domain design activity has finished successfully, then activity ❸ can be initiated. On the other hand, if some problems were detected during the domain design stage, and it was impossible to reflect all customer needs in the design, activity ❿ is done, returning the reuse infrastructure to the previous state, i.e, before of the domain analysis for new customer needs. The activity is carried out by the domain analysts and the domain designers in their respective stages.

A.2.6 Domain Implementation Artifacts Missing

The application coding stage begins with activity ❹ when the product implementer checks if all domain implementation artifacts were developed in such a way the generation of the products will be the result of using both available components and code generators. If some component or generator was not developed, then the domain implementation activity (⓭), must be done by the domain implementers.

Domain Implementation has two main activities. One of them is implementing and modifying preimplemented components, and the other one is creating the different code generators. When the domain implementation activity finishes correctly, then all artifacts are ready to be used to generate the application in activity ❺. If some problems were produced in the domain implementation, and it was impossible to create some components or generators, it is necessary to return the reuse infrastructure to the previous state, using activity ❿ for that purpose. The activity is carried out by the domain analysts, the domain designers and the domain implementers in their respective stages.

A.2.7 About Feedback

Feedback is an important aspect in SLP development. It permits to add requirements or features not considered initially in the SPL.

The SPL method presented in this thesis considers two kinds of feedback, mentioned in Section 2.5: horizontal and vertical. Figure A.3 explicitly shows these types of feedback.

Horizontal feedback is done through customer needs in the application requirements stage. New requirements not considered in the SPL can be incorporated following path ❶-❽-❾-⓫, which is considered the first vertical feedback.

New features need support from the reuse infrastructure. Here, two more vertical feedbacks are produced: the second one in the application design following the path ❷-⓬-❸, and the third one in the application coding following path ❹-⓭-❺.

Finally, three more vertical feedbacks are produced once the product or application is tested: the fourth one is ❻-⓮-❽-❾-⓫, the fifth one is ❻-⓮-⓬-❸, and the last one is ❻-⓮-⓭-❺.

Appendix B

Approaches for Building a SPL

There are several SPL approaches. Some of them are general, i.e., they can be used in any domain, any size of project or any quantity of individual products to produce, and others are specific, i.e., they can be used in specific domains because they were built specifically for taking advantage of certain aspects of the particular domains.

In the next sections, some general SPL approaches will be presented.

B.1 PuLSE

The PuLSE (Product Line Software Engineering) methodology [22] has been developed at the Fraunhofer Institute for Experimental Software Engineering, and it is a systematic approach for developing a software product line. The PuLSE methodology enables the conception and deployment of software product lines within a large variety of organizational contexts. In fact, Knauber et al. [123] report that this methodology is not only relevant for big companies but also for small and medium-sized companies.

`What is PuLSE`

Figure B.1 presents the phases and components of the methodology [22]. The *Deployment Phases* gathers the PuLSE life-cycle phases; the *Technical Components* is a set of components that gives support to PuLSE life-cycle phases; the *Support Components* includes a set components that give additional information to SPL development process.

According to Trigaux and Heymans [232], this methodology may be described as follows:

`PuLSE description`

- PuLSE provides a complete framework that covers the whole SPL development life cycle, including reuse infrastructure construction, usage, and evolution.
- PuLSE is modular and customizable: It consists of six technical components that can be selected and instantiated in order to satisfy the needs of specific companies.

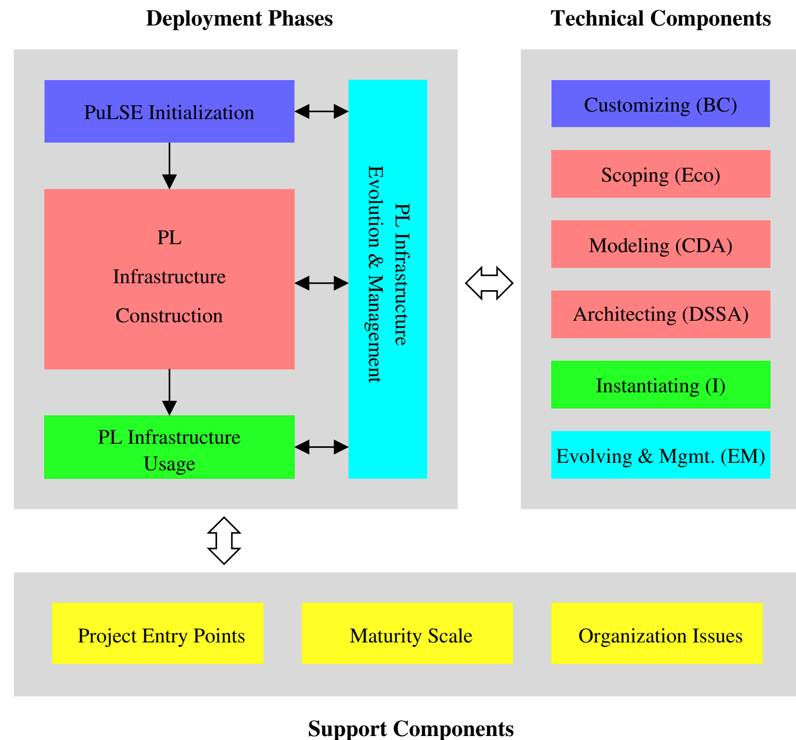


Figure B.1: PuLSE Overview

- PuLSE can be introduced incrementally by augmenting existing software development processes and products with product line specific aspects step by step.

One of the important features of PuLSE is that all elements of the methodology are appropriately documented through several articles [22, 24, 23, 68, 69, 193, 195, 198], with explanations about the components. An example of this is the use of diagrams, as is presented in Figure B.2, obtained from Bayer et al. [22], for showing and describing the process and their interactions.

PuLSE documentation

In the following sections each of the three elements of the methodology will be explained, according to Bayer et al. [22] and Trigaux and Heymans [232].

B.1.1 Deployment Phases

Deployment phases describe the activities performed to set up and use the product line and represent the logical stages of SPL. The phases are four:

1. Initialization: In this phase, an instance of the PuLSE methodology is produced and tailored to the organizational context in which it will be applied.
2. Infrastructure Construction: The purpose of this phase is to construct the product line infrastructure. It implies building a scope model and the definition of the product line

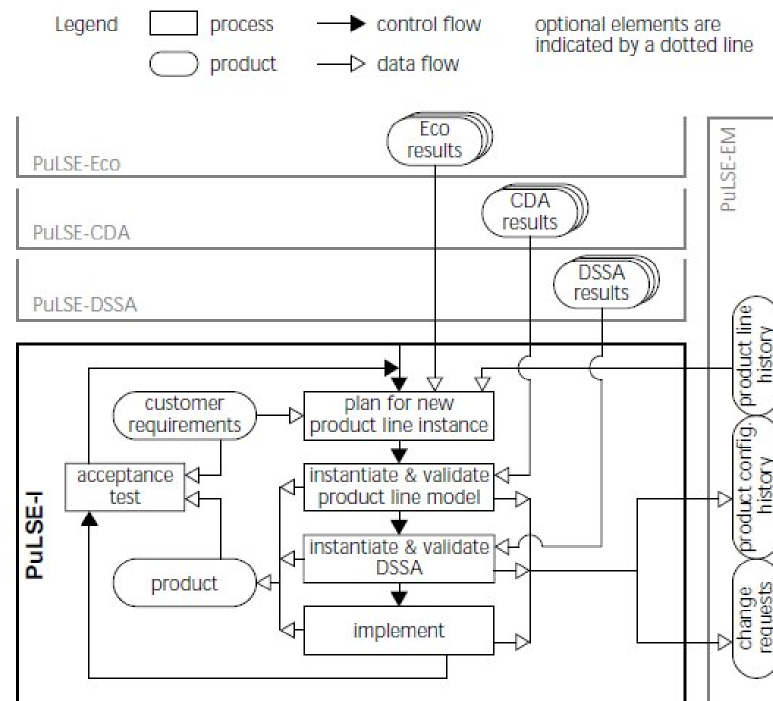


Figure B.2: PuLSE Usage Phase

architecture. This phase is decomposed in three parts, each of them performed by a technical component:

(a) PuLSE-Eco helps to determine an economic viable scope for the product line.

(b) PuLSE-CDA is used to elicit and articulate product line concepts and their interrelationships.

(c) PuLSE-DSSA is applied to define a software reference architecture for the product line.

3. Infrastructure Usage: This phase aims at specifying, instantiating, and validating one member of the product line. This encompasses the instantiation of the product line model and the reference architecture, the creation and/or reuse of products that constitute the instance, and validation of the resulting product.

4. Infrastructure Evolution and Management: The purpose of this phase is to monitor and control the evolution of the product line infrastructure which is built in the Construction phase.

B.1.2 Technical Components

Technical components provide the technical know-how needed to operationalize the product line development and are used throughout the Deployment Phases. The components are six:

1. Customizing (BC): Baselining and Customization is used to create an instance of the PuLSE method that is tailored to a specific organization context.

2. Scoping (Eco): Economic Scoping is concerned with an economic analysis of the product line. The goal is to identify candidate reusable assets that provide a high economic benefit to the organization at a low risk.

3. Modeling (CDA): Customizable Domain Analysis is a domain analysis approach that can be modified following the various characteristics of an organization. This technical component is essentially composed of requirements and knowledge engineering techniques.

4. Architecting (DSSA): Domain Specific Software Architecture is used to develop a reference software architecture for all products in the product line. This reference architecture covers current and future applications of the product line as described by a product line model.

5. Instantiating (I): Instantiation is concerned with product instance development, i.e., deriving a concrete product from the product line infrastructure. The key elements include instance planning, instantiation of domain model and reference architecture, as well as product construction.

6. Evolving and managing (EM): Evolution and Management describe how to integrate new products in the product line, and deal with configuration management issues as products accrue over time. This activity focuses on three main goals: managing the instantiation of the PuLSE process, defining feedback processes that allow continuous optimization of both the processes and the product line artifacts, and defining and realizing an appropriate configuration management strategy.

B.1.3 Support Components

Support components are packages of information, or guidelines, that enable a better adaptation, evolution, and deployment of the product line. These components are used by the other elements of the PuLSE methodology. The components are three:

1. Project entry points, which describe standard situations where PuLSE can be applied, and it helps to integrate PuLSE into a specific company context.

2. Maturity scale, which is designed to help an organization to adopt the methodology step by step. It drives towards the usage and integration of the different PuLSE phases and components so as to help an enterprise ultimately function fully according to a product line mode.

3. Organization issues, which provide guidelines to set up and maintain the right organization structure for developing and managing product lines.

B.2 Framework for Software Product Line Engineering

The framework for software product line engineering was developed by Pohl et al. [174], and descriptions of industrial case studies using the approach can be found in van der Linden et al. [236].

This framework is based on two development processes: domain engineering and application engineering. These processes are similar to traditional ones, as was explained in Section 2.5.

The framework is shown in Figure B.3, obtained from Pohl et al. [174].

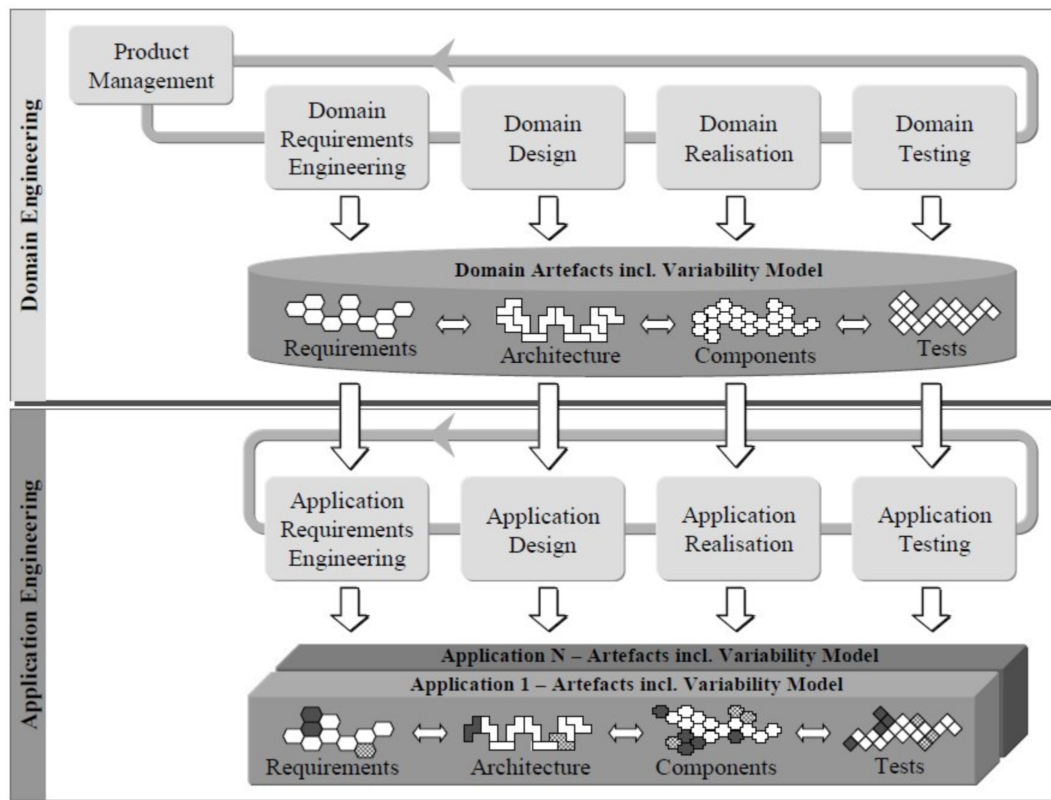


Figure B.3: The Two-life-cycle Model of Software Product Line Engineering

The upper part of the figure presents the domain engineering process. It is composed of five sub-processes: product management, domain requirements engineering, domain design, domain realization, and domain testing. On the other hand, in the lower part of figure, the application engineering process is presented. It is composed of the following four sub-processes: application requirements engineering, application design, application realization, and application testing.

Similarly to other approaches that use domain and application engineering phases, several artifacts are developed from each one of the sub-processes for each phase, and the artifacts produced by the domain engineering are used as a base for producing individual products during application engineering.

As in any SPL development method, variability is an important concept. Figure B.4, obtained from Pohl et al. [174], presents the graphical notation used for representing the existing variability in a family of products.

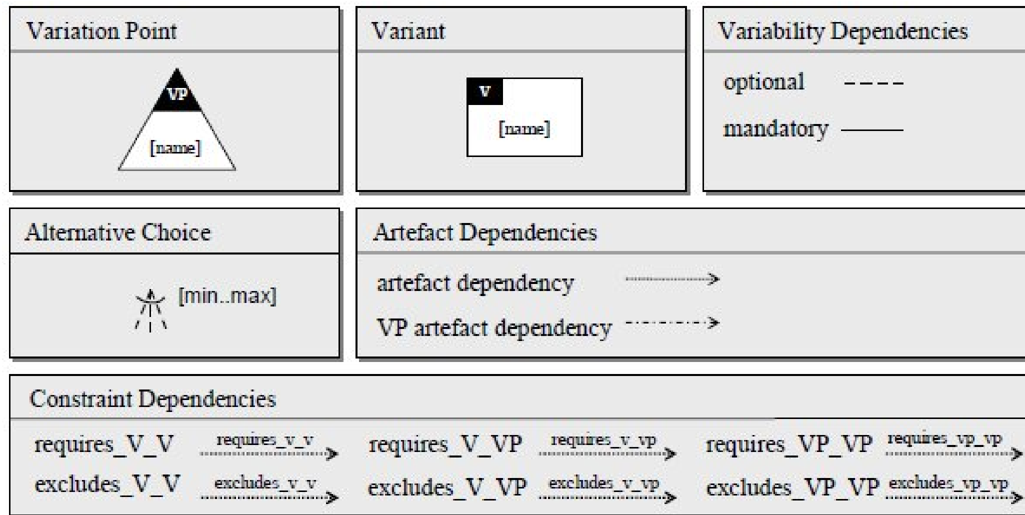


Figure B.4: Graphical Notation for Representing Variability

In summary, the *variation point* describes where differences exist in the final systems; the *variant* represents the different possibilities that exist to satisfy a variation point; the *variability dependencies* and *alternative choice* are used as a basis to represent the different choices (variants) that may fill a variation point, and the latter includes a cardinality which determines how many variants can be selected simultaneously; *artifact dependencies* describe dependencies among variation points or variants and other requirements, design, realization or test artifacts; *constraint dependencies* describe dependencies among certain variant selections: requires means that the selection of a specific variant may require the selection of another variant, and excludes means that the selection of a specific variant may prohibit the selection of another variant.

This notation is very similar to others sketched by authors as Kang et al. [115], Czarnecki and Eisenecker [60] and Czarnecki et al. [62]; in essence they consider the same elements considered by them. Probably, Pohl et al. [174]'s notation is less compact than others, and it could present some disadvantages when it is necessary to model a complex domain. On the other hand, this notation permits traceability between the variability model and the different artifacts produced on each one of the stages of the framework, which is an aspect that is not considered by the other authors.

There are some important elements used in the description of the framework. Figure B.5, obtained from van der Linden et al. [236], presents the four concepts that need to be considered in the product line engineering: Business, Architecture, Process and Organization.

The concepts are defined briefly as follows:

- Business: the costs and profits of the software, the application strategy and the production planning.

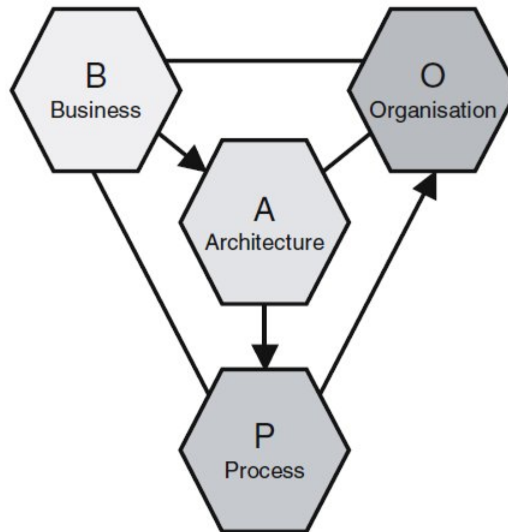


Figure B.5: BAPO Model

- Architecture: the technical means to build the software.
- Process: the roles, responsibilities and relationships within software development.
- Organization: the people and organizational structures that execute the software development.

As was shown in Figure B.5, the four concepts of BAPO are all interrelated, and if some changes are applied in one concept then those changes induce changes in other concepts. The arrows denote the order the elements should be to traversed in: Business is the most influential factor, architecture reflects these business concepts in software structure and rules, processes enable the development of the software, based on the architecture, and finally, organization hosts this process, assigning units and people who are responsible for business, architecture and process [236]. More details about BAPO can be found in van der Linden et al. [235].

Considering the above, this framework has some principles that are fundamental to its success. Two of them are business centric and architecture centric. The first one takes into account that product line engineering can only be successful if the reuse infrastructure is an adequate instrument to incorporate new products onto the market efficiently. The reuse infrastructure must be considered in the long term. For that, it is important to define the scope of the SPL, so the SPL can achieve the business goals and quality goals. In other words, the SPL must support all the defined goals, for each of the products that are in the scope of the SPL. The second one relies on the fact that there is no need to develop several components that address the same or similar functionality and differ only with respect to the environment they work in. In this principle it is necessary to define a product line architecture (reference architecture) in a way that this architecture allows taking advantage of commonalities among the individual products or systems.

In the following sections the stages of the framework are presented [174, 236].

B.2.1 Product Management

The product management stage deals with the economic aspects of the SPL. It aims at identifying the major commonalities and variabilities among the products. Its main concern is the management of the product portfolio of the organization, employing scoping techniques to define which products are in the scope and which products are not.

The input to this stage consists of the organizational goals defined by top management. The output is a product roadmap that determines the common and variable features of future products as well as a schedule with their planned release dates. Additionally, a list of existing products and development artifacts that can be reused and incorporated to the reuse infrastructure is provided.

B.2.2 Domain Requirements Engineering

The domain requirements engineering stage is in charge of eliciting and documenting the common and variable requirements of the SPL. An initial variability model is built for supporting the other development stages.

The input to this stage consists of the product roadmap. The output comprises reusable requirements and the variability model of the product line. The output does not include the requirements specification of a particular product.

B.2.3 Domain Design

The main concern of the domain design stage is to build the product line architecture, providing the base for the next stages. The architecture provides a common high-level structure for all product line applications.

The input to this stage consists of the domain requirements and the variability model from the domain requirements engineering stage. The output is the reference architecture.

B.2.4 Domain Realization

The domain realization stage covers both detailed design and implementation of the common and variable reusable software components.

The input to this stage consists of the product line architecture, including a list of reusable software artifacts to be developed in the domain realization stage. The output is mainly composed of the detailed design and implementation of reusable components.

B.2.5 Domain Testing

The domain testing stage is in charge of validation and verification of reusable components built in the domain realization stage, and the building of the reusable test artifacts to reduce effort during application testing.

The input to this stage are all artifacts developed in the previous stages. The outputs are the test results of the tests performed and the reusable test artifacts.

This stage is difficult mainly because there are no applications or systems to be tested. Only component tests are possible, integration tests are not.

B.2.6 Application Requirements Engineering

The application requirements engineering stage is centered in the production of the application requirements specification (individual product) by the identification of the specific requirements. It starts from the existing commonalities and variabilities model.

The input to this stage are the domain requirements (variability model) and the product roadmap. Sometimes, there may be additional specific requirements from a customer for the particular application that are not present in the variability model. The output is the requirements specification for the particular application.

B.2.7 Application Design

The application design stage produces an instance of the product line architecture, according to the requirements identified in the previous step. It selects and configures the required parts of the product line architecture and incorporates application specific adaptations.

The input to this stage is the product line architecture and the application requirements specification. The output is the architecture for the particular application.

B.2.8 Application Realization

The application realization stage develops the final implementation of the product, taking into account its requirements and architecture, reusing and configuring the existing components and building new components for product-specific functionality, if needed. In general, this stage can be seen as an assembly of components for building the product.

The input consists of the application architecture and the reusable components produced during the domain realization stage. The output consists mainly of an executable product.

B.2.9 Application Testing

This stage considers the verification and validation of the application that was generated in the previous stage.

The input to this stage mainly consists of the implemented application and the reusable test artifacts developed in the domain testing stage. The output is a test report with the results of all tests that have been performed.

B.3 Comments About the Approaches

There are several other software reuse approaches that have similarities with SPL, such as RiDE (The RiSE Process for Domain Engineering) by de Almeida [65], or The 3-Tiered Methodology by Krueger [128], but they are not as popular or widely used. Furthermore, it is highly probable that in the future the quantity of SPL methodologies will increase. Any new methodology must at least consider the three categories of SPL approaches described by Krueger [127]: software mass customization, minimally invasive transitions, and bounded combinatorics.

Considering each one of the approaches presented in Section B.1 and Section B.2, it is possible to establish that:

- The two approaches are sufficiently general as to consider their use in any domain.
- Their stages or practice areas cover almost any need in a development.
- Both input and output artifacts, are suitably and explicitly described.
- It is clear, from one stage to other one, which artifacts are inputs and outputs from which stage.

On the other hand, there are some aspects of the previous SPL approaches presented that make it difficult to apply them in any context or domain. Several of those aspects have been mentioned by other authors, and they are summarized in the next list:

- As they are general, it is necessary to tailor them for a specific domain. Unfortunately, it is not always clear what aspects, process, or artifacts of the methodologies will be used, and what can be omitted. In particular, Trigaux and Heymans [232] mention that SEI Framework must be tailored to the needs of specific projects.
- According to de Almeida [65], there are no step by step guidelines for customizing the SEI frameworks, or specific ways of performing the activities. The same issue

happens in PuLSE framework. Even though PuLSE itself has a technical component to customize it and produce an instance of it for the domain context in which it will be applied, it requires complete knowledge and experience with the PuLSE methodolgy for producing correct PuLSE instantiations.

- Authors such as Capilla and Dueñas [39] and Nedstam and Staples [151] believe that SEI Framework and PuLSE are too cumbersome to be effectively used by small and medium sized companies.

Considering the previous points, and taking into account that sometimes it is not necessary to consider all activities or steps in a SPL approach [39], it is possible to think in light-weight product-line approaches, or also in specific domain product-line approaches. The next is a non exhaustive list of them. The analysis is not a deep one; more information about each one of them can be found in the respective reference.

Other SPL approaches

1. Capilla and Dueñas [39] present a light-weight SPL model applied to the *Web* domain, mainly centered in evolution and maintenance. The idea of this approach is that existing web sites can be re-engineered without employing complex processes (e.g domain analysis or scope definition), reducing the initial investment barrier caused by the introduction of a SPL, by evolving the product line from already existing products.

2. Nunes et al. [159] developed an approach for the Multi-agent Systems (MAS) domain. MAS addresses the development of complex and distributed systems based on their decomposition into autonomous and pro-active agents, which together compose a Multi-agent System. An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [191].

   The approach covers all domain engineering process, using an UML-based notation, and it aggregates some activities that are specific to model software agents and their variabilities.

3. Thiel et al. [230] present the application of SPL concepts to *Car Periphery Supervision* systems, which is a limited and non complex domain, with the objective of reusing both software and hardware components. The authors follow standard software development procedures, only applying some stages of an SPL approach with successful results.

4. Zhang and Jarzabek [251] developed XML-based Variant Configuration Language (XVCL), which permits building product line assets as a set of x-frames that are capable of accommodating both commonality and variability in a domain. X-frames represent domain knowledge in the form of product line assets. Specific systems can be constructed by composing and adapting x-frames.

   XVCL is a general-purpose mark-up language developed for configuring variants in domain model, program code, test cases, and other kinds of documents. This approach has been applied to different domains.

Taking into account the previous discussion, the need for developing specific SPL approaches that consider the special characteristics of some domains and take advantages of their particularities is clear.

Finally, it is possible to find other approaches for software reuse, in particular centered in the domain analysis stage. These specific methods were analyzed in page 71 of Section 4.2.4.

Appendix C

ATL Transformation Rules

The following sections show some ATL rules built for the domain design stage for the architecture building transformational process in Section 7.2.2. Section C.1 shows ATL rules that allow building the architecture structural view, while Section C.2 presents rules for building the architecture behavioral view.

C.1 Rules for the Structural View

Listing C.1: CSG Rule for the Structural View

```
rule rCSG{
  from
    f : FCMMM!Feature (f.name = 'CSG')
  to
    c : PAMM!Component (name <- f.name, provided <- ip, required <- ir),
    ip : PAMM!Interface (name <- 'IGeometry'),
    ir : PAMM!Interface (name <- 'IGenerate')
}
```

Listing C.2: Generate initial mesh Rule for the Structural View

```
rule rGenerate_initial_mesh{
  from
    f : FCMMM!Feature (f.name = 'Generate initial mesh')
  to
    c : PAMM!Component (
          name <- f.name,
          components <- f.members,
          provided <- ip,
          required <- ir,
          connectors <- Set{xp, xr}),
    ip : PAMM!Interface (name <- 'IGenerate'),
    ir : PAMM!Interface (name <- 'IMesh'),
```

```
    xp : PAMM!Connector(
        name <- c.components->first().name + '_Generate',
        kind <- #Delegate,
        source <- ip,
        target <- c.components->first().provided->first()),
    xr : PAMM!Connector(
        name <- c.components->first().name + '_Mesh',
        kind <- #Delegate,
        source <- c.components->first().required->first(),
        target <- ir)
}
```

Listing C.3: Refine Rule for the Structural View

```
rule rRefine{
  from
    f : FCMMM!Feature (f.name = 'Refine')
  to
    c : PAMM!Component (
        name <- f.name,
        provided <- ip,
        required <- ir),
    ip : PAMM!Interface (name <- 'IRefine'),
    ir : PAMM!Interface (name <- 'IMesh')
}
```

C.2 Rules for the Behavioral View

Listing C.4: Meshing Tool Rule for the Behavioral View

```
rule rMeshingTool{
  from
    f : FCMMM!Feature (f.name = 'Meshing Tool')
  to
    s1 : PABvMM!Initial (number <- '1'),
    s2 : PABvMM!Normal (number <- '2'),
    t1 : PABvMM!Transition(
        sourceStatenumber <- s1.number,
        event <- 'select dimension',
        targetStatenumber <- s2.number)
}
```

Listing C.5: CSG Rule for the Behavioral View

```
rule rCSG{
  from
    f : FCMMM!Feature (f.name = 'CSG')
  to
    s1 : PABvMM!Normal (number <-'2'),
    s2 : PABvMM!Normal (number <-'3'),
    t1 : PABvMM!Transition(
        sourceStatenumber <-s1.number,
        event <- 'load input',
        targetStatenumber <-s2.number)
}
```

Listing C.6: Generate initial mesh Rule for the Behavioral View

```
rule rGenerate_initial_mesh{
  from
    f : FCMMM!Feature (f.name = 'Generate initial mesh')
  to
    s1 : PABvMM!Normal (number <-'3'),
    s2 : PABvMM!Normal (number <-'4'),
    t1 : PABvMM!Transition(
        sourceStatenumber <-s1.number,
        event <- 'produce initial mesh',
        targetStatenumber <-s2.number)
}
```

Listing C.7: Refine Rule for the Behavioral View

```
rule rRefine{
  from
    f : FCMMM!Feature (f.name = 'Refine')
  to
    s1 : PABvMM!Normal (number <-'4'),
    s2 : PABvMM!Normal (number <-'4'),
    t1 : PABvMM!Transition(
        sourceStatenumber <-s1.number,
        event <- 'refine mesh',
        targetStatenumber <-s2.number)
}
```

Listing C.8: Output Rule for the Behavioral View

```
rule rOutput{
  from
    f : FCMMM!Feature (f.name = 'Output')
  to
    s1 : PABvMM!Normal (number <-'4'),
    s2 : PABvMM!Final (number <-'5'),
```

```
    t1 : PABvMM!Transition(
        sourceStatenumber <−s1.number,
        event <− 'apply output',
        targetStatenumber <−s2.number),
    s3 : PABvMM!Final (number <−'5'),
    s4 : PABvMM!Final (number <−'5'),
    t2 : PABvMM!Transition(
        sourceStatenumber <−s3.number,
        event <− 'apply output',
        targetStatenumber <−s4.number)
}
```

Index

— A —

application engineering, 5
  definition, 17
  substages, 6
architecture description language, 96
  definition, 96
ATL, 106, 118, 176

— C —

commonalities, 5, 17

— D —

derefinement
  Catalan sequence, 45
  edge collapse, 45
  simplification operations, 44
  vertex remove, 44
domain
  stakeholders identification, 59
domain analysis, 127
  definition, 53
  goals, 53
  purposes, 27
  steps, 27
domain application coding
  purposes, 30
domain application design
  purposes, 30
domain application requirements
  purposes, 29
domain design, 128
  purposes, 28
domain engineering, 5
  definition, 17
  substages, 6
domain implementation
  purpose, 28
domain model
  definition, 55
domain-specific language, *see* DSL
DSL
  advantages, 34
  definition, 31
  development process phases, 33
  disadvantages, 34
  elements, 32
  examples, 31

— G —

grid generation, *see* mesh generation

— I —

improvement
  definition, 46
  diagonal swap, 46
  element open, 46

— M —

mesh, 2
  definition, 37
  structured, 38
    advantages, 39
    disadvantages, 39
  unstructured, 38
    advantages, 39
    disadvantages, 39

mesh generation
  definition, 35
  steps, 4, 37
meshing tool, 1
  application domains, 3
  definition, 37
  generator, 130
  price, 2
  traditional software development, 51
  with reuse software development, 51

— O —

octree
  definition, 40
optimization
  definition, 47

— P —

product derivation
  activities, 156
  definition, 155
  phases, 156
  process, 155
  purpose, 155
product line architecture
  assessment benefits, 100
  definition, 28, 95
program family, 15
  definition, 15
PuLSE
  description, 164
  documentation, 165
  purpose, 164

— Q —

quadtree
  balanced, 40
  definition, 40
quality attributes, 93, 103
  categories, 93

— R —

reference architecture, *see* product line architecture
refinement
  definition, 43
  triangle partitions, 43

— S —

scope
  definition, 56
  differences with requirements, 60
  evolution, 59
  process, 56
scoping
  classification, 57
software architecture
  definition, 92
  purpose, 93
software development trinity, 96
software family, *see* software product line
software product family, *see* software product line
software product line, 1, 5, 15
  adoption, 18
  advantages, 20
  benefits, 6
  definition, 16
  disadvanteges, 22
  drivers, 9
  evolution, 22
  example, 16
  hypotheses, 10
  other approaches, 174
  phases, 5
software reuse
  benefits, 14
  definition, 13
structured mesh, 38

— T —

time to market
  definition, 20

— U —

unstructured mesh, 38

— V —

variabilities, 5, 17