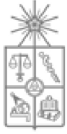

UNIVERSIDAD DE CHILE

FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

OPENING UP TRACE-BASED MECHANISMS

Application to Context-Aware Systems

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS
MENCIÓN COMPUTACIÓN

PAUL SAINT LEGER MORALES

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
NELSON BALOIAN TATARYAN
JOHAN FABRY
ROMAIN ROBBES
ERIC BODDEN (Technische Universitat Darmstadt)

SANTIAGO DE CHILE

This thesis is dedicated to all PhD students fallen in battle.

Esta tesis está dedicada a todos los estudiantes de doctorados caídos en batalla.

Acknowledgments

In the first place, I want to thank the Computer Science Department of the University of Chile for giving me the opportunity to conduct research at this prestigious institution. I am especially thankful to CONICYT, NIC Chile and CIAE for providing me with scholarships during my PhD. I am very grateful to the members of the PLEIAD laboratory for encouraging my research endeavors. In particular, I want to thank my advisor, Éric Tanter, whose expertise in the art of writing, research, and patience, were fundamental to my successful completion of my PhD. Although his collaboration came about towards the end of my process, I want to thank Renato Cerro for helping me improve my poor English.

I am very grateful to Johan Fabry, Romain Robbes, Nelson Baloian, and Eric Bodden, for accepting to be part of my jury, reviewing my dissertation, and providing constructive comments. I am particularly grateful to Eric Bodden for coming to my defense from so far away.

Those six years would denitely not have been the same without the company of Rodolfo Toledo (the procrastinator), Oscar Callaú (the swimmer), Guillaume Pothier (the Linux man), and many others. Apart from teaching and inspiring me, they gave me many a good laugh.

Last but not least, my warmest feelings go to my family, who gave me the hope and the strength to pursue this PhD. Finally, I do not have words to express my gratitude to Dayana (my partner), Kuky and Toti, for their unconditional company and support during these long years. I love you!

"Perfection is achieved, not when there is nothing more to add,
but when there is nothing left to take away."
The Little Prince, 1943.
Antoine de Saint-Exupéry

Resumen

En 1972, David Lorge Parnas argumentó que la programación modular es extremadamente valiosa para el desarrollo de grandes software. Esto es así porque un modulo puede ser escrito con un pequeño conocimiento de los otros módulos y ser remplazado sin la necesidad de reescribir los restantes módulos. Sin embargo, los paradigmas basados en proceduras como orientación a objetos no soportan completamente la programación modular debido a los intereses transversales (conocidos como *crosscutting concerns*) de un software. Afortunadamente, la programación orientada a aspectos provee un conjunto de abstracciones y mecanismos que permiten a los desarrolladores modular estos intereses; por lo tanto, volver a la programación modular. *Aspects* son abstracciones para modularizar intereses transversales. Un aspect intercepta un punto de ejecución, llamado *join point*, con el objetivo de ejecutar una acción, llamado *advice*, la cual implementa un particular interés transversal. No obstante, algunos intereses transversales no pueden ser modularizado a través de la intercepción de un solo punto, *e.g.* detección de errores. Trace-based mechanisms suportan las definiciones de *stateful aspects*, lo cuales pueden interceptar trazas de join points. Un stateful aspect se define a través de un patrón acerca de una traza de join points y un advice que es ejecutado cuando el patrón calza con la traza.

Varios trace-based mechanisms han sido propuestos, los cuales no permiten a los desarrolladores expresivamente definir patrones y la semántica de sus stateful aspects. Por lo tanto, los desarrolladores terminan sobrecargando inapropiadamente las definiciones de los stateful aspects o creando especializados trace-based mechanisms para modularizar un particular interés transversal. En esta tesis, nosotros proponemos un modelo de un expresivo y abierto trace-based mechanism, llamado OTM, el cual si permite a los desarrolladores flexiblemente definir patrones y definir la semántica de los stateful aspects (en particular, la semántica *matching* y *advising*). Además, la tesis presenta una practica implementación de OTM para JavaScript y una descripción funcional en Typed Racket.

Nosotros usamos la implementación para JavaScript de OTM para crear un número de software que adaptan sus conductas cuando ellos detectan ciertos contextos, llamados *context-aware software*. Un contexto es detectado por el análisis de la historia de eventos de un context-aware software. En este tipo de software, el patrón de un stateful aspect representa el contexto que debe ser detectado y el advice representa la adaptación del software. El desarrollo de estos sistemas usando OTM mostraron que nuestra practica implementación mejora el soporte a la implementación modular de aplicaciones.

Abstract

In 1972, David Lorge Parnas argued that modular programming is extremely valuable for the development of large pieces of code. This is so because a module can be written with little knowledge of the code of other modules and replaced without the need to rewrite the remaining other modules. However, paradigms based on general procedures like object-oriented programming do not fully support modular programming due to crosscutting concerns of a system. Fortunately, the aspect-oriented paradigm provides a set of abstractions and mechanisms that allow developers to modularize these concerns; therefore, improving modular programming. Aspects are widely-known abstractions to modularize crosscutting concerns. An aspect intercepts a single execution point, named join point, to execute a piece of code, named advice, that implements a crosscutting concern. However, some crosscutting concerns cannot be modularized through the interception of a single join point, *e.g.* error detections. Trace-based mechanisms support the definitions of stateful aspects that intercept join point traces. A stateful aspect is defined by a join point trace pattern and an advice that is executed when this pattern is matched.

Various trace-based mechanisms have been proposed. These mechanisms do not share the exact semantics, which suggests there is no silver-bullet trace-based mechanism for all purposes. In addition, existing mechanisms do not allow developers to expressively define patterns and semantics of their stateful aspects. Therefore, developers end up "coding around" these mechanisms or creating specialized ones to modularize a particular crosscutting concern. In this thesis work, we propose a model of an expressive and open trace-based mechanism, named OTM. This model allows developers to flexibly define patterns and adapt them at runtime. In addition, as OTM follows the open implementation guidelines, this model allows developers to customize the semantics of how stateful aspects match and advise join point traces. Finally, this dissertation presents a concrete and practical implementation of OTM for JavaScript and a typed functional description in Typed Racket.

We use the JavaScript implementation of our model to develop a number of systems that adapt their behavior when they detect certain contexts, know as context-aware systems. A context is detected by analyzing the history of events of a context-aware system. In this kind of systems, patterns represent the contexts that must be detected and advices represent the system adaptations. The development of these systems showed that our practical version improves modularity support to build (Web) applications, and that will eventually make it possible to empirically validate the benefits brought by an expressive and open trace-based mechanism.

Contents

Acknowledgments iii

Resumen iv

Abstract v

Nomenclature xiv

1 Introduction 1

2 Thesis and Hypotheses 6

2.1 Thesis 6

2.2 Hypotheses 7

3 Context-Aware Systems, Trace-Based Mechanisms, and Open Implementations 9

3.1 Aspect-Oriented Programming 9

3.1.1 Origins 10

3.1.2 Concepts of an Aspect Language 10

3.1.2.1 Join Point Models 10

3.1.2.2 The Pointcut and Advice Model 11

3.2 Context-Aware Systems 12

3.2.1 Origins and Concepts 12

3.2.2 Crosscutting Concerns in Context-Aware Systems 13

3.3 Trace-based Mechanisms 14

3.3.1 Origins 14

3.3.2 Concepts of a Trace-based Mechanism 15

3.3.2.1 Pattern Language 15

3.3.2.2 The Matching Process 16

3.3.2.3 The Advising Process . . . 17
3.3.3 A Stateful Aspect Example: intruder detector . . . 17
3.3.4 Existing Trace-based Mechanisms . . . 17
3.3.5 The Limitations of Existing Trace-based Mechanisms . . . 20
3.3.5.1 Pattern Language . . . 21
3.3.5.2 Matching and Advising Processes . . . 23
3.4 Open Implementations . . . 23
3.4.1 Why Should a System Be Open? . . . 23
3.4.2 Origins and Concepts . . . 24
3.4.3 Open Implementations and Trace-Based Mechanisms . . . 25
3.5 Problem Statement . . . 25

Part I: An Open Trace-based Mechanism **27**

4 AspectScript: Expressive Aspects **28**
4.1 Introduction . . . 28
4.2 AspectScript in Action . . . 30
4.2.1 A Simple Example . . . 30
4.2.2 Pointcuts: Matching Sequences . . . 32
4.2.3 Giving Life to JavaScript Values . . . 34
4.2.4 Access Control with Scoping Strategies . . . 35
4.2.5 Identifying New Kinds of Events . . . 37
4.2.6 Summary . . . 38
4.3 A Tour of AspectScript . . . 39
4.3.1 Hybrid Join Point Model . . . 39
4.3.2 Higher-Order Aspects . . . 40
4.3.3 Deployment and Scoping Strategies . . . 42
4.3.4 Control of Aspect Reentrancy . . . 45
4.4 Implementation . . . 46
4.4.1 Code Transformation . . . 47
4.4.2 Runtime Weaving . . . 49
4.4.3 Runtime Performance . . . 50
4.5 Aspect Languages for the Web &
Higher-Order Aspect Languages . . . 52

4.6 Summary 54

5 ETM: Expressive Stateful Aspects 55

5.1 Introduction 56

5.2 Trace-based Mechanisms 57

5.3 Limitations of Existing Trace-based Mechanisms 59

5.3.1 Reusability 59

5.3.2 Binding Expressiveness 60

5.3.3 Pattern Expressiveness 60

5.4 ETM 60

5.4.1 Pattern Language 61

5.4.1.1 Without Gathering Bindings 61

5.4.1.2 Gathering Bindings 62

5.4.1.3 ETM Pattern Language is Turing Complete 63

5.4.2 Advice Model 65

5.4.3 Stateful Aspect Model 66

5.5 ETM in Action 66

5.5.1 Reusability 66

5.5.2 Binding Expressiveness 67

5.5.3 Pattern Expressiveness 68

5.6 Summary 70

6 OTM: Opening ETM 71

6.1 Introduction 71

6.2 Limitations of Fixed Stateful Aspects 73

6.2.1 Fixed Matching Process 75

6.2.1.1 Policy Discount Restriction: One Computer by Category gets Discounts 75

6.2.1.2 Policy Discount Restriction: Some Promotions are Temporary . 76

6.2.2 Fixed Advising Process 77

6.2.2.1 Policy Discount Restriction: Only the Computer with more Pieces gets Discounts 77

6.2.3 Summary 78

6.3 OTM 78

6.3.1 Matching Process . . . 79
6.3.1.1 MatcherCells . . . 80
6.3.1.2 MatcherCells Implementation . . . 81
6.3.2 Advising Process . . . 84
6.3.3 Stateful Aspect . . . 85
6.4 OTM for New Semantic Variants . . . 85
6.4.1 Customizing the Matching Process . . . 86
6.4.1.1 Policy Discount Restriction: One Computer by Category gets Discounts . . . 86
6.4.1.2 Policy Discount Restriction: Some Promotions are Temporary . 87
6.4.2 Customizing the Advising Process . . . 87
6.4.2.1 Policy Discount Restriction: Only the Computer with more Pieces gets Discounts . . . 87
6.4.3 Conclusion . . . 88
6.5 OTM for Emulating Existing Trace-based Mechanisms . . . 88
6.5.1 ETM . . . 88
6.5.2 HALO . . . 89
6.5.3 Tracematches . . . 90
6.5.3.1 Tracematch Symbols . . . 90
6.5.3.2 Matching Process . . . 91
6.5.3.3 Advising Process . . . 93
6.5.4 Conclusion . . . 93
6.6 Evaluations . . . 93
6.6.1 Adding an Expert Mode in Tetris . . . 94
6.6.2 Benchmarks . . . 95
6.7 Design Discussion . . . 97
6.7.1 A New Algorithm to Match Traces . . . 97
6.7.2 Openness Points . . . 98
6.8 Summary . . . 98

Part II: Case Studies **100**

7 Modularly and Flexibly Control Causality on the Web 101

7.1 Introduction . . . 102
7.2 Ajax & Web 2.0 Applications . . . 103
7.2.1 Controlling Message Causality . . . 104
7.2.2 Reacting to Distributed Causal Relations . . . 106
7.2.3 State of the Practice . . . 106
7.2.4 WeCa Overview . . . 108
7.3 Distributed Computing . . . 108
7.3.1 Distributed Computing . . . 109
7.3.1.1 Controlling Message Causality . . . 110
7.3.1.2 Reacting to Distributed Causal Relations . . . 111
7.4 WeCa . . . 113
7.4.1 Controlling Message Causality . . . 113
7.4.2 Reacting to Distributed Casual Relations . . . 115
7.4.3 Summary . . . 118
7.5 Revisiting Web 2.0 Applications . . . 118
7.5.1 Controlling Message Causality . . . 118
7.5.2 Reacting to Distributed Causal Relations . . . 121
7.6 Evaluation . . . 123
7.6.1 Modular Use . . . 123
7.6.2 Controlling Server Responses . . . 124
7.6.3 WeCa Overhead . . . 125
7.6.3.1 Overhead of Controlling Message Causality . . . 125
7.6.3.2 Overhead of Reacting to Distributed Causal Relations . . . 126
7.7 Causality on the Web . . . 127
7.8 Summary . . . 128

8 Supporting Adaptive Software for Primary Students 130

8.1 Introduction . . . 131
8.2 Mental Calculation Strategies & System Requirements To React to Them . . . 132
8.2.1 Mental Calculation Strategies . . . 132
8.2.2 Computer System Requirements to React to Mental Calculation Strategies . . . 133

8.3 ECOCAM . . . 134
8.3.1 Number Iconizations . . . 134
8.3.2 Analyzing Student Steps . . . 135
8.3.2.1 Detecting Patterns of Student Steps . . . 135
8.3.2.2 Reacting to Patterns of Student Steps . . . 136
8.4 Promoting Mental Calculation Strategies using ECOCAM . . . 137
8.4.1 Detecting and Reacting to the Transfer Strategy . . . 137
8.4.1.1 Detecting the Transfer Strategy . . . 138
8.4.1.2 Reacting to the Transfer Strategy . . . 139
8.4.2 Evaluation . . . 140
8.5 Summary . . . 140

Part III: Conclusions **142**

9 Contributions **143**
9.1 AspectScript . . . 143
9.2 ETM . . . 144
9.3 OTM . . . 145
9.4 MatcherCells . . . 146
9.5 Case Studies . . . 147
9.5.1 WeCa . . . 147
9.5.2 ECOCAM . . . 148
9.6 Thesis Work in a Few Words . . . 149

10 Perspectives **150**
10.1 Formal Semantics for OTM . . . 150
10.2 OTM Performance . . . 151
10.3 OTM in Conscientious Software . . . 151

A Summary of the OTM API **152**

B A Typed Functional Description of OTM **154**

References **166**

List of Figures

2.1 *a)* A programmer provides the pattern and adaptation to create a monitor. *b)* Apart from an expressive pattern and adaptation, the programmer can provide the core semantics of the detecting and reacting processes. 7

3.1 Modifications of a system to support context awareness. 14
3.2 A programmer expresses a stateful aspect and later this stateful aspect matches a trace of join points. 16
3.3 A stateful aspect matches twice the trace $a \rightarrow b$. 16
3.4 Limitations in terms of expressiveness of existing trace-based mechanisms 22
3.5 A system that supports open implementations (From [Rao, 1991]). 24
3.6 A refined version of Figure 2.1. a) Traditional view of trace-based mechanism. b) The trace-based mechanism view proposed by this thesis. 26

4.1 Dynamic join points of AspectScript. 39
4.2 Some pointcuts available in AspectScript. 42
4.3 Aspect deployment in AspectScript. Scoping strategies can be optionally specified in the last two alternatives. 43
4.4 Subset of the JavaScript syntax. 47
4.5 Rewriting function. 48
4.6 Weaving process. 49
4.7 Performance overhead of AspectScript for CPU-intensive tests in the JQuery test suite. 51

5.1 A Web text editor. 57
5.2 Matching three editions. 62

6.1 Possible matches of a pattern. 72
6.2 The base and meta interface of a stateful aspect. 79
6.3 Different kinds of reactions of a cell to a reagent. 80
6.4 *a)* The cell creates a cell that expects to match the next join point and keeps the bindings. *b)* When a cell matches the last join point specified, the cell creates a match cell. . . . 81
6.5 Using MatcherCells to match multiple times. 81
6.6 Different semantics for a matching process. 82
6.7 Pieces of code of three tracematches: $t1$, $t2$, and $t3$. 90
6.8 A Tetris game with a new mode: "Expert Mode". 94
6.9 A sequence of two moves: right follows by left. 94
6.10 Overhead of OTM with two different matching semantics over AspectScript. . . . 96
6.11 Overhead of tracematches over AspectJ. 96
6.12 Overhead of AspectScript over JavaScript. 96
6.13 Overhead of AspectJ over Java. 96

7.1 Two possible scenarios when a Web application sends two Ajax requests. 104
7.2 Retweeting a tweet. 107
7.3 The solution proposed by WeCa. 108
7.4 WeCa overview. 109
7.5 Two possible distributed computations for the same input. 110
7.6 Join points of a distributed computation tagged with vector clocks. 111
7.7 Two aspects to apply a message ordering strategy. 114
7.8 The tagVC aspect that tags every join points with a vector clock. 116
7.9 The notify aspect that notifies every join point to other applications. 117
7.10 Modification of Ajax Video Player to support AjaxManager. 124
7.11 Benchmarks for the reaction to distributed causal relations. 127

8.1 Icons provided by the ECOCAM framework. 135
8.2 The different operations supported by number iconizations. 135
8.3 Manipulating number iconizations to alter an addition. 136
8.4 Two scenarios of the Web application developed using ECOCAM. The left scenario shows a student that is transferring a unit from 27 to 19. The right scenario shows the resultant addition after the transfer $(20 + 26)$. 138

9.1 The big picture of our work. Contributions are in **bold**. 149

B.1 The module of the ETM semantics: the default semantics of OTM. 154
B.2 The pattern module and two pattern designators. 155
B.3 The Cell module. 156
B.4 Some rules designators. 157
B.5 The OTM module. 158

Chapter 1

Introduction

The foundations of software development are themselves, programming languages. As a matter of fact, it is the strong trend towards the development and use of programming languages with novel abstractions and mechanisms to express programs. Programming languages aim to tackle the complexity faced by software developers. *Separation of concerns* [Dijkstra, 1968, Parnas, 1972], establishes that a program should be decomposed in a set of *modules*, and that each module should address a given concern of the software. Modules are crucial for raising the understandability, maintainability, reusability, and evolvability of software.

Research in programming languages has allowed language designers to include new abstractions, mechanisms, and even new paradigms to modularize concerns of the software. For instance, the object-oriented paradigm is a good example of this research by the encapsulation of behavior (code) and state in modular units, called objects. However, the convergence and wide use of information technologies in software development has pushed object-oriented programming to its limits and more [Hayes, 2003], compromising the modularity. For example, the advent of context-aware systems [Satyanarayanan, 2001], systems that *adapt* their behavior at runtime according to their execution *context*, has posed a menace to object-oriented programming. This is so due to the need to detect contexts and to adapt the software. Detection and adaptation are hard to modularize [Tanter et al., 2006].

Various approaches to modularize adaptations of software have been explored, and others are still under exploration. In 1993, Gassanenko [Gassanenko, 1993] comments about the modularization issues of the adaptation in context-aware systems. More recently, Context-Oriented Programming (COP) [Costanza and Hirschfeld, 2005, Hirschfeld et al., 2008] addresses these issues through mechanisms like *layers*: pieces of code that change when a context is explicitly

activated. However, COP has not addressed the issues related to the detection of contexts in a modular manner. To modularize the detection of contexts, various trace-based mechanisms have been proposed [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Martin et al., 2005, Ostermann et al., 2005, Eugster and Jayaram, 2009]. A trace-based mechanism supports the definitions of monitors that detect and react to the software execution, which represents a certain given context.

Problem Statement and Thesis in a Nutshell

Problem statement. Dey *et al.* [Dey and Abowd, 2000] define a context as any information that can be used to characterize the situation of an entity (*e.g.* person, place, security, etc). This definition implies that it is necessary to observe any event inside/outside of the software to detect a certain context, meaning that the detection of a context is potentially a *crosscutting concern*. Hence, the detection of a context cannot be modularized using paradigms based on generalized procedures (*e.g.* object-oriented). Tanter *et al.* [Tanter et al., 2006] use the aspect-oriented paradigm [Kiczales et al., 1997b] to modularize the detection of contexts through aspects that detect and react to certain events of the execution of a program. However, context-aware aspects can only detect contexts based on the current event, and several contexts, like a person cooking in a kitchen, are represented by complex sequences of events. Trace-based mechanisms like tracematches [Allan et al., 2005] allow developers to detect patterns of events in a modular manner. Various trace-based mechanisms have been proposed, specifically tailored to address particular domains (*e.g.* security, error detections, context-aware systems, etc). Each of them expresses these patterns through different and limited domain-specific languages, like regular expressions; in addition, each mechanism has fixed semantics to detect patterns of events. In spite of different variations of trace-based mechanisms [Meredith et al., 2011], there is no single trace-based mechanism that is versatile enough to allow developers to *a)* flexibly express patterns of events that should be detected and *b)* implement and take advantage of different semantics of existing trace-based mechanisms as well as new semantic variants.

Thesis. This thesis proposes a model of an expressive and customizable trace-based mechanism, which encompasses existing trace-based mechanisms and makes it possible to define new semantic variants. To achieve this goal, this model allows developers *a)* to expressively define patterns of events through the use of the full power of the base language and *b)* to customize the

core semantics of a trace-based mechanism using *open implementations* [Kiczales et al., 1997a]: guidelines to control details of implementation of a system. To validate this thesis, we develop a typed functional description of our model in Typed Racket [Tobin-Hochstadt and Felleisen, 2008] and a concrete and practical implementation for JavaScript. This last implementation allows us to empirically validate the benefits of our model through a number of Web applications.

How to Read this Dissertation

Chapter 2 concisely defines the thesis of this dissertation. In addition, the chapter presents the hypotheses chosen to validate our thesis. Chapter 3 is dedicated to the presentation of the necessary background concepts to the discussion of existing related work in the area of aspect-oriented programming (Section 3.1), context-aware systems (Section 3.2), trace-based mechanisms (Section 3.3), and open implementations (Section 3.4). This chapter concludes this introduction by elaborating on the problem statement of this thesis work (Section 3.5).

The core of this dissertation is divided into two parts. Part I presents the steps to achieve our model, while Part II presents two case studies of the application of our model.

Chapter 4 presents the base of our work: AspectScript, an expressive aspect language for JavaScript. This aspect language adopts higher-order programming and dynamicity as its core design principles. In AspectScript, pointcuts and advices[1] are standard JavaScript functions, bringing the benefits of higher-order programming patterns to define aspects. Chapter 5 presents a first step of our work: ETM, an Expressive Trace-based Mechanism. In ETM, patterns (instead of pointcuts) and advices are also standard JavaScript functions. patterns are used to modularly detect program executions that represent contexts, and advices are used to modularly adapt the behavior of the software. ETM is currently implemented as a seamless AspectScript extension. Chapter 6 presents the main contribution of this thesis: OTM, the open implementation of ETM. The core OTM semantics is customizable to satisfy the specific needs of the developers.

In Part II, chapters 7 and 8 discuss two case studies of OTM. In both case studies, we develop a piece of software based on OTM. Chapter 7 presents WeCa, a practical library to modularly and flexibly address causality issues on the Web. The distributed causal relations between interactions of Web applications are contexts that need to be detected by WeCa. Chapter 8 presents ECOCAM, a framework to support adaptative software for primary students. The mathematical skills to solve operations like additions are contexts that need to be detected by ECOCAM.

[1] During this dissertation, we use the word "advices" instead of "pieces of advice" for simplicity.

Finally, Part III concludes with a summary of the contributions of this thesis and perspectives for future research.

Finally, we provide two appendixes. Appendix A presents a table with a summary of the API of OTM, and Appendix B shows the typed functional description of OTM in Typed Racket.

Note for the reader

It is recognized that the open implementation design is an inherently iterative process [Kiczales et al., 1993]. Our experience with building an open trace-based mechanism is another testimony of this fact. As the understanding of the issues at stake evolves, and the domain supposedly covered by the implementation is extended, the customization interfaces are adapted and refined as well. Therefore, this dissertation reports on two iterations the design and implementation of OTM, and the APIs presented in this dissertation do not exactly correspond to the current implementation of OTM. As a matter of fact, OTM is bound to continue evolving in the future.

This dissertation follows the aspect-oriented programing terminology instead of using the event-based programming terminology [Luckham, 2001]. For example, we use the term "patterns" to refer to "complex events".

Finally, we assume that the reader is well aware of dynamic prototype-based programming with higher-order functions in general. Most of our experiments in this dissertation are carried out with the JavaScript language. However, JavaScript should not be an obstacle for the reader to understand experiments. Finally, our results are not JavaScript-specific because we provide a semantic artifact of OTM.

Availability. OTM and associated proposals, along with the examples presented in this dissertation, are available online at http://pleiad.cl/otm. The implementation of OTM for JavaScript supports the Mozilla Firefox browser without the need of an extension.

What this is Not About

Although this thesis is concerned with addressing applicability issues of open implementations of a trace-based mechanism, methodologies about analysis and design of open implementations are beyond the scope of this dissertation. In addition, this thesis is definitely experimental, rather than theoretical, and hence does not propose a strong formal treatment of the functional

description of OTM. This dissertation focuses on using and opening a trace-based mechanism, leaving concerns of optimization as future work.

There is a large body of literature about the meaning of context [Strang and Linnhoff-popien, 2004], however, the discussion of this topic is beyond the scope of this dissertation. Finally, a detailed study about the real use of context-aware software is also beyond the scope of this dissertation.

Chapter 2

Thesis and Hypotheses

2.1 Thesis

A model of an expressive and open trace-based mechanism allows developers to encompass existing trace-based mechanisms and makes it possible to define new semantic variants.

A trace-based mechanism supports the definitions of monitors that detect and react to the software execution[1]. Most proposed trace-based mechanisms [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Martin et al., 2005, Ostermann et al., 2005, Eugster and Jayaram, 2009] have specifically been tailored to address particular domains (*e.g.* security, error detections, context-aware systems, etc). As Figure 2.1a shows, a particular trace-based mechanism allows developers to modularly create a monitor with the specification of a pattern of events and an adaptation. A monitor executes a detecting process to detect the pattern. Whenever the pattern is detected, a monitor executes a reacting process, which invokes the adaptation. Depending on the domain, a particular trace-based mechanism creates monitors with *a)* a pattern expressed by a limited domain-specific language and *b)* processes of detecting and reacting that have fixed semantics. This thesis proposes a model of an expressive and open trace-based mechanism. In order to achieve this model, patterns are flexibly expressed through the use of base language and the core semantics of the processes of detecting and reacting can be customized. To validate the thesis, we develop a typed functional description in Typed Racket [Tobin-Hochstadt and Felleisen, 2008] and a concrete and practical implementation of our model for JavaScript. Despite this model being continuously updated, the current implementation for JavaScript can already be used by a large number of potential developers in the realm of JavaScript applications.

[1] This and the other definitions of this chapter are refined in the next chapter.

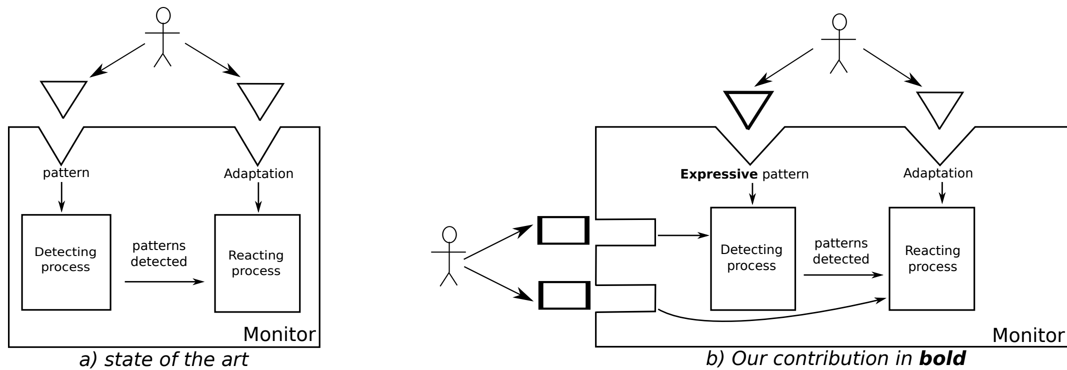

Figure 2.1: *a)* A programmer provides the pattern and adaptation to create a monitor. *b)* Apart from an expressive pattern and adaptation, the programmer can provide the core semantics of the detecting and reacting processes.

2.2 Hypotheses

In an ideal scenario, we would evaluate our model by instantiating and combining all existing trace-based mechanisms semantics, and developing several applications that uses the current implementation of our model. However, an evaluation on this scale is beyond this dissertation. We choose as validation *a)* the *expressiveness* to define a pattern and *b)* the *openness* (of the processes of detecting and reacting) to create and combine the semantics of existing trace-based mechanisms [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Martin et al., 2005, Ostermann et al., 2005, Eugster and Jayaram, 2009]:

Expressiveness. The trace-based mechanism allows developers to expressively define the pattern of events of a monitor.

We support this hypothesis by developing a trace-based mechanism model that uses the base language to express the pattern of events, which is commonly Turing complete.

Openness. The trace-based mechanism allows developers to customize the core semantics of a monitor.

We support this hypothesis by developing a trace-based mechanism model that opens the processes of detecting and reacting of a monitor. As Figure 2.1b shows, we open these processes to allow developers to customize their semantics.

Chapter 5 explains in detail how ETM, an Expressive Trace-based Mechanism, uses the power

of the base language to express a pattern of events, and Chapter 6 explains in detail how OTM, an Open Trace-based Mechanism, allows developers to customize the core semantics of a monitor.

Chapter 3

Context-Aware Systems, Trace-Based Mechanisms, and Open Implementations

This chapter reviews the state-of-the-art of the four themes surrounding this thesis work: aspect-oriented programming, context-aware systems, trace-based mechanisms, and open implementations.

This first section is dedicated to aspect-oriented programming because it is the basis of our thesis. Then, we briefly discuss context-aware systems and their possible crosscutting concerns. Since we are interested in this dissertation in opening a trace-based mechanism, Sections 3.3 and 3.4 discuss the area of trace-based mechanisms and open implementations respectively. Finally, Section 3.5 revisits the problem statement of this thesis in the light of the concepts, approaches, and issues presented in this chapter.

3.1 Aspect-Oriented Programming

This section briefly introduces Aspect-Oriented Programming (AOP) [Kiczales et al., 1997b] to establish the necessary concepts to understand our thesis work. Section 3.1.1 describes the origins of AOP, and Section 3.1.2 key concepts surrounding AOP.

3.1.1 Origins

Nowadays, an *aspect* is the modularization of a *crosscutting concern*: a concern whose implementation is tangled with other concerns and scattered through several parts of a system. However, the definition of an aspect was different in its origins.

The term aspect is coined in [Kiczales et al., 1997b], which denotes the issues addressed by design decisions which are hard to modularize in existing entities such as objects. Kiczales *et al.* argue that the issues related to aspects are inherently hard to modularize because of their nature: they, by essence, crosscut the basic functionality of a program. As a result, there are modules of the system that get "polluted" with code dedicated to address such aspects, therefore, the understandability and reusability of code is diminished.

AOP is proposed as a programming paradigm to cleanly express programs with such aspects, including isolation, composition, and reuse of aspect code. Through the task of implementing of a concern, Kiczales *et al.* also make precise that an aspect is different from a *component*:

- A concern is *"a component, if it can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API)"*.
- A concern is *"an aspect, if it cannot be cleanly encapsulated in a generalized procedure"* [Kiczales et al., 1997b].

By "cleanly encapsulated", the authors mean that a concern is well-localized and composed as necessary. From the text, it is possible to note that whereas components tend to be units of the functional decomposition of a program, aspects tend not to be so. Aspects, rather tend to be properties that affect semantics of the (several) components.

3.1.2 Concepts of an Aspect Language

This section reviews key concepts of any aspect language. First, this section reviews the notion of *join point models*. Then, the widely-known *Pointcut and Advice* (PA) model is reviewed.

3.1.2.1 Join Point Models

In [Masuhara et al., 2002], the authors discuss that a join point model consists of the three elements:

- The *join points*, which are the points of reference that an aspect can affect.

- A means of *identifying* join points.
- A means of *effecting* at join points.

Join points may be *dynamic* if they denote evaluations of expressions at runtime, such as method calls, method executions, objects and functions creations, etc. In addition, join points may be *lexical* if they simply refer to locations in the code, such as expressions. Apart from these kinds of join points, they could denote something else, for example, data flows in the component program [Kiczales et al., 1997b], a variable-sized piece of code [Rajan and Leavens, 2008] (*e.g.* loops), etc. Most AOP proposals use a join point model similar to AspectJ [Kiczales et al., 2001], an aspect language for Java, where join points are execution points in the execution of a program.

Identifying join points is typically done in a declarative manner. Whereas most AOP proposals use a domain-specific language [Kiczales et al., 2001, Aracic et al., 2006] to identify join points, other proposals like AspectScheme [Dutchyn et al., 2006] use the base language, which is Turing complete. Effecting at join points is commonly done in a procedural manner using the base language, possibly extended (*e.g.* the `proceed` construct of AspectJ).

3.1.2.2 The Pointcut and Advice Model

Various models to modularize crosscutting concerns in AOP have been proposed, for example, traversal specifications in Demeter [Lieberherr and Silva-Lepe, 1994], class compositions as in Hyper/J [Ossher and Tarr, 2001], and open classes as in MultiJava [Clifton et al., 2000]. This section reviews the Pointcut and Advice (PA) model, which is the most used nowadays. In [Masuhara et al., 2003], Masuhara *et al.* propose the operational semantics of a semantics-based compilation model for an aspect language that follows the PA model. We now describe the PA join point model:

Join points. Join points are execution points during the execution of a program. Maushara *et al.* propose the method calls, method executions, object creations, and advice executions as kinds of join points. Currently, many AOP languages are richer regarding the kinds of join points, *e.g.* field reads/writes, variables reads/writes, exceptions, initializations, etc.

Pointcuts. The means of identifying join points is the *pointcut* mechanism: a predicate on join points, used to identify any join point that an advice should affect. Like in AspectJ [Kiczales

et al., 2001], Maushara *et al.* propose pointcuts like call(m), execution(m), new(m), target(t v), and args(t v,...), and operator pointcuts like (&&, ||, and !). Most AOP proposals use these pointcuts (and others), which are expressed as patterns in some domain-specific language. Other proposals like AspectScheme propose pointcuts as functions that take a join point, which contains the necessary information to be identified, and return some value different from false to match[1] this join point.

Advices. The means of effecting the execution at join point is the *advice* mechanism. In most AOP proposals, an advice is composed of a pointcut and a block of code, which is executed when the associated pointcut matches a join point. Maushara *et al.* propose two types of advices: *before* and *after*. Before advice runs before the original action takes place, and after advice runs after the completion of the original action. Most of existing AOP proposals also include the around advice (*e.g.* AspectJ [Kiczales et al., 2001]), which is executed around the matched join point.

An Aspect Example: logging of users

The piece of code below is written in AspectJ. This piece of code shows that after calls to the login method of the LoginManager class, an advice logs the entered of a user. In this piece of code, the pointcut is call(..) && args(..) and the advice corresponds to the construct before and the block of code. The join point is implicitly passed to the pointcut and advice.

```
after: call(* LoginManager.login(User,Password)) && args(user,..) {
  log("login of the user:" + user); }
```

3.2 Context-Aware Systems

This section briefly defines the crosscutting concerns in the development of context-aware systems. This section begins with the description of context-aware systems. Then, Section 3.2.2 describes how crosscutting concerns are present in these systems and how an aspect language can modularize these concerns.

3.2.1 Origins and Concepts

In 1994, Schilit *et al.* in [Schilit et al., 1994] coin the term context-aware system. The authors define a context-aware system as a system that observes and reacts to changes of contexts.

[1] The current AOP terminology uses the verb “match” instead of “identify”. We use “match” in this dissertation.

These systems can promote people interactions with devices, computers, etc. Later in 2001, Satyanarayanan in [Satyanarayanan, 2001] describes scenarios and requirements of context-aware systems:

Scenarios. According to Satyanarayanan, there are many scenarios where context-aware systems can be applied: smart homes [Augusto and Nugent, 2006], e-learning [Razek et al., 2003], health-related applications [Zita et al., 2002], distributed applications [Benerecetti et al., 2001], etc. The defining characteristic of these scenarios is the *adaptation* of the behavior of a system when a certain *context* is detected. For example, in e-learning, a system changes the difficult of the mathematical exercises (adaptation) when this system detects the student's skills (context) to solve the exercises.

Requirements. Satyanarayanan describes context-aware system requirements, which can be classified in two categories: hardware and software. From a hardware viewpoint, the autonomy, the size, and the power of processing of the hardware of a context-aware system should be enough to analyze events generated by the system execution and adapting the behavior of it. From a software viewpoint, a context-aware system should be able to follow these steps:

1. **Listen and record events.** The system should have sensors that enable it to gather every event produced in the execution of the software.
2. **Analyze the event history.** The system should have a mechanism to analyze the history of events in order to detect a certain context. This mechanism has the responsibility to detect a context in a precise manner.
3. **Adapt behavior of the system.** If a context is detected, the system should have a mechanism to adapt to the system behavior, which satisfies the requirements of the context detected.

3.2.2 Crosscutting Concerns in Context-Aware Systems

In 2007, Augusto [Augusto, 2007] presents some designs of context-aware systems. The common property among these designs is the crosscut modifications in a system to support context awareness. Figure 3.6 shows crosscut modifications in a context-aware system. Developers have to manually insert several pieces of code that correspond to *sensors*, an *analyzer of events*, and

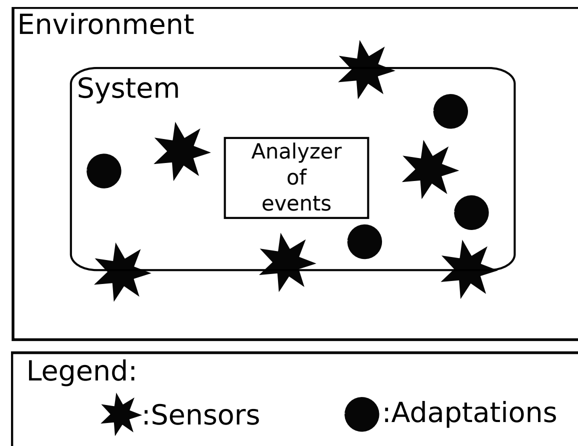

Figure 3.1: Modifications of a system to support context awareness.

adaptations. Sensors are used to gather events that occur inside/outside of the system, and the analyzer of events is used to determine if the system must trigger an adaption when a certain history of events occurs. Finally, adaptations are used to support the adaptation of the system. To address these issues, Augusto proposes something similar to a trace-based mechanism, which is used to modularize these crosscutting concerns. More recently, there are research has been performed, which concretely relates both AOP abstractions and context-aware systems [Ghezzi et al., 2011].

3.3 Trace-based Mechanisms

This section briefly introduces trace-based mechanisms using AOP terminology, which lies at the core of this dissertation. Section 3.3.1 describes the origins of these mechanisms. Section 3.3.2 introduces key concepts of these mechanisms. Section 3.3.3 exemplifies a stateful aspect with a widely-known trace-based mechanism. Section 3.3.4 describes some of these mechanisms, and Section 3.3.5 discusses the drawbacks of them.

3.3.1 Origins

Douence *et al.* [Douence et al., 2001, Douence et al., 2005] are pioneers that studied the surrounding topics on trace-based mechanisms; in particular, they discuss the limitation of an aspect to cleanly match the expected execution point where its advice should apply. For instance, it is often necessary to use various temporary aspects, which only pass information to each other, to

correctly trigger the advice of an aspect. Concretely, consider an aspect, named *intruder detector*, that prevents that a malicious application from inserting (random) credentials three or more times to log-in a user in a system. To correctly trigger the advice, the intruder detector aspect needs the other two temporary aspects to know when the malicious application tries inserting the credentials for the third time. This is because the aspect cannot cleanly refer to the whole history of a program execution. Douence *et al.* coin the term *stateful aspect* to refer to a monitor that detects and reacts to a program execution trace[1].

Trace-based mechanisms have numerous applications in domains like error detections, security flaws, and modular definitions of crosscutting concerns. Various of these mechanisms have been proposed, specifically tailored to address a particular domain [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Martin et al., 2005, Ostermann et al., 2005].

3.3.2 Concepts of a Trace-based Mechanism

This section reviews key concepts of a trace-based mechanism. First, this section reviews the notion of *patterns*, which is used instead of pointcuts. Although aspects are similar to stateful aspects, there are strong differences between them; hence, this section ends with the discussion of these differences.

3.3.2.1 Pattern Language

A stateful aspect is defined by a pattern and an advice. Using a language, a pattern is defined to match a *trace* of join points, and the advice specifies the piece of code is executed when the last join point of this trace is matched. In a high-level view, aspects are similar to stateful aspects, the only difference is that a stateful aspect uses a pattern instead of a pointcut, which identifies a single join point.

Figure 3.2 shows a stateful aspect, defined by a developer, that matches a trace of join points. The developer defines a stateful aspect in terms of a pattern and an advice. When the stateful aspect is defined and deployed, it monitors the trace of join points to identify whether or not the defined pattern matches this trace. If the pattern is matched by the trace, the stateful aspect executes its advice before/around/after the last join point matched.

[1] To follow the AOP terminology, we use the term “stateful aspect” instead of “monitor” during this dissertation.

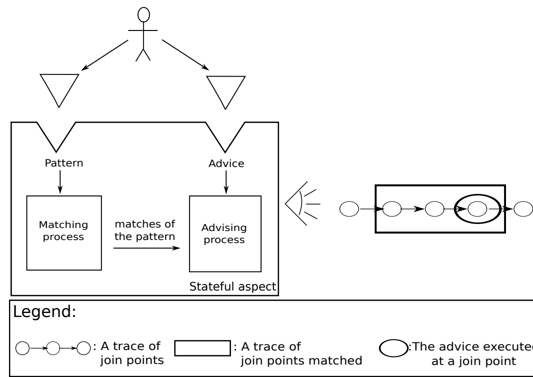

Figure 3.2: A programmer expresses a stateful aspect and later this stateful aspect matches a trace of join points.

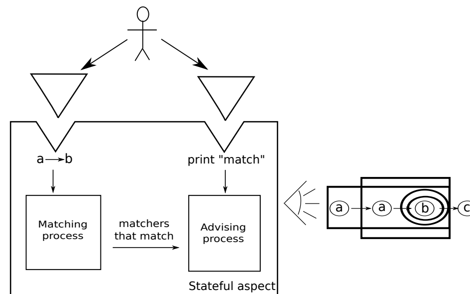

Figure 3.3: A stateful aspect matches twice the trace $a \to b$.

As an example, Figure 3.3 shows a developer that defines a stateful aspect that prints the message “match” when it matches the trace of join points $a \to b$ (*i.e.* a then b). The stateful aspect uses a *matching process* to match the trace specified by the pattern. Depending on the semantics of the matching process, two, one, or even zero matches can occur when the b join point of the trace $a \to a \to b \to c$ is matched by the stateful aspect shown in the figure. When one or more times the pattern is matched at the same time, the stateful aspect uses an *advising process* to execute the advice for every match of the pattern. Depending on the semantics of the advising process, two, one, or even zero advice executions can occur. In the next two sections, we detail how the processes of matching and advising work.

3.3.2.2 The Matching Process

The matching process matches the pattern of a join point trace. When a stateful aspect is deployed, the matching process creates a *matcher*, which is a mechanism used to match the trace of join points specified by the pattern. Apart from matching, a matcher gathers context

information from join points of the trace (*e.g.* parameters of the call to a method). Depending on the matching process semantics, this process can create several matchers, allowing a stateful aspect to match multiple times, even at the same time. For example, Figure 3.3 shows that a stateful aspect creates two matchers that match in the same trace of execution.

3.3.2.3 The Advising Process

Every time a matcher matches, the advising process executes the advice with context information gathered by this matcher. If several matchers match at a time, the advising process composes and executes the advices. For example, Figure 3.3 shows that the stateful aspect matches twice with the *b* join point and executes the advice twice, each one with context information gathered by its corresponding matcher. The composition and executions of advices depend on the advising process semantics.

3.3.3 A Stateful Aspect Example: intruder detector

The piece of code below is written in tracematches [Allan et al., 2005], a widely-known trace-based mechanism for Java (more on this in Section 3.3.4). This piece of code is the stateful aspect that implements the intruder detector mentioned in Section 3.3.1. The stateful aspect triggers an exception before the fourth call to the login method with the same user. The pattern is defined as four calls to login (*i.e.* login [4]) , and the user binding is defined as context information of the pattern using AspectJ style. The advice is the block of code that triggers the exception. If different users are trying to enter, the pattern is matched for each user.

```
tracematch(User user) {
  sym login before: call(* LoginManager.login (User,..)) && args(user,..);

  login [4] {
    throw new IntruderDetectedException();
} }
```

3.3.4 Existing Trace-based Mechanisms

In [Douence et al., 2005], Douence *et al.* propose a formal framework, where the distinctive characteristic of the framework is that a stateful aspect can execute different advices while a pattern is matching a trace. Apart from the Douence proposal, there is a large body of literature of trace-based mechanisms, which are reviewed in this section.

Tracematches. It is a widely-known and efficient trace-based mechanism. Tracematches [Allan et al., 2005], an AspectJ extension, allows developers to define stateful aspects (*a.k.a.* tracematches) to match traces of join points of a Java program execution. The pattern of a stateful aspect is expressed by a regular expression language, where the alphabet of this language is composed of symbols that correspond to AspectJ pointcuts. The matching process is implemented through a nondeterministic finite-state automaton, whose active states correspond to matchers (with different contextual information) of a stateful aspect. The advising process follows the guidelines of AspectJ: when two or more aspects match the same join point, the advice executions are chained and nested.

Alpha. Alpha [Ostermann et al., 2005] is an aspect-oriented extension of L2, a simple object-oriented language in the style of Java. Alpha uses Prolog queries to express patterns. The matching process is implemented through queries to a database that contains information about the static representation (*e.g.* abstract syntax tree) and the dynamic representation (*e.g.* trace of execution) of a program. The matching process corresponds to the internal process that Prolog realizes to answer a query. Every solution to a Prolog query corresponds to a matcher of a stateful aspect that matched a program execution trace. These solutions are passed to the advising process, which executes the same advice with the set of bindings gathered by the match of the query (*i.e.* pattern).

HALO. Herzeel *et al.* [Herzeel et al., 2006] propose HALO, a Common Lisp extension. The HALO proposal allows developers to use the base language to express patterns, which are first-class values. The matching process is implemented with the Rete algorithm [Forgy, 1982], an efficient pattern matching algorithm used for expert systems [Darlington, 2000]. In Rete, patterns are represented as rules that must be satisfied by join points matched. The advising process only executes the advice with each set of bindings gathered by each satisfaction of the rule. The advice can be executed before or after the last join point matched.

EventJava. EventJava [Eugster and Jayaram, 2009] allows developers to execute a piece of code when a set of distributed events (*i.e.* join points) has a correlation specified by developers. To specify the correlation, every distributed event contains a set of properties available to developers (*e.g.* the time at which the event is observed). For the matching and advising processes, EventJava

follows the same guidelines of HALO, but the advice can only be executed after the last join point matched.

Tracecuts. Tracecuts [Walker and Viggers, 2004] is a trace-based mechanism that works for Java as an AspectJ extension. This mechanism is used to check the use of protocols (*e.g.* FTP [Postel and Reynolds, 1985], a communication protocol). In tracecuts, if a pattern of join points does not follow a certain protocol, an action can be triggered. According to the tracecut authors, the checking of some protocols needs to properly identify the nested entries and exits of the executions of different methods of a class. This feature is reducible to recognition of properly nested parenthesis, meaning that a finite state machine cannot correctly check the use of these protocols. Therefore, tracecuts allow developers to express patterns using a context-free language. The matching process uses a pushdown automaton, and the advising process follow the same guidelines of the tracematches.

AWED. AWED is a language for Aspects with Explicit Distribution [Navarro et al., 2006, Benavides Navarro et al., 2008]. This aspect language explicitly supports the monitoring of distributed computations in Java. In addition, this aspect language takes into consideration distributed causal relations in tasks of debugging and testing of middleware. Similarly to tracematches, AWED patterns are expressed using a domain-specific language: a regular expression language. Similarly to tracematches, the matching process uses a finite state machine carry out the match of patterns. For the advising process, AWED follows the same process of EventJava.

PQL. Program Query Language (PQL) [Martin et al., 2005] is a tool to detect errors and check-/force protocols of programming (*e.g.* file handling). This tool uses a static analyzer to reduce the possible matches and then use a dynamic matcher that really matches the given pattern. A developer expresses a pattern describing the AST (using Java-like syntax) of program execution that should be matched. The dynamic matcher carries out the matching process through a state machine. The execution of the advising process is optional, a programmer can choice to execute the advice using the construct `replace` and `execute`. If the developer uses `execute`, the advising process can be opened. In practice, the previous point means that a method is executed, which takes an array of a set of bindings gathered by each match of the pattern.

PTQL. Program Trace Query Language [Goldsmith et al., 2005] is a tool to detect errors.

Developers use the SQL language to express a pattern, which is actually a SQL query. Join points are stored in databases, which are used by the matching process to match the query. PTQL does not allow developers to take actions if pattern is matched, *i.e.* there is no an advising process in PTQL.

JavaMOP. JavaMOP [Chen and Roşu, 2003, Chen and Roşu, 2007, Meredith et al., 2011] is a generic and efficient runtime-verification framework for Java. Patterns in JavaMOP can be expressed in different (previously defined) domain-specific languages: regular expressions, context-free grammars, linear temporal logic, string rewriting system, etc. In addition, a set of matching process semantics is available for the developers. As JavaMOP compiles their code to AspectJ code, the JavaMOP advising process follows the guidelines of AspectJ for its process.

The main concern of existing trace-based mechanisms is their performance. To achieve this concern, the expressiveness and customizations have been sacrificed in these mechanisms. These sacrifices bring some drawbacks, which are mentioned in the next section.

3.3.5 The Limitations of Existing Trace-based Mechanisms

The proposals mentioned in previous section are limited in terms of expressiveness and openness. We illustrate the limitations with a familiar example for trace-based mechanisms: the *autosave* feature [Allan et al., 2005]. Consider the addition of the autosave feature of a text editor application; this feature automatically saves the document every three editions. This feature is clearly a crosscutting concern that can be modularized through a stateful aspect:

```
tracematch() {
  sym edit before: call(* *.edit (..));

  edit [3] {
    Editor.save();
} }
```

An initial implementation in tracematches for the autosave feature is shown above. However, tracematches, like most trace-based mechanisms that *unalterably* support multiple matches of a pattern, fail. Once three `edits` happen, each subsequent edit triggers a save. This is so because a match occurs when a suffix of the current program trace is a symbol in the regular language specified by the pattern. The programmer has to code around the aspect definition to artificially introduce another symbol, `save`, which is then excluded from the regular expression (tracematches require contiguous occurrences of the events denoted by the symbols in the regular expressions):

```
tracematch() {
  sym edit before: call(* *.edit (..));
  sym save before: call(* *.save (..));

  edit [3] {
    Editor.save();
} }
```

The current trace-based mechanisms require to code around their stateful aspects in order to appropriately work. These modifications are required by limited expressiveness and openness of these mechanisms. This section discusses these limitations based on key elements of a trace-based mechanism.

3.3.5.1 Pattern Language

This section presents the limitations of the pattern language used in existing trace-based mechanisms. The pattern language limitations are split into three categories: *reusability*, *binding expressiveness*, and *pattern expressiveness*. Reusability measures how reusable and composable are patterns defined in a language. Binding expressiveness measures the expressiveness that has a developer to gather bindings during the match of a pattern. Finally, pattern expressiveness measures the expressiveness to define a pattern (more of these categories in Chapter 5).

Figure 3.4 presents some trace-based mechanisms and their limitations regarding the pattern language used. Each trace-based mechanism is evaluated under the three categories, and each evaluation shows the level of support to a category: none, basic, and full.

Reusability. As the figure above shows, patterns of most trace-based mechanisms are not reusable. HALO patterns are first-class values, which can be reused. Although PQL patterns are not firs-class values, PQL supports the reuse and composition of patterns. In Alpha, patterns are expressed in a different Turing complete language: Prolog. However, these patterns cannot be named and parametrized using values of the base language (a subset of Java). The pattern parameterization is useful to define patterns that depend on bindings that cannot be directly gathered by these patterns (*e.g.* patterns supervise the behavior of objects that come from untrusted/unknown applications).

Binding expressiveness. Patterns of most trace-based mechanisms cannot gather bindings out of the attributes of the current join point. Patterns in HALO, tracematches, and tracecuts

Trace-based mechanism	Reusability	Binding expressiveness	Pattern expressiveness
Tracematch	○	◐	○
Alpha	◐	○	◐
HALO	●	◐	◐
EventJava	○	○	○
PQL	●	○	◐
PTQL	○	○	○
MOP	○	●	◐
Tracecuts	○	◐	◐
AWED	○	○	○

Legend	Reusability	Binding expressiveness	Pattern expressiveness
○	Patterns are not reusable	Only values from the join point scope	A limited & fixed pattern definition
◐	Patterns are reusable but *poorly flexible* to compose	values from or out of the join point scope	An expressive but fixed pattern definition
●	Patterns are reusable and can be composed	Lists of values from or out of the join point scope	An expressive & adaptable pattern definition

Figure 3.4: Limitations in terms of expressiveness of existing trace-based mechanisms

can gather bindings beyond the join point; HALO uses the construct escape, and tracematches and tracecuts use the pointcut let, which is available in the *abc* compiler[1] for AspectJ. However, HALO patterns, like other trace-based mechanisms, cannot gather lists of bindings (*e.g.* (a^{v})*). In JavaMOP, every time that a join point is matched, it is possible to execute an advice, which allows developers to use instance variables of the stateful aspect to manually store in a set of bindings (*i.e.* developers hardcode advices to become stateful advices).

Pattern expressiveness. Several trace-based mechanisms can use the power of a Turing complete language to define patterns. It is important to mention that JavaMOP supports a Turing complete pattern language with the the SRS plugin[2] (String Rewriting Systems). However, there is no trace-based mechanism that allows developers to alter/adapt the definition of patterns at runtime.

[1] http://www.sable.mcgill.ca/abc/
[2] http://fsl.cs.uiuc.edu/index.php/Special:SRSPlugin3. This work is currently under revision in FSE 2012.

3.3.5.2 Matching and Advising Processes

The tweak used for the autosave example illustrates the contortive and necessary modifications that patterns (or advices in trace-based mechanisms) must suffer in order to achieve the behavior needed in a stateful aspect. These modifications are necessary because existing trace-based mechanisms does not allow developers to customize semantics of the internal processes (matching and advising) of their stateful aspects.

To the best of my knowledge, JavaMOP and PQL are only trace-based mechanisms that allow developers to customize their internal processes. JavaMOP developers can select a matching process for a stateful aspect. However, developers can only select a limited and fixed set of matching processes. PQL developers can completely write the advising process because the PQL interface only needs a method that takes an array of a set of bindings. In Chapter 6, we illustrate in detail the drawbacks of stateful aspects with fixed matching and advising processes.

3.4 Open Implementations

This section briefly introduces open implementation guidelines [Kiczales et al., 1997a]. Section 3.4.1 presents the need to follow these guidelines. Section 3.4.2 presents the origins and concepts of open implementations, and Section 3.4.3 explains how the expressiveness of trace-based mechanisms can improve if their implementations follow the guidelines of open implementations.

3.4.1 Why Should a System Be Open?

The widely-known *black-box* principle proposes that the design of a system should expose its functionality but hide its implementation. Therefore, if we follow black-box principles, the issues of any concrete implementation of a system should be completely hidden due to the fact that they are not part of the concerns of developers. However, any concrete implementation of a system requires fixing a number of tradeoffs [Kiczales, 1992]. In the context of programming languages, this was first noticed by Wirth:

> *"I found a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details"."* [Wirth, 1974].

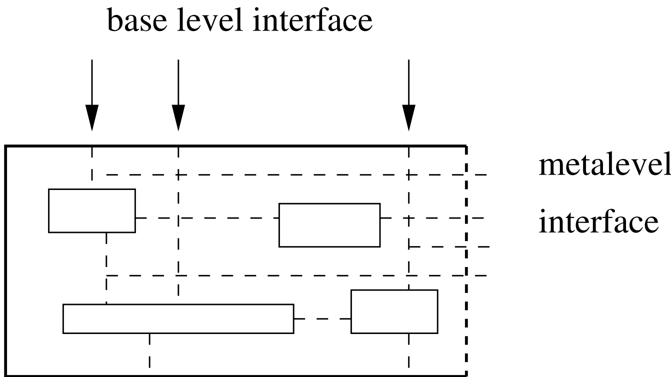

Figure 3.5: A system that supports open implementations (From [Rao, 1991]).

To illustrate the need to control the implementation details, consider a system with two classes: Position and Person. Kiczales *et al.* in [Kiczales et al., 1997a] affirm that the implementation of Position with two instance variables x and y should be different from the implementation of Person with a hundred instance variables. For the Position class, an array-like strategy is ideal, providing compact storage and quick access to both variables. Instead, for the Person class, a hashtable-like strategy would be more appropriate, avoiding the allocation of a large amount of memory when it is highly probable that not all variables will be used.

3.4.2 Origins and Concepts

The term *open implementations* was coined for the first time by Ramana Rao in [Rao, 1991]. The author establishes this term as a reformulation of the *reflective architecture* [Maes, 1987]:

> *"A system with an open implementation provides (at least) two linked interfaces to its clients: a base-level interface to the functionality of the system similar to the interface of other such systems, and a metalevel interface that reveals some aspects of how the base-level interface is implemented."*

The role of the base interface is to provide developers with a simple and natural interface for the use of a system, so they do not have to understand how the system works. Instead, the role of the meta interface is to provide developers with an interface to customize part of the semantics of the system. Here, developers are aware of how the system should work. Not all the semantics of a system is customizable via the meta interface. However, the customizable part is sufficient to allow developers, for example, to improve the performance of the system for a specific scenario.

In other words, the meta level interface allows developers to adapt a system in order to satisfy their specific needs.

Figure 3.5 shows how both interfaces (base and meta) interact to support open implementations. The meta interface can alter the behavior of the components of the base interface. Nevertheless, the design of both interfaces is independent.

3.4.3 Open Implementations and Trace-Based Mechanisms

As described in Section 3.3.4, existing trace-based mechanisms have fixed semantics and there is no trace-based mechanism that follows the open implementation guidelines. A trace-based mechanism that would follow these guidelines allows developers to customize its semantics in order to define the needed expressiveness of a stateful aspect to match a trace of join points.

3.5 Problem Statement

As mentioned in Section 3.2, context-aware systems have to be modified to intrusively insert pieces of code that correspond to sensors and adaptations. Section 3.3 presents trace-based mechanisms as a solution to modularize the crosscutting modifications of context-aware systems. We present trace-based mechanisms from an AOP viewpoint, meaning that these mechanisms deploy stateful aspects that execute advices when they match traces of join points. Trace-based mechanisms allow developers to express a pattern that is matched in a trace and an advice that is executed when the pattern is matched (Figure 3.6a). However, these mechanisms *a)* use a limited domain-specific language to express join point trace patterns and *b)* the semantics of their processes of matching and advising are fixed.

This thesis proposes a model of an open implementation of a trace-based mechanism, which encompasses existing trace-based mechanisms and makes it possible to define new semantic variants. Concretely, this model allows developers to *a)* expressively define the pattern of a stateful aspect and *b)* customize the semantics of its matching and advising processes.

Figure 3.6b shows a big picture of our model. In this dissertation, we explain this model in two stages. In the *a* stage, we explain how developers can expressively define patterns using the base language (Chapter 5). In the *b* stage, we explain how developers can customize the semantics of the processes of matching and advising (Chapter 6).

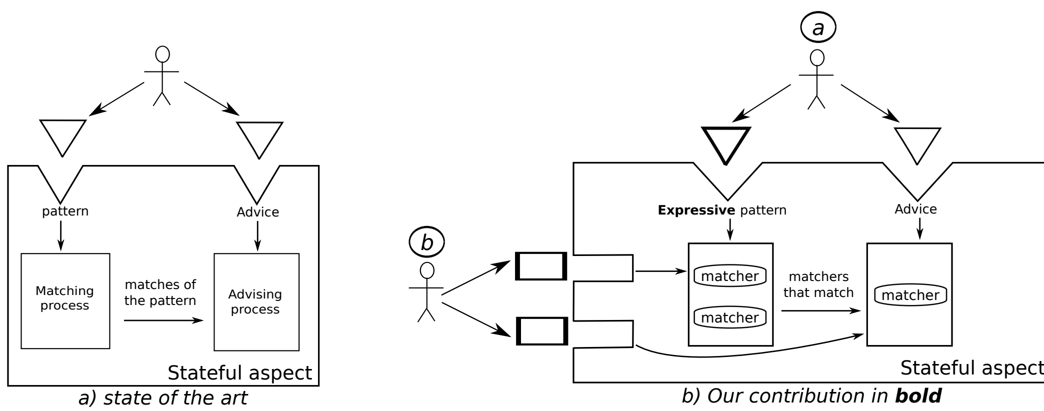

Figure 3.6: A refined version of Figure 2.1. a) Traditional view of trace-based mechanism. b) The trace-based mechanism view proposed by this thesis.

Part I: An Open Trace-based Mechanism

Chapter 4

AspectScript: Expressive Aspects[1]

This chapter presents AspectScript, an aspect language for JavaScript. This aspect language is extended in the following chapters to support a concrete implementation of OTM. AspectScript is a full-fledged AOP extension of JavaScript that adopts higher-order programming and dynamicity as its core design principles. In AspectScript, pointcuts and advices are standard JavaScript functions, bringing the benefits of higher-order programming patterns to define aspects. In addition, AspectScript integrates a number of state-of-the-art AOP features like dynamic aspect deployment with scoping strategies, and user-defined quantified events.

In this chapter, we illustrate AspectScript in action with several practical examples from the realm of client Web applications, and report on its current implementation. AspectScript is a practical extension that provides better modularity support to build Web applications, and that will eventually make it possible to empirically validate the benefits brought by advanced aspect language mechanisms in an ever-growing application domain.

Note. The author of this dissertation worked on earlier versions of the implementation of the core of AspectScript (*e.g.* weaver) and custom join points. The core implementation is essential to obtain a practical implementation of OTM and custom join points are used in Chapter 7.

4.1 Introduction

There is a clear trend in the software industry towards Web-based applications, as witnessed by the increasing number of popular Web-based applications like Facebook, YouTube, and Blogspot.

[1]The content of this chapter are based on our AOSD 2010 paper "AspectScript: Expressive Aspects for the Web" [Toledo et al., 2010].

For the development of these applications, the JavaScript language is one of the most used, mainly because almost all modern browsers support it. A consequence of this trend is that JavaScript, which was initially used only for small client-side scripting, is now used to build complex applications. In such applications, crosscutting concerns are likely to appear and end up being scattered at many places in the code, tangled with other concerns.

While modularization of crosscutting concerns has long been considered in Web technologies [Stamey et al., 2005] (*e.g.* separate CSS files for presentation style), there is not much support at the scripting code level. Aspect-Oriented Programming (AOP) [Elrad et al., 2001] addresses this issue by introducing new means of modularizing programs, in particular through the pointcut-advice mechanism. The potential benefits of AOP for JavaScript have already been identified and resulted in a number of aspect-oriented proposals for JavaScript [Ajaxpect, 2011, AspectJS, 2011a, Cerny, 2011, Humax, 2011, Prototype, 2011, Washizaki et al., 2009]. However, these proposals are fairly basic in that they at best attempt to mimick AspectJ [Kiczales et al., 2001] in JavaScript. However, beyond the fact that it is dynamically typed, JavaScript is a language that is also fundamentally different from Java. As a result, existing proposals fail to properly integrate with the characterizing features of JavaScript, *i.e.* a dynamic prototype-based object model with full support for higher-order functions.

In contrast, AspectScript builds upon advances in AO language design research to address the specificities of JavaScript in a novel way. Inspired by the work on aspects in higher-order procedural languages by Dutchyn, Tucker, and Krishnamurthi [Tucker and Krishnamurthi, 2003, Dutchyn et al., 2006], AspectScript supports first-class aspects; both pointcuts and advices are defined using first-class functions, providing the full benefits of higher-order programming. In line with this work and the inherently dynamic nature of JavaScript, AspectScript supports dynamic deployment of aspects, as found for instance in CaesarJ [Aracic et al., 2006] and AspectScheme [Dutchyn et al., 2006]. Also, AspectScript integrates scoping strategies [Tanter, 2008b, Tanter, 2009] for proper control over the scope of dynamically-deployed aspects. In addition, AspectScript avoids issues of infinite loops due to aspect reentrancy [Tanter, 2008a]. Finally, AspectScript not only supports implicitly-generated join points following the language model, but also provides the possibility to define custom join points triggered explicitly, as in Ptolemy [Rajan and Leavens, 2008]. This combination of features is unique in the space of current aspect languages.

This chapter puts a strong emphasis on the practical benefits of integrating these features in an AO extension of JavaScript. Section 4.2 therefore exposes several concrete examples addressed with AspectScript, progressively revealing how each feature comes into play. Section 4.3 then

reviews the main elements of the AspectScript language in a systematic manner. Section 4.4 presents key points regarding the implementation of the core of AspectScript.

Section 4.5 discusses other (higher-order) aspect languages for the Web and Section 4.6 concludes.

Availability. AspectScript is available online http://pleiad.cl/aspectscript and currently supports Mozilla Firefox, Opera, Chrome, and Safari.

4.2 AspectScript in Action

We introduce AspectScript through concrete examples. More precisely, we implement a number of extra functionalities to Facebook, a representative Web-2.0 application. For each of the five cases, we show how an aspect-oriented solution with AspectScript enables a modular and straightforward specification.

For conciseness, in the code fragments, we use the variables AS and PCs as abbreviations for AspectScript and AspectScript.Pointcuts respectively. All the examples are included with the distribution, and can be downloaded from the AspectScript website.

4.2.1 A Simple Example

In Facebook, users can freely tag people (supposedly) appearing in photos. If the user being tagged is a friend of the tagger, the tag becomes a link to the user profile, and the picture is added to the personal photo album. However, to avoid wrong tags, Facebook provides a way to remove them through a "remove tag" option.

For most cases, this simple solution is satisfactory. However, if a user has been wrongly tagged in hundreds of photos, he has to remove each tag individually. Let us devise an extension to Facebook that suggests an automatic removal of all tags once a user untags herself from a certain number of pictures (say 3) in a given album. This feature can be implemented as an aspect in AspectScript:

```
var pc  = PCs.exec(removeTag).and(nTimes(3)); // three times
var adv = function(jp) {
  var userId = jp.args[0]; //1st argument to removeTag
  var albumId = jp.args[1]; //2nd argument to removeTag
  showRemoveAllTagsDialog(userId, albumId);
};
AS.after(pc, adv); //deployment
```

This simple example illustrates how to define and deploy a simple aspect: defining a pointcut pc, an advice adv, and finally deploying the aspect. AspectScript provides AS.before/around/after(..) functions, which are syntactic sugar for manually creating and deploying an aspect with a particular advice kind. The aspect is deployed with global scope (more on this later).

The aspect matches three executions of the removeTag function. This function is parametrized by the id of the user, the id of the album, and the id of the photo (userId, albumId, and photoId respectively). The advice adv removes all tags of the user in the current album, after confirmation. The variables that identify the user and album are obtained from jp.args, which stores the values with which the function was executed. The pointcut pc is a conjunction of two pointcuts. The first is obtained using the exec pointcut designator (*i.e.* a function that returns a pointcut), and matches all executions of the removeTag function. The second pointcut is obtained using the nTimes pointcut designator. The and method has a lazy evaluation, that means that the right branch is only evaluated if the left branch matches the join point. The nTimes pointcut matches whenever it is evaluated a given number of times. The nTimes PCD is not part of the standard PCD library; it is defined as a higher-order function:

```
var pc.and = function(right) {
  var left = this;
  return function(jp) {
    if (left(jp))
      return right(jp);
  };
};

var nTimes = function(n) {
  var times = 0;
  return function(jp) {
    return ++times % n == 0;
  };
};
```

The actual conjunction of pointcuts is performed using the and method of the pointcut returned by exec(removeTag). This shows how pointcuts provided by AspectScript support a *fluent interface*[1] for operands such as or, and, and inCflow. Note how the combination of first-class functions and fluent interfaces allow for an extensible pointcut language within standard JavaScript syntax.

[1]Fluency is a structuring principle to make APIs closer to embedded DSLs, see http://martinfowler.com/bliki/FluentInterface.html

4.2.2 Pointcuts: Matching Sequences

While the previous aspect definition appropriately modularizes the new feature, it is a typical example of a *stateful* aspect [Douence et al., 2004], implemented with book-keeping code (the times variable in the pointcut). Recognizing this fact, we now present an alternative implementation that uses a general-purpose trace PCD for matching *sequences* of events:

```
var rt = PCs.exec(removeTag);
var adv = function(jp) { /* as above */ };
AS.after(trace(rt,rt,rt), adv);
```

Note that the pointcut rt only matches execution of removeTag, and that at deployment time, we actually specify that we are interested in a sequence of three such events.

The higher-order trace function is a pointcut designator that receives a list of pointcuts. This list specifies the sequence of join points to match. Interestingly, trace is also simply defined within AspectScript as follows[1]:

```
var trace = function() { //variable-arg function
  var state = 0;
  var pcs = arguments; //actual arguments
  return function(jp) {
    if(pcs[state](jp)) {
      if(state == pcs.length-1) {
        state = 0;    // reset state
        return true; //match
      }
      state++; //go to the next pointcut
    }
    return false;
};};
```

Note that the pointcut resets its state after matching the whole sequence (our previous definition with nTimes did not reset). The trace pointcut designator is mostly illustrative. It is far from a full-fledged tracematch mechanism in which sequences can be defined with regular expressions and free variables, and where multiple *instances* of a sequence can be matched simultaneously [Allan et al., 2005].

Exposing context information. In order to expose context information to either the advice

[1] In JavaScript, it is possible to omit the declaration of formal parameters. The actual parameters can then be accessed using the implicit variable arguments, as an array.

or other pointcuts, a pointcut can take as optional parameter an environment, and define new bindings in it. The pointcut returns the environment if it matches[1]. For instance, we can define a time PCD that always matches any join point and only extend the given environment with a new binding associating a given identifier to the current time:

```
function time(id) {
  return function(jp,env) {
    return env.bind(id, new Date().getTime());
  };
}
```

We can now revisit our simple trace PCD to allow pointcuts in a sequence to expose context information to be used in later pointcuts:

```
var trace = function() {
  //... as above
  var env = AS.emptyEnv;
  return function(jp) {
    match = pcs[state](jp,env);
    if(match) {
      env = match;
      if(state == pcs.length-1){ state = 0; return env; }
      // ... as above
```

Note how trace now passes a (local) environment to each pointcut in the sequence, and keeps the bindings defined by previous pointcuts. This progressively builds up a sequence-local environment. When the whole sequence finally matches, the environment is returned. We can use this enhanced trace PCD to express the fact that the removal of tagging advice is triggered only if the three tag removals are performed within a given time interval:

```
var timeDiff = function(jp,env) {
  return env.t1 - env.t0 < 10000; // 10 seconds
};

var seq = trace(rt.and(time("t0")),rt,rt.and(time("t1")));
var timedSeq = seq.and(timeDiff);
AS.after(timedSeq, adv);
```

We define a timeDiff pointcut that expects t0 and t1 to be bound in the environment and checks the time difference. We then define the sequence pointcut using trace, making sure that the first and third occurrences of rt bind the current time as expected. This brief example obviously does

[1]Returning true in a pointcut has the same meaning as returning the same environment originally passed to the pointcut.

not take into account all the intricacies of a proper tracematch mechanism, but it does illustrate the flexibility brought by first-class, user-definable pointcuts in AspectScript.

4.2.3 Giving Life to JavaScript Values

In Facebook, user pages are updated with the last messages from friends without the need to reload the page. To achieve this, Facebook uses Ajax [Garrett, 2005]: a set of Web development techniques for asynchronous client-to-server communication. However, when page elements (*e.g.* the last messages) depend on Ajax responses, they must be updated after every communication with the server to show up-to-date information. The complexity of Ajax updates can be alleviated by introducing basic support for *reactive values*: when a value originating from an Ajax response changes, all page elements depending on it are updated accordingly. Using AspectScript, we introduce a library for basic reactive values, *e.g.*:

```
var msgs = new ReactiveValue("lastMsgs.php?id=...", 3000);
newMessages.innerHTML = msgs;
```

We create a new reactive value msgs, passing the URL from which the value must be obtained, as well as the update interval. We then assign this value to the innerHTML property of the page element displaying new messages. Every time the reactive value is updated, the property is automatically updated.

The library has two elements: the ReactiveValue constructor and an aspect that takes care of the assignment of reactive values to page elements. The aspect, defined below, intercepts all assignment join points where the left-value is a page element and the right-value is a ReactiveValue object. Instead of proceeding, the around advice stores the join point in the reactive value itself. Therefore, the assignment proper does not happen yet.

```
var pc = function(jp) {
  return jp.isPropWrite() &&
    isDomElement(jp.target[jp.name]) && //left-value
    jp.value instanceof ReactiveValue;  //right-value
};
var adv = function(jp) {
  var reactiveValue = jp.value;
  reactiveValue.add(jp); //store jp for future executions
};
AS.around(pc, adv);
```

A reactive value therefore holds a list of assignment join points, corresponding to assignments to page elements that must be refreshed whenever the value changes. This is done each interval

milliseconds by scheduling the Ajax query. Whenever the new value of the Ajax query is ready, onreadystatechange is applied. This function simply invokes proceed on all the assignment join points:

```
function ReactiveValue(url, interval) {
  var ajaxResquest = ...; //new remote connection
  var jpList = []; //list of assignment join points
  function query() {
    //make request
  }
  this.add = function(jp) {
    jpList.push(jp);
  };
  ajaxRequest.onreadystatechange = function() {
    for(var i = 0; i < jpList.length; ++i)
      jpList[i].proceed(ajaxRequest.responseText);
  };
  //schedule query for repetition
  setInterval(query, interval);
}
```

As a result, all page elements that directly depend on a reactive value are reassigned (join points are re-proceeded) each time the value changes. Note that this simple library for reactive values does not support full-fledged reactive computation (including indirect dependencies), like Flapjax [Meyerovich et al., 2009]. This said, the library is still useful to address direct dependencies between values

4.2.4 Access Control with Scoping Strategies

Most modern Web applications allow third-party applications to provide extra functionality through an API. However, one of the most attractive features of Facebook is the ability to include them right *inside* Facebook pages, and since recently, third-party applications can use JavaScript to provide a richer user experience. Sadly, JavaScript can also be used by malicious applications to fool users. For instance, the following application tries to change the “home” link to point to an external page, identical to the login page of Facebook, thereby misleading the user to reinsert his access credentials:

```
var maliciousApplication = {
  fakeURL : '123.45.56.78/facebook.com',
  action : function() {
    var homeElem = ...;
```

```
    homeElem.href = this.fakeURL;
  } };
```

To avoid these kinds of applications, Facebook limits the area of the Web page that an external application can access. Suppose that a function isOutsideAppArea(elem) exists for checking whether a DOM element elem is outside the area bounds of an application page, and hence its access is forbidden[1]. Using AspectScript, we can build a modular solution based on aspects:

```
var forbiddenElement = function(jp) {
  var elem = jp.target;
  return isDOMElement(elem) && isOutsideAppArea(elem);
};
var fa   = PCs.get("*").and(forbiddenElement);
var forbid = function(jp) {
  throw "forbidden access";
};
var securityAspect = AS.aspect(AS.AROUND, fa, forbid);
AS.deployOn(securityAspect, maliciousApplication);
```

AS.aspect is used to create the securityAspect. The fa pointcut of aspect matches all accesses to DOM elements that are outside the page area of the application, and the forbid around advice immediately throws an exception. In this example, securityAspect is deployed using deployOn. The semantics of *per-object* deployment are the same as in *e.g.* CaesarJ [Aracic et al., 2006] and AspectJ (*e.g.* perthis): the aspect "sees" all join points occurring *lexically* within the methods of maliciousApplication.

Although the securityAspect aspect prevents the maliciousApplication object from directly accessing elements outside of its page area, it does not forbid the application to *indirectly* access. This is because the aspect does not see the join points that are not lexically within any method of maliciousApplication. For instance, it can use an external function to change the home link:

```
var maliciousApplication = {
  //... as above
  action : function() {
    setHomeLink(this.fakeURL); //indirect modification
} };
```

While the above pattern could be detected by using a control flow check, the malicious application can also schedule setHomeLink(..) for later execution, thereby completely invalidating any kind of control flow reasoning:

[1] The actual implementation in Facebook rewrites the application code to replace all references to objects in the DOM by references to objects that check whether an application is trying to access an element outside its area. We use the isOutsideAppArea abstraction in this example for the sake of simplicity.

```
//modification scheduling in 100 ms
setTimeout(function(){ setHomeLink(this.fakeURL);}, 100);
```

In both cases discussed above, per-object aspect deployment is unable to properly prevent the malicious application to perform its evil deed. A solution to this problem is to use a more expressive scoping strategy for the security aspect [Tanter, 2008b]. In particular, we want to deploy the aspect in the maliciousApplication with a *pervasive* scoping strategy [Tanter, 2009]:

```
AS.deployOn(securityAspect, maliciousApplication, pervasive);
```

In general, a scoping strategy specifies how an aspect propagates along the call stack as well as in newly-created procedural values (objects and functions in JavaScript)[1] [Tanter, 2008b, Tanter, 2009]. The pervasive strategy simply states that the aspect *always* propagates in both function applications and newly-created procedural values.

In our example, this ensures that the security aspect is propagated unconditionally on the stack and therefore sees all accesses to elements in the control flow of any method of maliciousApplication. The security aspect is propagated to all new procedural values as well, so the anonymous function passed to setTimeout is also considered potentially malicious. Therefore, using the pervasive scoping strategy, the security aspect prevents maliciousApplication from directly *and* indirectly accessing elements outside of its page area.

4.2.5 Identifying New Kinds of Events

In Facebook, a user can interact with applications, events, messages, and many other elements. When a user interacts with one of these elements, a description is added to the user wall (*i.e.* a single page overview of recent user activity). Wall reporting is clearly a crosscutting concern; it can be modularized using an aspect:

```
var pc = ...; //user interactions with elements
var adv = function(jp) {
  var elem = ...; // extract elem from jp
  wall.add(elem.getDescription()); //desc. of the element
};
AS.after(pc, adv);
```

Specifying the appropriate pc pointcut is not straightforward. The standard approach is to define pc as the union of all calls (or executions) of the functions "corresponding to" interactions of the user with Facebook elements. Nevertheless, this does not always work. For instance, when the

[1] A scoping strategy also permits to refine the pointcuts of an aspect for a particular deployment, but we do not need this feature here.

user adds an already-added application, the change is discarded. This is reflected in the code of addApplication:

```
function addApplication(app) {
  if (!alreadyAdded(app)) {
    //1. check compatibility between the user and the app
    //2. synchronous Ajax call to register the app
    //3. add the app to the apps bar
} }
```

Defining a pointcut that matches when addApplication *effectively* adds an application is not straightforward. For these kinds of cases, AspectScript provides custom join points as in Ptolemy [Rajan and Leavens, 2008], whose underlying idea is to trade obliviousness for precision and abstraction. Using a custom join point, the addApplication function can signal precisely when it actually adds an application:

```
  if (!alreadyAdded(app)) {
    AS.event("addApp", {elem: app}, function() {
      //steps 1, 2, 3 as above
    }); }
```

A custom join point is generated by a call to AS.event, passing three parameters: an event type (addApp), an object containing contextual information provided to pointcuts and advices ({elem: app}), and a block of code (a thunk) specifying the base code associated to the join point. With this custom join point, the specification of the pc pointcut above is straightforward:

```
var pc = PCs.event("addApp").or(/*other interactions*/);
```

The PCs.event matches all custom join points with the specified name. The context information can be accessed as properties of the join point: in this example, jp.elem is the app object.

4.2.6 Summary

This informal presentation of AspectScript has illustrated several features of AspectScript through concrete applications. We have shown: how to define and create aspects with different advice kinds; how to compose pointcuts and define custom PCDs, taking advantage of higher-order programming patterns; how to expose context information; how join points can be used as first-class values, stored in data structures and proceeded multiple times; how to deploy aspect globally, per-object/function, and to control the scope of deployed aspect with scoping strategies; and how to use custom join points to identify new points of interest. The following section undertakes a more systematic exposition of the main features of AspectScript.

join point	points in the program execution at which...
new	a function or object is created.
init	a function or object is initialized.
call	a function is called.
exec	a function is applied.
p-read	an object property is read.
p-write	an object property is written.
event	AS.event(..) is called.

Figure 4.1: Dynamic join points of AspectScript.

4.3 A Tour of AspectScript

We now describe AspectScript in more details, reviewing its hybrid join point model (Section 4.3.1), followed by the aspect model (Section 4.3.2), and the deployment model (Section 4.3.3). Finally, Section 4.3.4 describes how aspect reentrancy is controlled.

4.3.1 Hybrid Join Point Model

At its core, AspectScript adopts a join point model in the line of that of AspectJ [Kiczales et al., 2001], but tailored for the JavaScript language. As in any aspect-oriented language, join points are generated implicitly upon certain evaluation steps of the program. Pointcuts can then quantify over these join points. While quantification is a crucial feature of aspect languages, the obliviousness brought by implicitly-generated join points is more controversial. In particular, it suffers from a number of problems related to the difficulty of reconciling the (low) level of abstraction of standard join points with the need for quantifying over application-specific events. To this end, AspectScript also includes a mechanism for explicitly triggering custom join points, in the style of Ptolemy [Rajan and Leavens, 2008]. Combining both implicit and explicit join point generation is a distinguishing feature of AspectScript, which is shared by other proposals [Inostroza et al., 2011, Núñez et al., 2009].

Standard join points. Figure 4.1 presents the standard join points supported in AspectScript. Save the last kind, which corresponds to custom events, all these join points are generated implicitly when a related expression is about to be evaluated. The JavaScript object model is peculiar in several respects. In particular, any function is considered a method of an object:

top-level functions and anonymous functions are considered methods of the global root object of JavaScript. In addition, executing this.foo() in the context of an object o, where foo is a method of o, is different from invoking simply foo(). In the latter case, foo executes in the context of the global object, not of o. Also, a function is an object per-se, which can have properties on its own. AspectScript considers these peculiarities when generating join points, setting the target and current object properties of the join point appropriately.

Custom join points. In addition to standard, implicitly-generated join points, AspectScript also supports explicitly-generated custom join points. This mechanism corresponds to typed events in Ptolemy [Rajan and Leavens, 2008]. It addresses limitation of both implicit invocation and aspect-oriented languages, by making it possible to treat any (block of) expression(s) execution as an event that can be quantified over by pointcuts. Events have a type that closely corresponds to their intended (application-specific) semantics, rather than being tied to the base language operational semantics. It is also possible to communicate an arbitrary set of context information to other pointcuts and advices without introducing unnecessary coupling to program details. In AspectScript, typed events are supported as custom join point with a type attribute. This attribute can be a string (JavaScript does not support symbols) or any object. Because AspectScript does not modify the syntax of JavaScript, the (block of) expression(s) that corresponds to the custom join point is defined in a thunk. In addition to the type and the thunk, a custom join point has an arbitrary number of properties, which can then be used in pointcuts and advices. The generation of a custom join point with AS.event was illustrated in Section 4.2.5. Aspects perceive custom join points like standard ones. Similarly, if no aspect apply, the original computation takes place, *i.e.* the specified thunk is applied.

4.3.2 Higher-Order Aspects

AspectScript is directly inspired by AspectScheme [Dutchyn et al., 2006], in which aspects, pointcuts, and advices are first-class values. Consequently, they can be created and manipulated at runtime. As illustrated in Section 4.2, an aspect in AspectScript is a pointcut-advice pair; pointcuts and advices are JavaScript functions.

Pointcut model. Following standard practice, it is standard to define a pointcut as a function that takes a join point as parameter and returns an environment if it matches, or false if it

does not [Masuhara et al., 2003]. The environment is used by pointcuts to pass information to the corresponding advice. A peculiarity of AspectScript in this respect is that pointcuts are also parameterized by an environment. This permits inner pointcuts of a given pointcut to communicate using the environment. For instance, in the pointcut (pc1 && pc2), the environment returned by pc1 (if it matches) is then passed to pc2. We have illustrated the use of this feature in Section 4.2.2[1].

Figure 4.2 presents some of the pointcut designators available in AspectScript. Because pointcuts are standard JavaScript functions, their definition does not rely on any additional syntactic constructs. Also, pointcut designators take full advantage of the potential of higher-order functions. No standard pointcut in AspectScript modifies the environment. Pointcuts exposing typical contextual information like this, target, and args in AspectJ would be redundant here because these values are available as join point properties (*e.g.* jp.target, jp.args), as illustrated in Section 4.2.

It is important to notice that AspectScript pointcuts do not rely on meta-data like types or annotations in order to match join points. The reason is that JavaScript does not support types nor annotations. AspectScript pointcuts also avoid using variable names to discriminate first-class values; rather they rely on value identity. This can be seen in the definitions of the call and exec pointcut designators in Figure 4.2, where the reference equality operator (===) is used to compare functions. Identity-based comparison is important in a language where functions are first-class values, and hence can be bound to many names, or none at all. Name-based selection in this context can result in many false negatives (*e.g.* a function execution not matched because it is applied through an alias) as well as false positives (*e.g.* a variable name that is used to refer to different functions at different moments in time). This concern has been identified in [Lerner et al., 2010], another aspect language for JavaScript.

Advice model. Advices in AspectScript are functions parameterized by a join point and an (optional) environment. The join point object passed as parameter has a proceed method, which permits the execution of the original computation at the join point. In the case of a custom join point, proceed evaluates the associated thunk. The environment corresponds to the environment returned by the associated pointcut. Like pointcuts, advices can access the bindings in the environment (*e.g.* in the example of Section 4.2.2, the advice can access both t0 and t1).

[1]The examples of Section 4.2 also make use of the fact that pointcuts can return true instead of the empty environment, and that all function parameters in JavaScript are optional.

```
function call(fun) { //call
  return function (jp,env) {
    return (jp.isCall() && jp.fun == fun)? env : false;
} }

function exec(fun) { //execution
  return function (jp,env) {
    return (jp.isExec() && jp.fun == fun)? env : false;
} }

function set(target,name) { //property write
  return function(jp,env) {
    return (jp.isPropWrite() &&
      jp.target == target && jp.name == name)? env : false;
} }

function cflow(pc) { //control flow
  return function cflow(jp,env) {
    if (jp == null) {
      return false;
    }
    return pc(jp,env)? env : cflow(jp.parent,env);
} }

function not(pc) { //negation
  return function(jp,env) {
    //'not' does not return the (possibly) modified env.
    return (pc(jp,env) != false)? true : false;
} }
```

Figure 4.2: Some pointcuts available in AspectScript.

AspectScript supports a basic scheme for the composition of aspects: when several aspects apply over the same joint point, the applications of their advices are nested like in AspectJ. Like in AspectScheme, the precedence for the application of nested advices is determined by the order in which the aspects are deployed: the last-deployed aspect goes first. We come back on aspect weaving in Section 4.4.2, when describing the implementation of AspectScript.

4.3.3 Deployment and Scoping Strategies

Dynamic deployment of aspects has been shown to enhance reuse and to better support software variability in general [Mezini and Ostermann, 2004, Rajan and Sullivan, 2003]. A number of aspect languages and frameworks hence support dynamic aspect deployment, under different flavors and scoping semantics. In fact, a language with higher-order aspects but without dynamic deployment makes little sense [Dutchyn et al., 2006]: if aspects can be crafted at

Deployment expression	Scope
deploy(asp) / undeploy(asp)	global
deploy(asp, fun [,ss])	on block: dynamic by default
deployOn(asp, val [,ss])	on object: lexical by default

Figure 4.3: Aspect deployment in AspectScript. Scoping strategies can be optionally specified in the last two alternatives.

runtime (*i.e.* by a higher-order function), it should definitely be possible to *deploy* them at runtime. Furthermore, controlling the scope of aspects is crucial, not only for software variability. When analyzing the potential problems that can arise when composing modules containing woven aspects, McEachen and Alexander make clear that developers need more control over scoping of aspects [McEachen and Alexander, 2005]. We believe this is all the more important in a dynamically-typed setting. AspectScript supports dynamic aspect deployment, further refined with scoping strategies [Tanter, 2008b].

Dynamic Deployment. As in CaesarJ [Aracic et al., 2006] and AspectScheme [Dutchyn et al., 2006], AspectScript supports dynamic deployment of aspects. Deployment options are presented in Figure 4.3. The first option is global scope: deploy(a) deploys aspect a such that it sees all join points in the execution of the program, until it is explicitly undeployed with undeploy(a). Internally, deploy (resp. undeploy) just adds (resp. removes) an aspect to the global aspect environment, globalAspects.

In order to deploy an aspect with dynamic scope (*e.g.* fluid-around in AspectScheme), deploy can take a thunk (no-arg function) as an extra argument. In this case, the aspect sees all join points in the dynamic extent of the evaluation of the thunk. Internally, this variant of deploy temporarily adds the aspect to the global environment, executes the thunk, and then removes the aspect:

```
function deploy(aspect, fun){
  globalAspects.add(aspect);
  var r = fun();
  globalAspects.remove(aspect);
  return r;
}
```

This mutation-based implementation is necessary because JavaScript does not support dynamic binding.

The third deployment option is per-value deployment, which follows the semantics of per-instance deployment in *e.g.* CaesarJ and AspectJ (per-this). The scope of an aspect deployed using deployOn is lexical, therefore the aspect only sees join points occurring lexically within method bodies of the objects it is deployed on. If deployed on a function, the aspect sees all join points in the body of the function, including inner functions.

Scoping Strategies. Both deployment on a block (deploy) and deployment on values (deployOn) can be refined using a scoping strategies, specified as a last parameter. Scoping strategies [Tanter, 2008b, Tanter, 2009] permit fine-grained control over the *scope* of a dynamically deployed aspect. A scoping strategy is a triple of functions [c,d,f]. c (resp. d) is propagation function specifying whether or not an aspect propagates along the call stack (resp. in newly-created functions or objects). f is an activation function making it possible to filter out certain join points. In other words, c permits to stop the unconditional propagation of dynamic scope at certain points, d enables aspects to be captured in certain procedural values as they are created, and f specifies a deployment-local refinement of the aspect pointcut.

All three functions are boolean-returning functions that take a join point as parameter[1]. In the case of c, the join point is the call the aspect can propagate through; in the case of d, the join point is the creation of the new object or function; and in the case of f, the join point is the one subject to filtering. As an example, the pervasive scoping strategy [Tanter, 2009] used in the example of Section 4.2.4 is defined as follows:

```
var pervasive = [true,true,true];
```

This strategy specifies that the aspect propagates unconditionally on the call stack (c always evaluates to true). Similarly, the aspect propagates to all new procedural values (d always evaluates true). Finally, the filter function f always evaluates to true as well, therefore no filtering is performed and the aspect sees all join points. Many examples of scoping strategies have been formulated elsewhere, for both local aspects [Tanter, 2008b] and distributed aspects [Tanter et al., 2009], as well as variable bindings [Tanter, 2009].

[1] In AspectScript, c, d and f can be specified directly as values; this is syntactic sugar for the corresponding constant function.

4.3.4 Control of Aspect Reentrancy

Thus far we have ignored a fundamental issue with the proposed design of AspectScript: if a function application triggers a join point, and a pointcut is a JavaScript function, then applying a pointcut pc triggers a function application join point against which pc should itself be evaluated, leading to an infinite loop! And the same happens with an advice that applies a function whose application is matched by its associated pointcut. In fact, while this issue of *aspect reentrancy* is exacerbated in a higher-order aspect language like AspectScript, it is latent in any aspect language[1]. Some mechanism must be provided to avoid aspects potentially matching join points triggered by their *own* execution [Bodden et al., 2006, Tanter, 2008a].

Limitations of current solutions. Higher-order procedural aspect-oriented languages like AspectML and AspectScheme adopt different solutions to this problem, though both rely on a mechanism to deactivate weaving in some way. Indeed, AspectScheme uses a primitive operator app/prim to apply a function without generating a join point, and AspectML suggests a similar disable primitive that hides the whole computation of an expression. The difference is their scope: app/prim only hides a single function application join point, but does not hide the computation triggered by that application; conversely, disable has dynamic scope. Current AspectJ patterns to address these issues are similar to disable: adding a control-flow condition to pointcuts such that join points occurring in the dynamic extent of advice execution are ruled out [Bodden et al., 2006]. The AspectJ pattern however does not work in the case of reentrancy caused by if pointcuts [Tanter, 2008a].

As argued extensively elsewhere [Tanter, 2010], relying on control flow checks to avoid reentrancy is flawed for several reasons. Most importantly, since proceed is called as part of an advice, reasoning on control flow conflates advice computation and base computation, resulting in aspects not applying when then should. Also, disabling weaving for aspectual computation simply makes it impossible for aspects to advise aspects.

Solution in AspectScript. In order to properly address issues of reentrancy without entailing conflation or sacrificing visibility of aspect computation, AspectScript relies on the notion of

[1] http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html

execution levels [Tanter, 2010], a refinement of the proposal of stratified aspects [Bodden et al., 2006]. In a nutshell, the idea is to structure computation into levels, starting with base computation at level 0. Aspect computation is by default considered as ocurring at level 1, and is therefore invisible to other aspects. If needed, aspects can be deployed at higher levels of execution, thereby possibly observing other aspects, or an aspect can explicitly lower part of its computation (if so, a mechanism ensures that it does not see its *own* computation). A detailed description of the implementation of reentrancy control and execution levels is however outside the scope of this chapter. In-depth motivation and formal description of execution levels can be found in [Tanter, 2010].

The bottom line is that, for the programmer, AspectScript just works as if the issue of reentrancy did not exist, *precisely* because AspectScript handles execution levels behind the scene. We have not illustrated advanced scenarios with explicit level shifting, such as aspects of aspects (see [Tanter, 2010]); rather we have focused on illustrating the simplicity of the default, most common cases. All the pointcuts in Section 4.2 and Figure 4.2 are defined naturally, without having to resort to primitive operators like app/prim and disable. Similarly, advices can perform any computation and they never enter infinite loops caused by aspect reentrancy.

4.4 Implementation

In this section we detail the implementation of AspectScript. In particular, Section 4.4.1 details the code transformation phase required for weaving, and Section 4.4.2 describes the weaving process. Finally, we end with a preliminary analysis of AspectScript performance in Section 4.4.3.

Given the dynamic nature of JavaScript, weaving in AspectScript is mostly done at runtime. In order to be able to dynamically weave aspects without modifying a particular JavaScript engine, our AspectScript implementation first performs a code transformation phase in which some expressions are rewritten into invocations of *reifiers* (Section 4.4.1). These reifiers then make it possible to perform runtime aspect weaving (Section 4.4.2). Relying on code transformation means that AspectScript can run on any client browser, without requiring the installation of dedicated complements or add-ons. Currently, our AspectScript implementation has been tested with the Mozilla Firefox browser (versions 3.0.* and 3.5.*). The complete transformation phase is implemented in JavaScript, using an optimized version of the parser of the Narcissus project[1]. A JavaScript implementation is interesting in order to be able to do client-side parsing; this could

[1] http://mxr.mozilla.org/mozilla/source/js/narcissus/

be useful for example in the case of remote code loading or eval invocations. These applications are subject of further exploration.

4.4.1 Code Transformation

The code transformation phase rewrites JavaScript source code[1] by introducing invocations of *reifiers*. Reifiers are functions that generate the join points associated with the rewritten expression.

For the reader unfamiliar with JavaScript, Figure 4.4 presents a subset of its syntax, relevant to the transformation performed by AspectScript (we substitute fun for function to save space). A script is a list of statements, which can be either a function or variable declaration, a block or an expression. Expressions include object creation (either with a constructor or literally), definitions of arrays and anonymous functions, function invocation and variable/property access and assignment.

$$
\begin{array}{rcll}
\textit{Script} & z & ::= & \overline{s} \\
\textit{Statement} & s & ::= & \mathsf{fun}\ id(\overline{id})\ \{\overline{s}\} \mid \mathsf{var}\ id = e \mid \{\overline{s}\} \mid e; \\
\textit{Expression} & e & ::= & \mathsf{new}\ e_c(\overline{e}) \mid \{\overline{id : e}\} \mid [\overline{e}] \mid \mathsf{fun}(\overline{id})\{\overline{s}\} \\
 & & & \mid e_f(\overline{e}) \mid e_t.id(\overline{e}) \\
 & & & \mid id \mid e.id \mid id = e \mid e_t.id = e
\end{array}
$$

Figure 4.4: Subset of the JavaScript syntax.

The transformation is defined by the syntax-driven rewriting function $[\![\cdot]\!]$, presented in Figure 4.5. The first four rules deal with statements. Rule 1 recursively triggers the transformation of the expression being used to initialize the declared variable. A function declaration is rewritten into a var declaration, binding the function name to the (transformation of) the anonymous function definition (rule 2). Rule 3 groups functions declared in a block, and moves them to the beginning of that block. This reordering is necessary because of rule 2: in JavaScript a variable needs to be *explicitly assigned* to a value in order to be used, whereas a function declaration makes the function name available globally in its declaring block. Finally, rule 4 rewrites the expression part of the statement.

[1] AspectScript per se does not extend the *syntax* of JavaScript; as illustrated in previous sections, it is provided as a library.

Statements

$$[\![\text{var } id = e]\!] = \text{var } id = [\![e]\!] \quad (1)$$
$$[\![\text{fun } id_f(\overline{id})\ \{\overline{s}\}]\!] = \text{var } id_f = [\![\text{fun } (\overline{id})\ \{\overline{s}\}]\!] \quad (2)$$
$$[\![\{\overline{s}\}]\!] = \{[\![\overline{s}_{fun-decls}]\!][\![\overline{s}_{other-decls}]\!]\} \quad (3)$$
$$[\![e;]\!] = [\![e]\!]; \quad (4)$$

Object Creation

$$[\![\text{new } e_c(\overline{e})]\!] = \rho_{new}(\text{fun}(k,\overline{a})\{\text{return new } k(\overline{a})\}, [\![e_c]\!], [[\![\overline{e}]\!]]) \quad (5)$$
$$[\![[\overline{e}]]\!] = \rho_{new}(\text{fun}(k,\overline{a})\{\text{return } [\overline{a}]\}, \text{Array}, [[\![\overline{e}]\!]]) \quad (6)$$
$$[\![\text{fun}(\overline{id})\ \{\overline{s}\}]\!] = \rho_{wrap}(\text{fun}()\{\text{return fun}(\overline{id})[\![\{\overline{s}\}]\!]\}) \quad (7)$$
$$[\![\{\overline{id : e}\}]\!] = \rho_{lit}(\text{fun}()\{\text{this}.id = [\![e]\!]\}) \quad (8)$$

Function Invocation

$$[\![e_f(\overline{e})]\!] = \rho_{call}(\text{AS.globalObject}, [\![e_f]\!], [[\![\overline{e}]\!]]) \quad (9)$$
$$[\![e_t.id(\overline{e})]\!] = \rho_{call}([\![e_t]\!], [\![e_t.id]\!], [[\![\overline{e}]\!]]) \quad (10)$$

Property Access

$$[\![e.id]\!] = \rho_{read_p}([\![e]\!], \text{"}id\text{"}) \quad (13)$$
$$[\![e_t.id = e]\!] = \rho_{assign_p}([\![e_t]\!], \text{"}id\text{"}, [\![e]\!]) \quad (14)$$

Figure 4.5: Rewriting function.

The next rules rewrite expressions. The general scheme is the same: all the information required to generate a join point is gathered and passed as parameters to the appropriate reifier, which then triggers weaving (Section 4.4.2). Object creation (rule 5) is rewritten into an invocation of the ρ_{new} reifier, passing as argument a first-class anonymous function that encapsulates the actual instantiation (parametrized by the constructor and actual arguments). This function is then used at runtime for the proceed method of the constructed join point. The same approach is used for array creation (rule 6). The reifier also receives the constructor and the arguments. In order to support generation of function execution join points, functions are wrapped; thus, rule 7 uses a different reifier ρ_{wrap}. Finally, literal object creation (rule 8) uses yet another reifier, ρ_{lit}. This reifier receives as argument a first-class anonymous function that encapsulates the initialization of the properties specified in a given literal object creation.

The remaining rules are straightforward. Function invocations (rules 9 and 10) are transformed into invocations of the ρ_{call} reifier, passing as parameters the target of the call, the

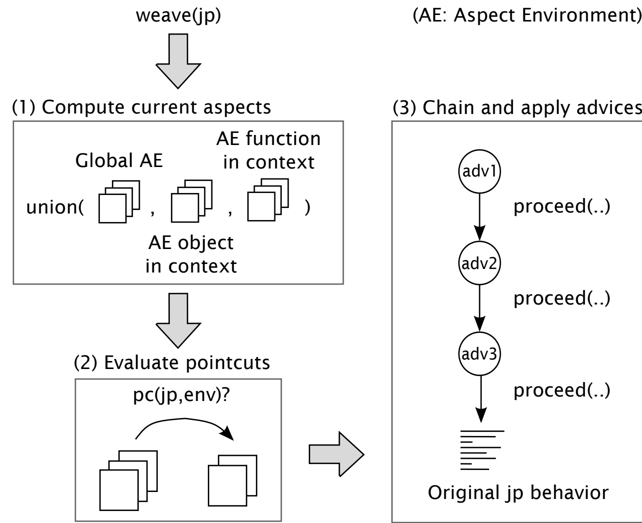

Figure 4.6: Weaving process.

function being invoked, and its arguments. Rule 9 uses the global object as a target because the JavaScript semantics specifies that when no target is specified when invoking a function, the global object must be used. Rules for property access (13 and 14) transform reads and writes of properties into invocations of the ρ_{read_p} and ρ_{write_p} reifiers, passing as parameter the name of the property being accessed, and in the case of a property write, the value being assigned. The owner of the property is passed as the first argument to both reifiers.

4.4.2 Runtime Weaving

At runtime, the calls to the reifiers inserted by rewriting are evaluated. Apart from creating the corresponding join point, the reifiers also trigger the weaving process by invoking the weave function of AspectScript. To illustrate reifiers, below is the (simplified) implementation of the ρ_{call} reifier:

```
function r_call(obj, fun, args){
  var jp = new CallJP(obj, fun, args, currentJoinPoint);
  return weave(jp);
}
```

The r_call function receives as arguments the target of the call (obj), the function being invoked (fun), and the arguments to that function (args). This is the JavaScript code executed to weave a call expression, corresponding to the transformation rules 9 and 10 in Figure 4.5.

The weave function weaves the join point it receives as argument. Figure 4.6 depicts the weaving process initiated by weave, and its simplified definition is as follows:

```
function weave(jp){
  // 1. compute current aspects
  var currentAspects =
    union(globalAspects, aspectsIn(ctxObj), aspectsIn(ctxFun));
  // 2. evaluate pointcuts
  var advices = match(jp, currentAspects);
  // 3. chain and apply advices
  return chainAndApply(advices);
}
```

The first step is to determine the set of aspects that may potentially apply. Recall from Section 4.3.3 that AspectScript supports different deployment mechanisms: global, per-function, and per-object[1]. In consequence, at a given join point, the list of aspects that may apply is the *union* of the aspects that are deployed in several aspect environments (step 1). First, the *global* aspect environment contains all aspects deployed using deploy (either deployed globally or with dynamic scope). In addition, aspects may have been deployed with deployOn, therefore each object and function has its own aspect environment. At a given join point, aspects in the currently-executing function (ctxFun) and the currently-execution object (ctxObj) have to be considered (accessed with aspectsIn in the code above).

Once the list of current aspects has been determined, pointcuts of all these aspects are evaluated against the current join point (step 2). The advices of aspects that matched the current join point are then chained together, each one nesting the next one (step 3). Executing proceed in one advice triggers the following advice; in the last advice in the chain, proceed runs the original base computation. Note that following AspectScheme [Dutchyn et al., 2006], before and after advices are only syntactic sugar for around advices. Aspects are deployed in the reverse order of their deployment, like in AspectScheme.

4.4.3 Runtime Performance

The primary design goal of AspectScript is expressiveness. When conceiving the language, we have not sacrificed any potentially valuable feature on the basis of its expected cost. Still, we are interested in making AspectScript a practical solution for aspect-oriented programming in JavaScript; it therefore makes sense to evaluate its overhead. In order to do so, we selected a subset of the jQuery [jQuery, 2011] test suite (+1300 tests, around 90% of the total number

[1] Scoping strategies are an extension to AspectScript (Section 7.3.1).

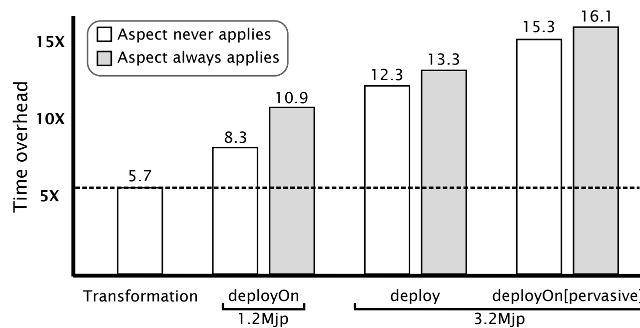

Figure 4.7: Performance overhead of AspectScript for CPU-intensive tests in the JQuery test suite.

of tests) that includes only CPU-intensive tests, as this constitutes a worst-case scenario for AspectScript. For this experiment, we used jQuery 1.3.2 (118KB of source code) on an Intel Core 2 Duo, 2.66 GHz PC with 2GB of RAM running Ubuntu 9.04 (kernel 2.6.28) and Firefox 3.5.2. Running the selected tests takes approximately 20s in this setting, without AspectScript.

We then transformed the jQuery library as described in Section 4.4.1 in order to support aspects. The transformed code ended up weighing 326KB (a 2.8x factor). Just running the test suite with the transformed library, without any aspect weaving at all, is 5.7 times slower. This overhead includes the generation of approximately 3.4 million join points (Mjp). Considering this baseline, Figure 4.7 describes the overhead introduced by an aspect following different deployment scenarios: *(a)* a globally-deployed aspect, *(b)* an aspect deployed with the pervasive scoping strategy we used in Section 4.2.4, and *(c)* an aspect deployed on an object (the $ object, entry point of the jQuery library). In both cases *(a)* and *(b)*, the deployed aspect sees all 3.4Mjp; in case *(c)*, due to lexical scoping, the aspect only sees 1.2Mjp. The advice of the aspect only calls proceed on the join point. For each case, we measured the overhead with both a pointcut that never matches and a pointcut that always matches, thus giving a lower and an upper bound for the overhead of the presence of the aspect. We did experiments where the pointcut does a function comparison (using reference equality) instead of just returning a boolean, without any noticeable difference.

The overhead of a global aspect is between 12.3x (never match) and 13.3x (always match). Activating scoping strategies and deploying a pervasive aspect implies a relative overhead of 20% only (15.3x-16.1x). Using a per-object aspect (which results in the aspect seeing only 1.2Mjp) reduces the overhead down to 8.3x-10.9x. The results are overall encouraging, especially considering the low amount of time we spent working on optimizing the implementation so far; our

focus has rather been to get the semantics right and working. We believe there are many venues for optimization to be explored, both in the runtime itself and by adding the possibility for the programmer to statically declare certain aspects and restrictions on the general dynamicity of AspectScript. Finally, raw overhead in a CPU-intensive scenario does not really reflect the fact that JavaScript is more widely used for interactive applications, that may even include remote communication. For instance, we have tested AspectScript (global aspect always matching) on a JavaScript Tetris game, without any noticeable difference[1].

4.5 Aspect Languages for the Web & Higher-Order Aspect Languages

Modularization of crosscutting concerns has long been considered in Web technologies [Stamey et al., 2005]. An example of this is the separation of an HTML document in different sources, such as CSS files for presentation style and JavaScript files for functionality. However, within these sources, crosscutting concerns are still present. For this reason, diverse tools to modularize these concerns are available in the Web. For instance, the jQuery library [jQuery, 2011] allows programmers to separate crosscutting concerns in the DOM of a Web page, *e.g.* for adding borders to all tables. We now review AOP frameworks for JavaScript, as well as AOP proposals for other higher-order procedural languages.

AOP for JavaScript. A large number of lightweight AOP frameworks for JavaScript have been made available on the Web by programmers, that rely on function wrappers. In the simplest case, there is no quantification at all, and one explicitly has to give both a function and the advice function that should be added to it with wrapping [Prototype, 2011]. Some frameworks make it possible to wrap methods of an object by specifying the names of the methods to wrap [AspectJS, 2011a, AspectJS, 2011b, Humax, 2011], and others support regular expressions for describing method names to wrap [Ajaxpect, 2011, Cerny, 2011].

AOJS [Washizaki et al., 2009] is a more mature AOP framework that takes quantification more seriously, and relies on code transformation. However, it does not embrace the features of JavaScript as AspectScript does. Aspects (with before and after advice only) are specified in a separate XML file; therefore, aspects cannot enjoy the full power of higher-order programming. In addition, function identification is name-based, rather than identity-based. As discussed in

[1] Both versions of the Tetris game can be tested online on the AspectScript website (http://pleiad.cl/aspectscript)

Section 4.3.2, this leads to fragile pointcuts, that may (not) match when expected, and excludes the execution of anonymous functions. The join point model of AspectScript is also considerably richer, and extensible. AOJS does not support dynamic deployment of aspects. While this is beneficial in terms of performance, it is limitating in terms of expressiveness. We plan to improve performance of AspectScript in the future by supporting static specifications to restrict full dynamicity to a certain extent. Finally, aspect reentrancy in AOJS is avoided by simply deactivating weaving during pointcut and advice evaluation, an insufficient solution (Section 4.3.4).

AOP for higher-order procedural languages. AspectScript draws significantly on previous work on AOP extensions of higher-order procedural languages like Scheme, Standard ML and Caml[1].

AspectScheme [Dutchyn et al., 2006] is an aspect-oriented extension of Scheme, which is, like JavaScript, dynamically typed. In order to support the full power of higher-order programming, pointcuts and advices in AspectScheme are standard functions, aspects are dynamically deployed (either with dynamic or lexical scope), and function identification is identity-based rather than name-based. AspectScript imitates AspectScheme in all these dimensions. This said, AspectScript supports a richer join point model, including custom join points. Context exposure in AspectScript is more powerful, allowing pointcuts in a composition to use bindings exposed by prior pointcuts. Finally, AspectScript supports more expressive aspect deployment (deployOn) and scoping (scoping strategies).

AspectML [Dantas et al., 2008] is an AO extension of Standard ML [Milner et al., 1990]. Like AspectScript, pointcuts are first-class values, but advices are not. Aspects are deployed with lexical scope only. AspectML does not use function identity to match pointcuts; instead, pointcuts use function names and argument types to match functions that are currently in lexical scope. Anonymous functions cannot be advised (although an as-yet-unsupported any keyword is discussed). As discussed before, relying on function names in a language where functions are first-class values easily leads to both false positives and false negatives.

Aspectual Caml [Masuhara et al., 2005] extends Caml [Leroy et al., 2011] with aspect-oriented constructs. Like AspectML, Aspectual Caml integrates well with the type system of the host language. In addition, Aspectual Caml takes into account the fact that Caml supports

[1]We do not repeat here the limitations of the mechanisms provided by AspectScheme and AspectML to avoid reentrancy (Sect. 4.3.4).

object-oriented programming. In essence, the aspect-oriented features of Aspectual Caml are very similar to AspectML, except for the fact that Aspectual Caml supports pattern matching over method names (in any scope), as well as advising anonymous functions. However pointcuts are not first-class values.

4.6 Summary

This chapter presented AspectScript, an aspect-oriented extension for JavaScript. AspectScript fully embraces the characteristic features of JavaScript, by supporting higher-order aspects, a full-fledged join point model, customizable quantified events, and dynamic aspect deployment with expressive scoping. Aspect reentrancy is avoided without causing any burden on programmers. This combination of features is unique in the current design space of aspect languages.

AspectScript is the base of our work because the concrete and practical implementation of OTM is an extension of AspectScript. We chose AspectScript because pointcuts and advices can be expressed using the full power of the base language. For example, the body of an AspectScript pointcut can implement the full matching process of a stateful aspect (Chapter 3, Section 3.3.2). In the following chapters, we use this property to implement ETM (Chapter 5), an expressive trace-based mechanism to define stateful aspects, and OTM (chapter 6), an open implementation of ETM.

Chapter 5

ETM: Expressive Stateful Aspects[1]

Real-world applications bring together common issues related to security flaws, application errors, and crosscutting concerns. Different trace-based mechanisms with distinctive features have been proposed to solve these particular issues. However, these mechanisms are not sufficiently expressive to match complex program execution traces. For example, patterns of tracematches are expressed by a limited domain-specific language: regular expressions. The lack of expressiveness does not allow developers to define, for example, a security stateful aspect with a pattern that adapts its specification according to the security context (*e.g.* low and high).

In this chapter, we describe a model of an Expressive Trace-based Mechanism (ETM), which allows developers to flexibly express patterns. In ETM, patterns (and advices) are first-class values because they are expressed as functions, reaping all the benefits of higher-order patterns. In addition, these patterns can expressively gather and compare any kind of value during their matching.

This chapter presents a complete and practical implementation of ETM for JavaScript as an AspectScript (Chapter 4) extension. This implementation allows us to show its usefulness through a better modularity support for building Web applications. In addition, we use Typed Racket [Tobin-Hochstadt and Felleisen, 2008] to provide a typed functional description of the API of ETM.

[1]The content of this chapter is a summary and adaptation of our FOAL 2010 paper "Towards An Open Trace-Based Mechanism" [Leger and Tanter, 2010b].

5.1 Introduction

Trace-based mechanisms support the definition of stateful aspects (Chapter 3) that observe and react to certain program executions; they have numerous applications in domains like security, distributed computing, error detections, and modular definition of crosscutting concerns.

Stateful aspects are composed of a pattern (instead of a pointcut in aspects [Kiczales et al., 1996]) and an advice. The pattern is the specification of the program execution trace that should be matched. During the matching, a pattern can gather bindings and utilize them to match more accurately. The advice is executed when its associated pattern is matched. This advice only affects the last join point matched and can use the bindings gathered by the match of the pattern. Most trace-based mechanisms have different limitations that we summarize in the following categories:

Reusability. A stateful aspect is composed of a pattern and an advice. In most trace-based mechanisms, patterns are not first-class values and reusable, *i.e.* the pattern of a stateful aspect cannot be reused by another stateful aspect. The reusability of patterns can be useful, for example, to express patterns of different levels of abstraction, where a developer that expresses a high-level pattern does not need to know the details of low-level patterns that compose it.

Binding expressiveness. During the matching, a pattern can gather bindings that are exposed at intermediate join points. In most trace-based mechanisms, patterns cannot gather bindings out of the attributes of the current join point or create specific data structures like lists of bindings. The binding expressiveness can be useful, for example, to gather a variable-sized list of values (or the maximum value) in a pattern like $\texttt{a}^\star$.

Pattern expressiveness. A pattern can be expressed in different ways. Patterns of most trace-based mechanisms are expressed by a limited domain-specific language, *e.g.* regular expressions or SQL. These languages do not allow developers to express *advanced* patterns like patterns that alter what they should match at runtime.

In this chapter, we present a model of an Expressive Trace-based Mechanism, named ETM. Patterns in ETM are expressed by functions, reaping the benefits of high-order functions. Hence, patterns can be reused and composed to express advanced ones. In addition, the expressiveness

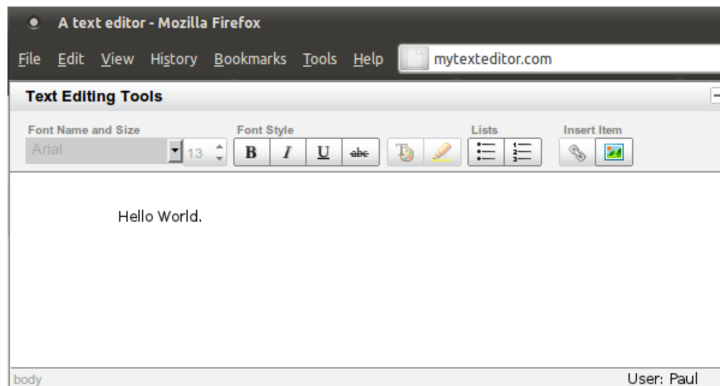

Figure 5.1: A Web text editor.

to specify a pattern allows developers to use an explicit environment to flexibly gather and use bindings during the matching of a trace.

The rest of this chapter is organized as follows. Section 5.2 briefly introduces trace-based mechanisms. Section 5.3 shows limitations of existing trace-based mechanisms using simple examples. Section 5.4 introduces ETM with its type description (Appendix B for more details) and current implementation. Section 5.5 shows how ETM addresses the limitations we described previously. Section 5.6 summaries this chapter.

Note. ETM, along with the examples presented in this chapter, is available online at http://pleiad.cl/otm. The ETM version for JavaScript currently supports the Mozilla Firefox browser without the need of an extension.

5.2 Trace-based Mechanisms

This section illustrates trace-based mechanisms (Section 3.3.4) in action using a Web text editor (see Figure 5.1) similar to Google Docs[1]. Using this text editor, a user can create, edit, and save a document on the Web. Suppose that developers of this application need to add the *autorevision* feature: a revision of the document is automatically generated when it is edited three times. This feature is clearly crosscutting because it is necessary to modify every method that edits the document (*e.g.* insert, delete). The pattern of a stateful aspect that allows developers to modularly add this feature is:

[1] http://docs.google.com.

$$\text{threeEditions} = \text{edit}^{e1,t1} \rightarrow \text{edit}^{e2,t2} \rightarrow \text{edit}^{e3,t3}$$

The threeEditions pattern is needed by the stateful aspect to support the autorevision feature. This pattern represents the match of three editions in the document. The term edit represents any call to methods that edit the document, this is concretely a pointcut. The superscript terms represent the bindings that are gathered during the matching of the pattern. In the threeEditions pattern, e1 represents the first edition and t1 represents the timestamp of the first edition. Whenever threeEditions matches, the three editions with their timestamps are gathered. Whenever the pattern is matched, the advice of the stateful aspect generates a new revision, which contains each edition with its timestamp.

To match three editions in a document, an simple aspect could work. However, this aspect should maintain a counter to keep track of the number of editions. For every edition, the aspect increases the counter by one; and if this counter reaches three, the advice is triggered. The state of the matching process (*i.e.* the counter) should be explicit in this aspect. In the autorevision feature, this leads to only minor complications, but as we shall see below, often the burden of such state maintenance is much greater.

Implementing the autorevision feature in tracematches. Allan *et al.* propose tracematches [Allan et al., 2005], a widely-used trace-based mechanism for Java. Tracematches are implemented as an AspectJ [Kiczales et al., 2001] extension. The piece of code below shows the implementation of the autorevision feature using tracematches. A pattern is defined by a regular expression, whose alphabet is composed of symbols that are AspectJ pointcuts[1]. The bindings exposed by pointcuts are gathered in the pattern. The advice is a block of code that can use these bindings.

```
tracematch(Text e1,Text e2,Text e3) {
   sym edit1 before: edit() && args(e1);
   sym edit2 before: edit() && args(e2);
   sym edit3 before: edit() && args(e3);

   edit1 edit2 edit3  //regular expression
   {
      revisionControl.add(new Revision(e1,e2,e3));
 } }
```

[1]To be precise, a symbol is composed of an AspectJ pointcut and a modifier of the join point: *before*, *around*, and *after*. See Chapter 6 for more details.

In this tracematch, there are three symbols that are used to match the three editions. These symbols are composed of the edit pointcut, which matches calls to any method that edits the document (*e.g.* insert and erase), and a pointcut that exposes the edited text. The advice adds a new revision to a revision control system. Notice the timestamp of every edition is not gathered and used by the advice. This is because tracematches cannot gather values out of the context of the current join point; in this case, a call join point. If this tracematch could gather the timestamp of editions, developers would be able, for example, to discard matches of the pattern of three editions longer than a period of time. In the following section, we discuss this and other limitations of existing trace-based mechanisms.

Note. Tracematches and other trace-based mechanisms support that a pattern that can be matched multiple times in a program execution trace. In this chapter, we assume, for simplicity, that a pattern will only matched once at a time[1].

5.3 Limitations of Existing Trace-based Mechanisms

In this section, we categorize and illustrate the limitations of existing trace-based mechanisms using the example of the Web text editor.

5.3.1 Reusability

A pattern and an advice compose a stateful aspect. In most trace-based mechanisms, patterns are not first-class values and cannot be reused. For example, consider that developers now need to add the *autoreport* feature: the Web application sends a report to all editors of a document when four autorevisions are generated. To implement this feature, developers need to reuse the threeEditions pattern (Section 5.2):

$$\text{fourRevisions} = \text{threeEditions} \rightarrow \text{threeEditions} \rightarrow \text{threeEditions} \rightarrow \text{threeEditions}$$

The fourRevisions pattern is composed of four threeEditions patterns. Apart from reusing threeEditions to match, fourRevisions can use the bindings gathered by threeEditions to add information to the report that the advice generates.

[1] The discussion of this and other features like simultaneous advice executions are expressed in the next chapter.

5.3.2 Binding Expressiveness

A pattern can gather bindings during its matching. In most trace-based mechanisms, the expressiveness to gather bindings is limited. For example, as mentioned in the previous section, tracematches cannot gather bindings out of the context of the current join point, and therefore they cannot do *e.g.* temporal matching. As another example, consider that developers now need to only generate one revision for every session of a user, *i.e.* a revision with all editions is generated when the user logs out of the Web text editor. To implement this feature, named *session revision*, developers need to store all editions of the document until the user logs out of the application:

$$\mathsf{allEditionsBySession} = (\mathsf{edit}^{e,t})^{*} \rightarrow \mathsf{logout}$$

The allEditionsBySession pattern matches all editions in the document until the user logs out of the editor. During its matching, the pattern gathers a list of editions and another list of the timestamps of these editions. These lists are created from every occurrence of the join point matched by edit.

5.3.3 Pattern Expressiveness

In most trace-based mechanisms, developers express patterns using a limited domain-specific language, *e.g.* regular expressions or SQL. Therefore, developers cannot express advanced patterns. For example, consider that developers now need to add an *adaptive* autorevision feature: if the user is *writing* the document, a revision is generated every five minutes; instead if the user is (selectively) *editing* the document, a revision is generated every three editions. To implement this feature, developers need to express a pattern that changes what it should match at runtime:

$$\mathsf{adaptiveAutorevision} = (\mathsf{edit}^{e,t})^{*}{}_{\Delta \mathrm{t}<5*60} \quad \overleftrightarrow{\Delta e>3\ \&\ \Delta t<30} \quad \mathsf{threeEditions}$$

The adaptiveAutorevision pattern is composed of two patterns that interchange with each other according to the detected context: “writing” or “editing”. The context is detected through the number of editions by a period of time. Concretely, if there are three editions within thirty seconds, the context is “writing”; otherwise the context is “editing”.

5.4 ETM

This section presents the model of an Expressive Trace-based Mechanism, ETM. We explain ETM using the JavaScript language, a dynamic prototype-based language with higher-order functions,

for which we have developed a concrete and complete implementation of ETM. In addition, we show a typed functional description of our proposal in Typed Racket [Tobin-Hochstadt and Felleisen, 2008]. The implementation for JavaScript allows us to show its usefulness through better modularization support in developing Web applications. We use the autorevision feature of the Web text editor (Section 5.2) to introduce ETM.

Like existing trace-based mechanisms, ETM supports the definition of stateful aspects, which are composed of a pattern and an advice. We now describe the models of these three components.

5.4.1 Pattern Language

The pattern model of ETM is the natural extension of the pointcut model of an aspect language. For clarity, we explain this affirmation in two parts. In the first part, pointcuts and patterns do not gather bindings during the matching; and the second part, pointcuts and patterns can gather bindings.

5.4.1.1 Without Gathering Bindings

Whereas a pointcut is a function Pc: JoinPoint $\rightarrow$ Boolean, a pattern is a function with the type Pattern: JoinPoint $\rightarrow$ Boolean $\bigcup$ Pattern. A pointcut and a pattern take a join point and return a boolean value to determine whether there is a match or not with this join point. In addition, a pattern can return another pattern, which specifies the next step in the matching process. For example, the implementation of a pattern that represents the match of a call to a method that edits the document is:

```
var edit = function(jp) {
  return jp.isCall() && (jp.fun == insert || jp.fun == erase);
};
```

The function above returns true if it matches the call to methods insert or erase; this function returns false otherwise. In this example, the pattern is really a pointcut. As another example, consider the implementation of a pattern combinator that represents the match of a (sub)pattern followed by another (sub)pattern:

```
var seq = function(left, right) {
  return function(jp) {
    var result = left(jp);
    if (isFunction(result)) { //Is a (sub)pattern of left?
      return seq(result, right);
    }
```

seq(edit,seq(edit,edit)) —edition→ seq(edit,edit) —edition→ edit —edition→ MATCH!

Figure 5.2: Matching three editions.

```
    return result? right: false;
} };
```

The seq pattern designator (*i.e.* a function that returns a pattern) returns a pattern that matches the left pattern followed by the right pattern. This pattern returns a new pattern if left advances in its matching. Instead, the pattern returns right if left matches. In this example, the pattern returns a new function to specify the continuation. The threeEditions pattern can be implemented through compositions of seq:

```
var threeEditions = seq(edit,seq(edit,edit));
```

Figure 5.2 shows how the threeEditions pattern varies throughout the matching of a program execution trace. This pattern changes every time it matches a call to methods insert or erase. In the beginning, the pattern begins with the pattern expressed by the programmer. For the first edition, the pattern changes to a pattern that only matches two editions. The second edition, the pattern is only the edit pattern. Finally, once the third edition occurs, the whole pattern matches.

5.4.1.2 Gathering Bindings

The previous description of the pattern model is incomplete because a pattern cannot gather bindings during the matching process. Pointcuts and patterns should be able to gather bindings during this process. The standard vision of the pointcut-advice model [Masuhara et al., 2003] establishes a pointcut as a function Pc: JoinPoint $\rightarrow$ Env $\bigcup$ False. This definition means that a pointcut returns an environment of bindings (instead of true) if the pointcut matches the current join point. Our model is an extension of the standard pointcut-advice model because a pattern is a function:

$$\text{Pattern: JoinPoint} \times \text{Env} \rightarrow \text{PcMatch} \bigcup \text{Advance}$$

where

$$\text{PcMatch: Env} \bigcup \text{False} \qquad \text{and} \qquad \text{Advance: Pattern} \times \text{Env}$$

A pattern now also takes an environment as a parameter. This environment contains the values previously gathered by the pattern. Like pointcuts, a pattern returns an environment

(instead of true) when it matches. In addition, a pattern can return a pair (instead of a pattern only) composed of a pattern and an environment if the pattern advances in its matching. To exemplify the extension of the pattern model, we redefine the patterns edit and seq:

```
var edit = function(jp,env) {
  if (jp.isCall() && (jp.fun == insert || jp.fun == erase)) {
    return env.bind("e",jp.args[0]). // 1st argument to the function
                bind("t",getTime());
  }
  return false;
};
```

The edit pattern now takes a join point and an environment. The environment is an object with functional behavior. An environment has the bind method to bind an identifier to a value (or to gather a binding). The edit pattern now returns an environment, which contains the edition and the timestamp of this edition, when it matches.

```
var seq = function(left,right) {
  return function(jp,env) {
    var result = left(jp,env);
    if (isPair(result)) { //Did left return a pair [pattern X env]?
      return [seq(result[0],right),result[1]];
    }
    return isEnv(result)? [right,result]:false;
} };
```

The pattern returned by the evaluation of seq returns a pair composed of a pattern and an environment when the pattern advances in its matching. The returned environment is used for the next evaluation of the pattern.

Note that the definition of a pointcut of AspectScript (Chapter 4) is also a pattern of ETM because the pointcut returns an environment or false, which is a subset of what an ETM pattern returns. Therefore, the implementation of ETM can reuse all AspectScript pointcuts.

5.4.1.3 ETM Pattern Language is Turing Complete

During this dissertation, we have mentioned that the proposed pattern language is Turing complete. In this section, we use Lambda Calculus [Barendregt, 1984] to demonstrate the completeness of the ETM pattern language. The demonstration goal is to simulate Lambda Calculus, a Turing complete language, with our proposed language.

In 1936, Alonzo Church proposed the untyped Lambda Calculus (*a.k.a.* Lambda Calculus now) language as a formal system in computation. In a few words, the Church language only permits to define functions with one parameter, and the composition and recursiveness of functions achieve the completeness property of language. In our pattern language, a pattern is defined through a composition of functions with the following signature:

$$\text{Pattern: JoinPoint} \times \text{Env} \rightarrow \text{False} \bigcup \text{Env} \bigcup \text{Pattern} \times \text{Env}$$

Demonstration. We simulate each of component of a Pattern. For simplicity, we use JavaScript syntax to show the simulation.

- **Several parameters in a function.** Using the higher-order function property, it is possible several parameters in Lambda Calculus. For example, an function add, which adds two numbers is:

```
var add = function(a) {
  return function(b) {
    return a + b;
} };
```

- **Abstractions.** False, Env, and JoinPoint are abstractions that can be simulated in Lambda Calculus. While the widely-known simulation of False is function (x) {return function (y) {return y;}}, the remaining two abstractions correspond to environments of bindings, which is simulated with the following of kind of function:

```
var addBinding = function(idBounded, value, previousEnv) {
  return function(id) {
    return id == idBounded? value: previousEnv(id);
} };

//where the empty environment is the following function
var mtEnv = function(id) {
  return "value not found for the id:"+id;
};
```

- **Pair.** The convention used to represent a pair in Lambda Calculus is:

```
var pair = function(x) {
  return function(y) {
    return function(f) {
      return (f(x) y);
} } };
```

```
var first = function(p) {
  return p(true); //true is function (x) {return function(y) {return x;}}
};

var second = function(p) {
  return p(false); //false as above
};
```

- **Body pattern.** Finally, the body of a pattern does not impose any restriction, *i.e.* all JavaScript can be used. Therefore, the body of a pattern is Turing complete.

5.4.2 Advice Model

An advice in ETM is a function parameterized by a join point and an environment; and the advice returns a value of the base language. This value is discarded when the advice kind is before or after.

$$\text{Advice: JoinPoint} \times \text{Env} \rightarrow \text{Value}$$

The join point that is passed as parameter to the advice function is the last join point matched by the pattern. The join point object has a proceed method, which permits the execution of the original computation at this join point. The environment corresponds to the environment returned by the associated pattern. Like pattern, advices can access the bindings of the environment. For example, the implementation of the advice of the stateful aspect used to support the autorevision feature is as follows:

```
var generateRevision = function(jp,env) {
  var editionList = env.e;
  var timeList = env.t;
  revisionControl.add(new Revision(editionList,timeList));
};
```

The generateRevision advice gets the list of editions and the list of timestamps from the environment. These values (editions and timestamps) were gathered by the edit pattern. Note that if two values are bound to the same identifier (*e.g.* e), they are aggregated as a list[1]. Unlike tracematches, a revision is generated with the list of timestamps because ETM allows developers to gather values beyond the context of the current join point.

[1]The most recent value is added at the end of the list.

5.4.3 Stateful Aspect Model

A stateful aspect is an object with at least three properties: a pattern, an advice, and a kind. The two first properties are functions, and the last property is a constant that represents the kind of the advice, signifying when the action of the advice is taken: before, around, and after the join point. For example, the implementation of the stateful aspect that implements the autorevision feature is:

```
var autorevision = {
  pattern: threeEditions,
  advice: generateRevision,
  kind: AFTER
};
```

The autorevision stateful aspect is composed of the threeEditions pattern and the generateRevision advice. The advice of this stateful aspect is triggered after the third call to methods insert or erase.

As the piece of code below shows, the deploy method of ETM is used to deploy a stateful aspect with global scope. As the current implementation of ETM is an extension of AspectScript, the scope of stateful aspects can be customized just like in AspectScript (*e.g.* using scoping strategies [Tanter, 2008b]).

```
ETM.deploy(autorevision);
```

Semantics of multiples matches in ETM. For simplicity, a stateful aspect of ETM cannot match multiple times. The following chapter describes OTM, which opens, amongst others, the semantics of multiples matches in ETM.

5.5 ETM in Action

This section shows how ETM addresses the limitations described in Section 5.3. Concretely, we implement every feature with the corresponding stateful aspect.

5.5.1 Reusability

Patterns are not reusable in most trace-based mechanisms. For example, consider the autoreport feature, where the pattern of a stateful aspect is composed of other patterns. We implement this feature using a stateful aspect in ETM:

```
var autoreport = {
  pattern: seqn(threeEditions,4),
  advice: function(jp,env) {
    var editionList = env.e;
    var timeList = env.t;
    sendReport(new Report(editionList,timeList));
  },
  kind: AFTER
};
```

The seqn pattern designator returns a pattern that matches n times the pattern passed as parameter. In this piece of code, the reused threeEditions pattern (Section 5.3) has to match four times. When the pattern is matched, the advice creates and sends a report with the corresponding editions. seqn is just a convenient abstraction on the top of the seq binary pattern designator:

```
var seqn = function(subSeq,n) {
  var mainSeq = subSeq;
  for (var i = 0; i < n - 1; ++i)
    mainSeq = seq(subSeq,mainSeq);
  return mainSeq;
};
```

For example, the evaluation of the expression seqn(threeEditions,4) becomes the expression seq(threeEditions,seq(threeEditions,seq(threeEditions,threeEditions). As the patterns threeEditions and seq show, patterns in ETM can be reused to create advanced patterns (or pattern designators).

In order to enhance the reusability of the threeEditions pattern, this pattern can be rewritten as the evaluation of a pattern designator. The piece of code below shows the threeX pattern designator and its evaluation to create threeEditions.

```
var threeX = function(pat) {
  return seqn(pat,3);
};
var threeEditions = threeX(edit);
```

5.5.2 Binding Expressiveness

A pattern can gather bindings during its matching. In most trace-based mechanisms, the expressiveness to gather bindings is limited. For example, consider the revision session feature, which generated a revision by session. In this feature, the pattern has to gather a variable-sized list of values. We implement this feature using a stateful aspect of ETM:

```
var revisionSession = {
  pattern: starUntil(edit,logout),
  advice: generateRevision,    //as Section 5.4
  kind: AFTER
};
```

The pattern returned by starUntil matches all editions until the user logs out of the text editor. Like the autorevision feature, the advice generates a revision of the document. The starUntil pattern designator returns a pattern that matches the star pattern until the until pattern matches:

```
var starUntil = function (star,until){
  return function (jp,env) {
    var result = until(jp,env);
    if (isEnv(result)){
      return result;
    }

    result = star(jp,env);
    if(isEnv(result)){
      return [starUntil(star,until),result];
    }

    return false;
} };
```

The starUntil pattern designator allows developers to express patterns that gather variable-sized lists of bindings.

5.5.3 Pattern Expressiveness

In most trace-based mechanisms, the flexibility to express patterns is limited. For example, consider the adaptive autorevision feature, where the pattern must alter its definition to match according to the current context. We implement this feature using a stateful aspect of ETM:

```
var adaptiveAutorevision = {
  pattern: adaptive(starUntil(edit,fiveMinutes),threeEditions,writingContext),
  advice: generateRevision,
  kind: AFTER
};
```

The adaptiveAutorevision stateful aspect matches a pattern of editions during five minutes if the context is “writing”. If the context is not “writing” (*i.e.* the context is “editing”), the stateful aspect instead matches a pattern of three editions. When some of two patterns match, the advice generates a revision of the document.

```
var fiveMinutes = function(jp,env) {
  if (env.t) {  //are there timestamps of editions?
    var firstTime = env.t[0];
    var lastTime = env.t[env.t.length - 1];
    return lastTime - firstTime >= 5*60? env: false; //5 minutes
  }
  return false;
};

var writingContext = function(jp,env) {
  if (env.e) {
    var firstTime = env.t[0];
    var lastTime = env.t[env.t.length - 1];
    return lastTime - firstTime <= 30 || env.e.length >= 4? env: false;
  }
  return true;  //no editions mean the writing mode
};
```

The fiveMinutes pattern matches if the time difference between the first and the last edition is greater than five minutes. The writingContext pattern matches if there are four editions during thirty seconds. The adaptive pattern designator returns a pattern that tries matching one of two patterns according to the context:

```
var adaptive = function(pat1,pat2,context) {
  return function(jp,env) {
    var result;
    //Context of the first pattern
    if (isEnv(context(jp,env))) {
      result = pat1(jp,env);
      return isPair(result)? [adaptive(result[0],pat2,context),result[1]]: result;
    }
    //Context of the second pattern
    else {
      result = pat2(jp,env);
      return isPair(result)? [adaptive(pat1,result[0],context),result[1]]: result;
    }
} };
```

Thanks to the fact that a pattern returns its next step or continuation at runtime, developers can express an ETM pattern (*e.g.* adaptive) that can adapt its own definition at runtime.

5.6 Summary

This chapter presented ETM, a model of a trace-based mechanism that allows developers to flexibly express stateful aspects. Various limitations of existing trace-based mechanisms are addressed because of the flexibility to express stateful aspects in ETM. Patterns and advices of ETM are functions, bringing all the benefits the base language to express them. The flexibility to express patterns allows developers to specify, for example, patterns that alter (or adapt) their own definitions at runtime, which is a unique feature in the current design space of trace-based mechanisms. In addition, the expressiveness of binding allows developers, for example, to gather variable-sized lists of bindings. Finally, this chapter presented a concrete and practical implementation, with its typed description, for JavaScript through different features of a Web application.

The processes of matching and advising are still hidden and cannot be customized in ETM. In the following chapter, we extend this model to allow developers to customize the processes of matching and advising. Through this extension, ETM becomes an Open Trace-based Mechanism, named OTM. To the best of our knowledge, OTM is the first trace-based mechanism that supports the customizations of its core semantics.

Chapter 6

OTM: Opening ETM[1]

ETM is a trace-based mechanism that supports the definition of expressive stateful aspects. However, the stateful aspects of ETM, like existing trace-based mechanisms, have fixed semantics, and a variety of semantics for stateful aspects is required to address some specific issues. For example, the semantics to match a trace of join points based on the criterion of time differs if a system is stand-alone or distributed because of the absence of a global clock for the latter system [Lamport, 1978][2]. As a consequence, developers must "code around" stateful aspects in contortive ways or create specialized trace-based mechanisms to satisfy their specific needs. This is because developers cannot flexibly express new semantics and/or combinations of different stateful aspect semantics of existing mechanisms.

In this chapter, we present a model of an Open implementation of a Trace-based Mechanism (OTM), where the core semantic elements of stateful aspects are open to customization. In addition, we present a complete and practical implementation of OTM for JavaScript. Using this implementation, we show how new semantic variants are useful in the development of (Web) applications; and we emulate the semantics of some trace-based mechanisms like ETM, HALO, and tracematches. As for ETM, we use Typed Racket [Tobin-Hochstadt and Felleisen, 2008] to offer a typed functional description of the API (Application Programming Interface) of OTM.

6.1 Introduction

Security, error detection, and crosscutting concern issues can modularly be addressed by stateful aspects (Chapter 3). Various trace-based mechanisms like ETM (Chapter 5) have been pro-

[1]The content of this chapter is a summary and adaptation of our SBLP 2010 paper "An Open Trace-Based Mechanism" [Leger and Tanter, 2010a].

[2]A detailed analysis of this subject is discussed on Chapter 7.

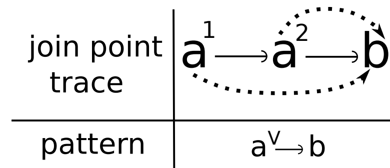

Figure 6.1: Possible matches of a pattern.

posed [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Eugster and Jayaram, 2009, Martin et al., 2005, Ostermann et al., 2005]. Each of these mechanisms establishes fixed and different semantics for their stateful aspects [Meredith et al., 2011]. In spite of this, there is no trace-based mechanism that allows developers to obtain custom semantics for a stateful aspect, like new possible variants and/or combinations of existing mechanisms. Concretely, developers cannot customize the semantics of the *matching process* and *advising process* of a stateful aspect (Figure 3.6a):

Matching process. This process carries out the match of a pattern. The pattern of a stateful aspect can match and bind free variables multiple times in a program execution trace. Figure 6.1 shows the possible matches of a pattern in a trace of join points. Apart from the definition of the trace, this pattern specifies that a free variable v is bound when a join point a is matched. Depending on the matching process semantics, the pattern can be matched either once or twice, where the first match binds v to 1 and the second one binds v to 2. In a particular trace-based mechanism, all stateful aspects have the same fixed matching process semantics. For example, EventJava [Eugster and Jayaram, 2009] only supports multiple matches of a pattern if the bound values are different. Therefore, if the semantics of this process must be different for a particular stateful aspect, developers end up overburdening the definitions of the pattern and/or advice. For instance, developers cannot cleanly express a stateful aspect that matches only once or once at a time.

Advising process. This process executes the advice of a stateful aspect. Whenever a pattern is (simultaneously) matched, the advice is executed (multiple times). Existing trace-based mechanisms commonly execute the advice for every match of the pattern, using the bindings gathered by each associated match. For example, if there are two matches in Figure 6.1, the first advice execution uses $v = 1$ and the second execution uses $v = 2$. Trace-based mechanisms have dif-

ferent semantics to compose and execute these advices. For instance, tracematches [Allan et al., 2005] chain and nest advice executions like in AspectJ [Kiczales et al., 2001]. Developers cannot currently customize the advising process of a stateful aspect. For example, existing trace-based mechanisms do not allow developers to repetitively execute the advice only while a resource, which is consumed by advice executions, is available.

The features of multiple matches of a pattern and simultaneous advice executions denote the main difference between aspects (Chapter 3) and stateful aspects[1]. Developers cannot currently customize the processes that support these features, which distinguishes one trace-based mechanism from another. In order to overcome the lack of customizations of semantics of matching and advising process, developers end up "coding around" stateful aspect definitions (patterns and advices) in contortive ways or creating specialized trace-based mechanisms (e.g. the *autosave* feature shown in Section 3.3.4). In this chapter, we present an Open Trace-based Mechanism (OTM), which follows open implementation design guidelines [Kiczales et al., 1997a]. OTM allows developers to customize the core semantic elements of a stateful aspect, concretely, the matching and advising process. In OTM, these semantic elements are flexibly customized using functions.

The rest of this chapter is organized as follows. Section 6.2 shows limitations of fixed stateful aspects through a Web application. Section 6.3 describes the OTM model with its typed description (Appendix B for details). Section 6.4 shows how OTM addresses limitations presented in Section 6.2 through new semantic variants. Section 6.6 describes how OTM emulates some trace-based mechanisms. Section 6.6 presents two evaluations of our proposal. Section 6.7 discusses design decisions. Section 6.8 summarizes this chapter.

Note. OTM, along with the examples presented in this chapter, is available online at http://pleiad.cl/otm. The OTM version for JavaScript currently supports the Mozilla Firefox browser without the need of an extension.

6.2 Limitations of Fixed Stateful Aspects

This section illustrates the limitations caused by fixed semantics of stateful aspects through the development of a Web application.

[1] Thereby, the choice of using the semantics of "a single match of the pattern at a time" for ETM helps as a first-step in understanding stateful aspects.

Adding a discount policy to an online computer store. A Web application of an online computer store is used to order one or more personalized computers, *i.e.* computers with personalized hardware. A client chooses a certain computer from the catalogue and adds this computer to a virtual shopping cart, which may contain more computers. Then, the client can personalize computer pieces. Finally, the Web application contains a checkout form asking for a desired payment method. Consider that the store now wants to add a discount policy, where every piece has a potential discount that is applied when the piece is added to a computer. Each discount that is associated to a piece is only valid for a period of time; however, *a)* this discount must be applied (even if it is not valid anymore) when the client checks out and *b)* only the three best discounts of each computer are applied. Implementing this discount policy in this Web application is clearly a crosscutting concern that can be modularized using a trace-based mechanism like ETM (Chapter 5). The following ETM stateful aspect implements the previous discount policy:

```
var discountPolicy = {
  pattern: seq(callAddComputer, starUntil(callAddPiece, callCheckout)),
  advice: function(jp, env) {
    var computer = env.computer;
    var idPieces = env.idPiece;
    var discounts = env.discount;

    //apply the three best discounts
  },
  kind: BEFORE
}
```

The discountPolicy stateful aspect applies three best discounts to a computer. During the matching of the pattern, it gathers all available discounts at that moment; these discounts are associated to pieces added to a computer. The piece of code below shows the patterns callAddComputer, callAddPiece, and callCheckout. The first pattern gathers the computer, the second pattern gathers the idPiece and discount values, which becomes lists when callAddPiece matches twice or more times[1]. The third pattern conserves the environment when it matches (*i.e.* when the client does checkout).

```
var callAddComputer = function(jp, env) {
  if (jp.isCall() && jp.fun == addComputer) {
```

[1] In Chapter 5, we explain that if the same identifier in ETM is bound with two or more values, the identifier is bound with a list that contains these values.

```
    return env.bind("computer",jp.args[0]); //1st argument to addComputer
  }
  return false;
};

var callAddPiece = function(jp,env) {
  if (jp.isCall() && jp.fun == addPiece && env.computer == jp.args[0]) {
    var idPiece = jp.args[1]; //2nd argument to callAddPiece
    return env.bind("idPiece",idPiece).bind("discount",getDiscount(idPiece));
  }
  return false;
};

var callCheckout = function(jp,env){
  return jp.isCall() && jp.fun == checkout? env: false;
};
```

Suppose that the store now wants to add three restrictions to the discount policy. These restrictions need to customize the semantics of the discountPolicy stateful aspect. For every semantics that needs to be customized, we show how patterns and advices are currently modified to overcome the limitations of the fixed stateful aspects.

6.2.1 Fixed Matching Process

In the previous chapter, we mention that the matching semantics of ETM stateful aspects only supports a single match of a pattern at a time. In ETM, therefore, the discount policy can apply discounts to only one computer in the shopping cart: the first computer added. Fortunately, most trace-based mechanisms [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Martin et al., 2005, Ostermann et al., 2005, Eugster and Jayaram, 2009] support multiple matches of a pattern; therefore, the discount policy can apply discounts to various computers of the cart, one computer for each match. However, developers cannot customize the semantics of the matching process, which is necessary in scenarios like the followings two.

6.2.1.1 Policy Discount Restriction: One Computer by Category gets Discounts

Consider a restriction to the existing policy discount of the online store. This restriction establishes that the discount policy applies the three best discounts to one computer by category (*e.g.* laptops, desktops, tablets) in the shopping cart. For example, if there are two laptops in the shopping cart, the discount policy only applies discounts to one laptop. To implement this

restriction, it is necessary to have a new potential match of the pattern only if a computer of a different category is added.

As existing trace-based mechanisms do not allow developers to customize the matching process of a stateful aspect, developers modify advices in order to overcome the lack of this customization. For example, the implementation of this restriction in existing trace-based mechanisms requires the following modification of the advice:

```
var categories = //a container of categories
var advice = function(jp,env) {
  var category = getCategory(env.computer);
  if (!categories.contains(category)) { //Is this category different?
     categories.add(category);
     //execute the original advice
} };
```

This advice overlaps its responsibility to apply discounts with the responsibility to correctly match the pattern (*i.e.* the pattern is tangled with the definition of the advice). Therefore, the execution of this advice does not necessarily mean that the discounts are applied.

6.2.1.2 Policy Discount Restriction: Some Promotions are Temporary

Suppose that we have an additional restriction to the existing policy discount. This restriction establishes that the discounts of some computer categories last a period of time (*e.g.* the Christmas season) and these discounts are not valid if a client checks out after this period. To implement this restriction, the possible matches that contain computers with categories that have an unsatisfied temporary restriction must be removed.

To implement this restriction in most trace-based mechanisms, it is necessary to modify the callCheckout pattern to verify whether the category of a computer still has promotions associated or not:

```
var callCheckout = function(jp,env){
  return jp.isCall() && jp.fun == checkout &&
  //new conditions
  !(hasTemporaryPromotion(getCategory(env.computer)) && afterChristmas())? env: false;
};
```

In tracematches [Allan et al., 2005], the implementation of the policyDiscount stateful aspect and this restriction could be different:

```
tracematch(int computer) {
   sym callAddComputer before: pcCallAddComputer() && args(computer);
```

```
  sym callAddPiece before: pcCallAddPiece();
  sym callCheckout before: pcCallCheckout();

  sym afterChristmas: //the after Christmas event

  callAddComputer callAddPiece* callCheckout { //regular expression
    //the advice body
} }
```

This tracematch stateful aspect almost[1] implements the policy discount. In addition, it implements the restriction using the definition of the afterChristmas symbol, which does not appear in the pattern definition, expressed by a regular expression. This symbol is used to avoid a match of the pattern if the after Christmas event occurs during the matching (tracematches require contiguous occurrences of the events denoted by the symbols in the regular expression). However, this solution is not completely correct because it does not take into consideration the category of a computer. In tracematches, this category cannot be gathered if there is no symbol (pointcut) that exposes this binding and a regular expression uses this symbol.

6.2.2 Fixed Advising Process

A pattern can match simultaneously. If there are simultaneous matches of a pattern, the same advice is executed several times, each one with bindings gathered by the match of the pattern. Existing trace-based mechanisms do not allow developers to customize the advising process, which is necessary in scenarios like adding the last restriction to the discount policy.

6.2.2.1 Policy Discount Restriction: Only the Computer with more Pieces gets Discounts

Here, we focus on a policy discount with a strong restriction. This restriction establishes that the discount policy only applies the discounts to the computer with the most pieces added. In this scenario, it is not possible to overburden the pattern definition to implement this restriction because the number total of pieces of every computer is only known when a client goes to checkout. The implementation of this restriction requires that the stateful aspect only executes the advice with the match whose bindings gathered contains the computer with more pieces.

[1] In tracematches like other mechanisms, a pattern cannot gather a list of bindings during its matching (see Chapter 5); therefore, the callAddPiece symbol of this piece of code cannot gather all discounts of the pieces added.

As existing trace-based mechanisms do not allow developers to customize the advising process, developers have to code around the advice. For example, the implementation of this restriction in existing trace-based mechanisms requires the following modification of the advice:

```
var maxPieceNumber = 0;
var computerCounter = 0;
var computerSelected;

var advice = function(jp,env) {
  var computerList = jp.args[0];
  //select the computer with the most pieces
  if (maxPieceNumber < getNumberOfPieces(env.computer)) {
    maxPieceNumber = getNumberOfPieces(env.computer);
    computerSelected = env.computer;
  }

  if (++counter == computerList.length) {
    //execute the original advice for the selected computer
  }
};
```

The advice of the discountPolicy stateful aspect must be modified in order to overcome the lack of customization of this process. The original advice is only executed once when the computer with more pieces is found.

6.2.3 Summary

Like the solutions described in this section, coding around patterns and advices in contortive ways is a common solution used to overcome the limitations of fixed stateful aspects. Apart from modifying the code, these solutions overlap the responsibilities of patterns with advices: the aspect issue (between pointcuts and advices) that stateful aspects address (Chapter 3). In other words, if stateful aspects are fixed, the issue about the coupling between the *selection* of an execution point and the *action* of this point reappears.

6.3 OTM

OTM is a model of an open implementation [Kiczales et al., 1997a] of a trace-based mechanism. OTM allows developers to flexibly customize the matching and advising processes of a stateful aspect. Therefore, developers can address the previous issues through these customizations without the need to modify patterns and/or advices in an *ad hoc* manner.

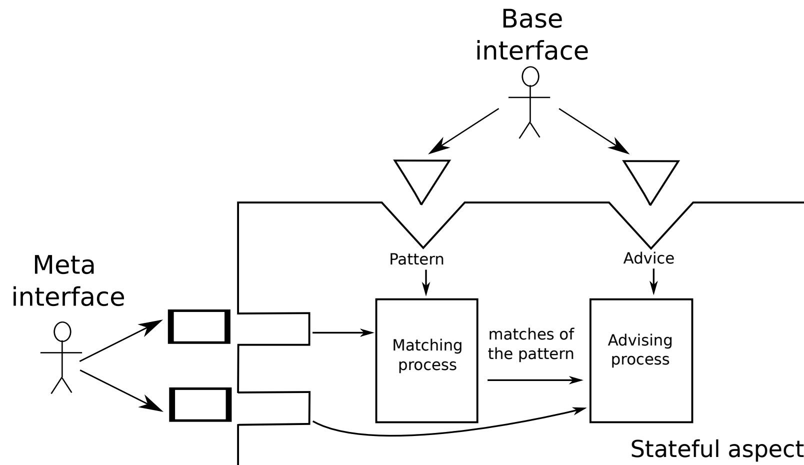

Figure 6.2: The base and meta interface of a stateful aspect.

Trace-based mechanisms support the definitions of stateful aspects. Figure 3.6 shows that a stateful aspect is composed of a pattern and an advice. The pattern is the specification of a program execution, which the matching process of a stateful aspect has to match (multiple times); and the advice is the piece of code that is (simultaneously) executed by the advising process. We now explain the OTM model, which focus on the openness of the matching and advising processes of a stateful aspect.

OTM overview. As mentioned in Chapter 3, a system that supports open implementations provides both a *base* and a *meta* interface. The role of the base interface is to provide developers with a simple and natural interface for the use of the system. The role of the meta interface is provide developers with an interface to customize part of the system semantics. Figure 6.2 shows the interface base interface (triangles) and meta interface (rectangles) of a stateful aspect. Like ETM, OTM allows developers to cleanly express stateful aspects (pattern and advice), it also allows developers to customize the semantics of matching and advising process of every stateful aspect.

6.3.1 Matching Process

Conceptually, when a stateful aspect is deployed, the matching process creates a *matcher* that has to match the program execution trace specified by the pattern. If a matcher matches a join point, this matcher can advance to match the next join point of the pattern or end matching if

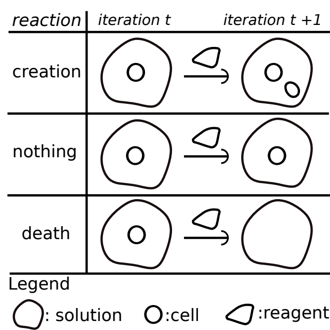

Figure 6.3: Different kinds of reactions of a cell to a reagent.

there is no next join point to expect. To support the multiple matches of a pattern, a stateful aspect can have various matchers at the same time. Several algorithms have been proposed to implement this process: tracematches [Allan et al., 2005] use a nondeterministic finite-state automaton, Alpha [Ostermann et al., 2005] uses queries to a database, HALO [Herzeel et al., 2006] and EventJava [Eugster and Jayaram, 2009] use the Rete algorithm [Forgy, 1982], an efficient pattern matching algorithm used for expert systems [Darlington, 2000], etc. These algorithms define the unalterable semantics of a matching process. Therefore, these algorithms are not very useful to achieve the goal of OTM. We develop a self-replication algorithm [Neumann, 1966], called MatcherCells, that allows developers to flexibly express the semantics of a matching process [Leger and Tanter, 2012]. Next, we briefly present MatcherCells.

6.3.1.1 MatcherCells

Self-replication algorithms in a nutshell. Self-replication algorithms are inspired by the cellular behavior [Neumann, 1966]. Concretely, these algorithms emulate the reactions of a set of biological *cells*, in a *solution*, to a trace of *reagents*. Figure 6.3 shows the different possible reactions of a cell to a reagent. The reaction of a cell to a reagent can be *a)* the creation of an identical copy of itself with a small variation in order to persist in the solution, *b)* nothing, *c)* death, *d)* or some of these combinations. An algorithm that follows self-replicating behavior is defined by a pair (C_0, R), where C_0 is the set of first cells into a solution (*a.k.a.* seeds) and R is the set of rules that governs the evolution of the solution.

In MatcherCells, a cell contains the pattern that should be matched, bindings gathered during this matching, and a reference to its creator. Cells react to join points, which correspond to

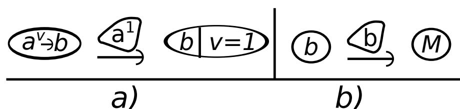

Figure 6.4: *a)* The cell creates a cell that expects to match the next join point and keeps the bindings. *b)* When a cell matches the last join point specified, the cell creates a match cell.

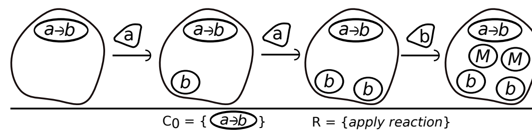

Figure 6.5: Using MatcherCells to match multiple times.

reagents. As Figure 6.4a shows, if a cell matches a join point, this cell creates a new cell that expects to match the next join point specified by the pattern. In addition, this new cell contains the possible bindings gathered when the join point was matched. These bindings are stored on an environment, which can be accessed by the pattern. Figure 6.4b shows that when there is no next join point to match, a *match cell* is created to indicate a match of the trace of join points.

MatcherCells is an algorithm to flexibly implement a matching process, whose semantics is the defined by a set of rules (R). Each rule defines a portion of the semantics of the matching process and the set of rules entirely defines the semantics. For example, Figure 6.5 shows that the "apply reaction" rule only applies the reaction of each cell. This simple rule supports the semantics of multiple matches of a pattern. This figure also shows that the solution begins with a set of only one seed (C_0), which contains the specified pattern, and evolves until reaching two matches of the pattern.

Figure 6.6 shows that different sets of rules state different semantics for a matching process. Figure 6.6a shows that the "kill creators" rule kills the cells that create a new cell. Adding this rule, a pattern cannot match multiple times anymore. Figure 6.6b shows that the "add seed" rule adds a seed if there are no cells, or only match cells, in the solution. This rule allows a pattern to match again. Figure 6.6c shows that the "keep seed" rule always keeps a seed to permit to begin a new match of the pattern at any moment[1]. Finally, Figure 6.6d shows that the "life-time for a trace" rule kills all cells whose period of time of the join point trace time has exceeded a determined period. This rule allows developers to define patterns that only match traces of join points that occur at a period of time.

6.3.1.2 MatcherCells Implementation

This section first explains how a cell reacts to a join point, and then explains how a rule can be created and composed with other rules.

[1] This rule supports multiple matches in a restricted way: multiple matches of a pattern can begin.

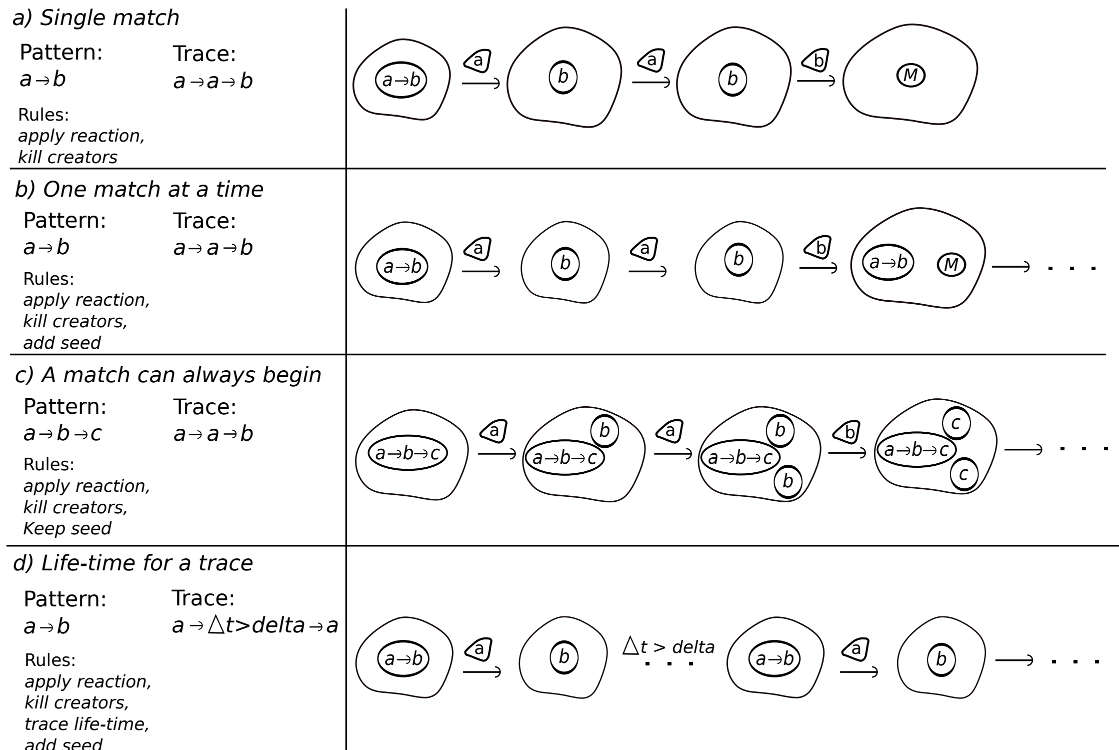

Figure 6.6: Different semantics for a matching process.

```
react: Cell × JoinPoint × [Env × Pattern × Cell → Env] → Cell
```

The react function carries out the reaction of a cell to a join point. If the cell matches the join point, the function returns a new cell, otherwise the function returns the same cell. The last and optional parameter[1] of react is a function that allows developers to add information to a cell in its creation (more on this at the end of this section).

```
rule: List<Cell> × JoinPoint → List<Cell>
```

A rule is a function that takes as parameters a list of cells and a join point, and returns the list of cells of the next iteration. For example, the "apply reaction" rule implementation is:

```
var applyReaction = function(cells,jp) {
  return removeDuplicates(
          append(cells,map(function(cell){return react(cell,jp);},cells)));
};
```

The applyReaction function returns the cells and their reactions. A cell, whose reaction is itself, is in the list of cells and the list of reactions. This means that this cell is duplicated when

[1]The notation [...] denotes an optional parameter.

both lists are joined. To prevent this duplication, the removeDuplicates function is used. Using rule designators (*i.e.* functions that return rules), developers are able to create rules that can be composed:

```
var killCreators = function(rule) {
  return function(cells,jp) {
    var nextCells = rule(cells,jp);
    return difference(nextCells,getCreators(nextCells,cells));
} };

var addSeed = function(pattern) {
  return function(rule) {
    return function(cells,jp) {
      var nextCells = rule(cells,jp);
      return length(nextCells) == 0 || onlyMatchCells(nextCells)?
        append(nextCells,[seed(pattern)]): nextCells;
} } };

var keepSeed = function(pattern) {
  return function(rule) {
    return function(cells,jp) {
      var nextCells = rule(cells,jp);
      return countSeeds(nextCells) == 0?
           append(nextCells,[seed(pattern)]): nextCells;
} } };
```

These rule designators are parametrized by a rule, which corresponds to the rule that should be applied before the current one. The rule returned by killCreators first applies a previous rule (*e.g.* applyReaction) to obtain a list of cells, where the cells that created new cells are removed for the next iteration. The addSeed[1] rule designator returns a rule that adds a seed if there are no cells or only match cells. Finally, keepSeed always keeps a seed in the solution. The composition of rules allows developers to define the full semantics of a matching process. For example, the compositions of rules to obtain the different semantics of Figure 6.6 are:

```
var singleMatch = killCreators(applyReaction);
var oneMatchAtATime = addSeed(pattern)(killCreators(applyReaction));
var aMatchCanAlwaysBegin = keepSeed(pattern)(killCreators(applyReaction));
var timingToMatch = addSeed(pattern)(traceLifeTime(delta)(killCreators(applyReaction)));
```

Adding information to cells. Some rules may need that all cells contain specific information.

[1]Notice that addSeed is in fact a higher-order rule designator, parameterized by the original pattern or a new one.

For example, traceLifeTime needs that all cells have the time when the matching of the trace begins (cell.env.time):

```
var traceLifeTime = function(delta) {
  return function(rule) {
    return function(cells,jp) {
      var nextCells = rule(cells,jp);
        return filter(function(cell) {
          return currentTime() - cell.env.time <= delta;
        },nextCells);
} } };
```

To add information to cells, the third parameter of the react function is used. This parameter is a function that receives the values with which a cell will be created, and returns the initial environment of bindings of a cell. For example, to annotate a cell with the trace time, one needs to provide the following function:

```
var creation = function(env,pattern,creator) {
  var time = isSeed(creator)? currentTime(): creator.time;
  return env.bind("time",time);
};
```

In some scenarios like traceLifeTime, intrinsic attributes (*e.g.* time, physical space, etc) should be present in each cell from its creation until the matching of the trace in order to be able to compare cells. In other words, these attributes crosscut through the matching of a trace [Holzer et al., 2011]. The feature of customized information of cells and rules that use this information allow MatcherCells developers to modularize the treatment of these intrinsic attributes. For example, if MatcherCells would not support the adding of information to a cell, the implementation of a pattern that must match at a specific period of time requires its modification every time a join point is matched in order to verify the time passed.

6.3.2 Advising Process

When one or more matchers match, the advising process is executed. In MatcherCells, these matchers correspond to match cells. As mentioned previously, they are passed from the solution to the advising process. The advising process is carried out by a function advising: List<Cell> $\times$ JoinPoint $\times$ Advice $\rightarrow$ Value. This function takes a list of match cells and executes the advice of the stateful aspect with the last join point matched and the bindings gathered by each match cell. The value returned by this function depends on the advice kind;

if the advice kind is around, the function returns a value, which replaces the value returned by the original computation. Instead, if the advice kind is before or after, the returned value is discarded. For example, the advising process that just executes the advice with each match cell is[1]:

```
var advisingProcess = function(matchCells,jp,advice) {
  return last(map(function(matchCell) {
    return advice(jp,matchCell.env);
  },matchCells));
};
```

The advice of a stateful aspect is executed with the last join point matched and the environment of bindings gathered, which are available on cell.env. This advising process only returns the result of the last advice executed.

6.3.3 Stateful Aspect

In OTM, the customizations of the semantics of the matching and advising processes are implemented through functions. In the OTM implementation for JavaScript, a stateful aspect object is customized through adding the property matching and/or advising:

```
var statefulAspect = {
  //pattern, advice, and kind
  matching: function(..) {..},
  advising: function(..) {..},
}
```

Cells and matchers. In this section, we used MatcherCells to describe how the processes of matching and advising of a stateful aspect work and are customized. A cell really corresponds to a matcher in some state (*e.g.* beginning the matching of a trace).

6.4 OTM for New Semantic Variants

This section presents the new semantics of the matching and advising processes to implement the three restrictions to the policy discount presented in Section 6.2.

[1] If the advice kind is around, a nested advice composition is required.

6.4.1 Customizing the Matching Process

Using MatcherCells, OTM allows developers to customize the matching process of a stateful aspect. This section illustrates how to cleanly address the limitations described in Section 6.2.1.

6.4.1.1 Policy Discount Restriction: One Computer by Category gets Discounts

The implementation of this restriction in the store Web application requires a new potential match of the pattern if only the category of the computer gathered by this pattern differs from the computers already added in the shopping cart. To customize the matching process as is required by this restriction, the following composition of rules must be specified in the discountPolicy stateful aspect:

```
discountPolicy.matching = keepSeed(pattern)(filterByCategory(killCreators(applyReaction)));
//where the pattern is seq(callAddComputer,starUntil(callAddPiece,callCheckout))
```

This composition represents the “a match can always begin” semantics (Figure 6.6c) plus the filtering by categories. After applying the reaction to each cell and killing the cells that created new cells, the rule returned by the evaluation of filterByCategory filters the new cells that contain computers whose categories are different from current ones. The implementation of the filterByCategory rule designator is:

```
var filterByCategory = function(rule) {
  return function(cells,jp) {
    var nextCells = rule(cells,jp);
    var categories = getCategories(getComputers(nextCells));
    var newCells = difference(nextCells,cells);

    var filteredNewCells = filter(function(newCell) {
      return some(function(category) {
         return getCategory(newCell.env.computer) == category;
      },categories);
    },newCells);

    return difference(nextCells,filteredNewCells);
} };
```

Categories of every computer are obtained from the environment of bindings that each cell contains. Each category of the new cells is compared to current categories to know whether this category is new or not. Finally, only the new cells with new categories are returned. Notice that the decision of filtering depends on category, a binding that is not directly included in cell environments. Given that the matching process is closed for existing trace-based mechanisms, it

is not possible that the semantics of multiple matches of a pattern depends on bindings that are not available on the pattern definition.

6.4.1.2 Policy Discount Restriction: Some Promotions are Temporary

The implementation of this restriction in the Web application requires removing cells whose categories of computers have an obsolete promotion. The following composition of rules allows this restriction:

```
discountPolicy.matching =
  keepSeed(pattern)(removeObsoletePromotions(killCreators(applyReaction)));
```

Similar to the previous example, this composition represents the semantics of "a match can always begin" (Figure 6.6b), plus the removing of cells with obsolete promotions. The implementation of the filterByCategory rule designator just removes cells whose categories have an expired promotion:

```
var removeObsoletePromotions = function(rule) {
  return function(cells,jp) {
    var nextCells = rule(cells,jp);
    return filter(function(nextCell) {
      return !(hasTemporaryPromotion(getCategory(nextCell.env.computer)) && afterChristmas());
    },nextCells);
} };
```

6.4.2 Customizing the Advising Process

OTM allows developers to customize the advising process. This section shows that the customization of the advising process allows developers to implement the last restriction to the discount policy shown in Section 6.3.2.

6.4.2.1 Policy Discount Restriction: Only the Computer with more Pieces gets Discounts

The implementation of this restriction requires to only execute the advice with the match cell whose computer has added more pieces by a client. The following function is used to customize the advising process as required by this restriction:

```
discountPolicy.advising = function(matchCells,jp,advice){
  //sorting cells by number of pieces added
  matchCells = sort(function(matchCell1,matchCell2){
    return length(matchCell2.env.idPiece) - length(matchCell1.env.idPiece);
```

```
  },matchCells);

  var cellWithMorePieces = matchCells[0];
  advice(jp,cellWithMorePieces.env);
};
```

The match cell that represents the computer that has more pieces added is precisely the match cell that has gathered more pieces. Hence, this function executes the advice with only a match cell: cellWithMorePieces. In detail, the function sorts all match cells and executes the advice with only the match cell that has gathered more pieces. Finally, the value returned by the advice is ignored because the advice kind of the policyDiscount stateful aspect is before.

6.4.3 Conclusion

This scenario, which places certain restrictions on a discount policy for an online store, shows the need to have new and different stateful aspect semantics. Using points of openness of OTM, we can express a stateful aspect with the semantics that requires the discount policy restrictions. The next section shows how to customize the stateful aspect semantics to emulate some existing trace-based mechanisms.

6.5 OTM for Emulating Existing Trace-based Mechanisms

OTM is a model of an open implementation of a trace-based mechanism that allows developers to emulate the semantics of existing trace-based mechanisms. In this section, we use OTM to emulate the semantics of ETM (Chapter 5), HALO [Herzeel et al., 2006], and tracematches [Allan et al., 2005]. For each mechanism, we show the necessary customizations of matching and advising processes.

6.5.1 ETM

ETM is an expressive trace-based mechanism presented in Chapter 5, where developers can express adaptable patterns using the full power of the base language. As mentioned in the previous chapter, the ETM semantics, for simplicity, only permits one match at a time. Currently, the default OTM semantics follows this semantics.

Matching process. ETM stateful aspects need to keep and evaluate only one cell at the same

time. In other words, the matching process of ETM is the composition of rules described in Figure 6.6b (one match at a time):

```
var etmMatching = addSeed(pattern)(killCreators(applyReaction));
```

Advising process. As only one match cell can be found at a time, the advising process has to execute the advice for only one match cell:

```
var etmAdvising = function(matchCells,jp,advice){
  return advice(jp,matchCells[0].env);
};
```

6.5.2 HALO

The HALO [Herzeel et al., 2006] semantics intuitively[1] signals that if a pattern matches twice or more times simultaneously, the advice is only executed using match cells whose gathered bindings differ from one another.

Matching process. HALO stateful aspects support the multiple matches of a pattern without any restriction, therefore, the matching process only uses the applyReaction rule (Section 6.3.1.1):

```
var haloMatching = applyReaction;
```

Advising process. If there are two or more matches of a pattern simultaneously, the advice is only executed with match cells with different bindings:

```
var haloAdvising = function(cells,jp,advice){
  var envs = getEnvs(cells);

  //filtering environments that contain the same contextual information
  var filteredEnvs = removesDuplicates(function(env1,env2) {
    return equal(env1,env2); //same bindings?
  },envs);

  return last(map(function(env) {
    return advice(jp,env);
  },filteredEnvs));
};
```

[1] The real implementation of HALO uses the Rete algorithm [Forgy, 1982] to efficiently match program execution traces. Despite this point, the semantics presented here does not differ from HALO semantics.

```
tracematch() {
  sym a after: call(* *.a(..));
  sym b after: call(* *.b(..));
  sym c after: call(* *.c(..));

   a b  {print("tm 1");}
}
```

t1

```
tracematch(int v) {
  sym a after: call(* *.a(..));
  sym b after: call(* *.b(int))
        && args(v);
  sym c after: call(* *.c(..));

  a b  {print("tm 2");}
}
```

t2

```
tracematch(int v) {
  sym a after: call(* *.a(..));
  sym b after: call(* *.b(int))
        && args(v);
  sym c after: call(* *.c(..));

  a b c {print("tm 3");}
}
```

t3

Figure 6.7: Pieces of code of three tracematches: $t1$, $t2$, and $t3$.

The haloAdvising function first filters environments in order to only catch the environments with different bindings. Finally, the function executes the advice with each environment of filteredEnvs.

6.5.3 Tracematches

Like HALO, the semantics of tracematches intuitively signals that the advice is only executed using matches with different bindings. Unfortunately, this intuitive semantics is not completely correct because it does not take into account the effect of *symbols* on the matching of a program execution trace. The use of symbols is one of the defining characteristics of tracematches, which complicates the understandability of its semantics at first glance. For example, we see that with two similar patterns $a \to b^v \to c$ and $a \to b \to c$ and the program execution trace $a \to b^1 \to b^2 \to c$, there are two matches of the first pattern but there is *no match* of the second one.

6.5.3.1 Tracematch Symbols

Like JavaMOP [Chen and Roşu, 2007, Meredith et al., 2011], tracematches use of symbols to define the pattern of stateful aspect. A tracematch symbol is composed of a pointcut plus a set of bindings gathered by the pointcut. The symbols are used to define patterns as regular expressions that should match. In addition, these symbols are used *a)* to discard evaluations of cells with unnecessary join points, *b)* to remove cells that do not satisfy conditions imposed by these symbols, and *c)* to selectively create cells. We now illustrate these three points.

Figure 6.7 presents pieces of code of three tracematches ($t1$, $t2$, and $t3$), and Table 6.1 shows the number of matches of these tracematches for the three different traces. In the first trace, $t2$ and $t3$ match twice because there are two occurrences of the call to the b function, and $t1$ only matches once. Although $t1$ and $t2$ match the same program execution ($a \to b$), the binding of v

Program execution trace	t1	t2	t3
a(); b(1); b(2); z(); c();	1	2	2
a(); b(1); b(1); c();	1	1	0
a(); b(1); b(2); b(1); c();	1	2	1

Table 6.1: Numbers of matchings for each tracematch.

of the b symbol allows $t2$ to match twice. This is so because a symbol is also identified by the set the of bindings gathered, meaning that calls b(1) and b(2) represent two different instances of the b symbol for $t2$. Also, notice the call to the z function is not considered by either of these tracematches because there is no symbol that matches this call. If there were a symbol that identifies the call to z in any tracematch, there would not be any match in the second and third trace. In the second trace, $t3$ matches zero times because its regular expression ($a \to b \to c$) does not match with the trace of symbols $a \to b \to b \to c$ because there are two b symbols identified and not only one. In the third trace, the binding gathered by the pointcut that matches the second call b(1) does not again allow $t3$ to match more times.

We now present the emulation of the tracematches semantics in OTM through the customizations of the matching and advising processes. Basically, the matching process in tracematches does the following: *a)* only evaluates a cell with join points that correspond to symbols of it, *b)* removes a cell that matches a symbol (of it) that does not correspond to the next symbol in the pattern, and *c)* only permits the creation of a new cell if its creator has not created another cell or matches a join point that corresponds to a new instance of a symbol. For example, b(1) and b(2) are two different instances of the b symbol. The *c* point allows tracematches to match multiple times a pattern. The advising process in tracematches composes and executes advices like in AspectJ [Kiczales et al., 2001].

6.5.3.2 Matching Process

In tracematches, in order to *a)* evaluate cells with only certain join points and *b)* remove cells that match a wrong symbol, it is necessary to define a new version of the applyReaction rule:

```
1 var tracematchApplyReaction = function(alphabet){
2   return function applyReaction(cells,jp){
3     var nextCells = [];
4     //Visiting each cell
5     for (var i = 0; i < length(cells); ++i) {
6       //a) evaluate with only certain join points
```

```
 7         if (isThisJpASymbolForThisCell(alphabet,jp,cells[i])){
 8           var nextCell = react(cells[i],jp);
 9           if(nextCell != cell) {
10             nextCells = add(cells[i],nextCells); //b) do not remove cell
11             nextCells = add(nextCell,nextCells);
12           }
13         }
14         else
15           nextCells = add(cells[i],nextCells); //b) do not remove cell
16       };
17
18       return nextCells;
19   } };
```

The applyReaction rule is now a rule designator that takes a set of definitions of symbols, named alphabet. Line 8 shows that a cell is only evaluated if the jp join point is a symbol for this cell. The lines 10 and 15 show that a cell is not removed if this cell advances in the matching of the pattern or this cell is not evaluated with the join point. In other words, a cell is removed if it matches an incorrect symbol. Line 11 shows that each time a cell creates a new cell, this latter cell is added. The following rule designator is used to know whether this new cell will be for the next iteration or not:

```
var differentBindings = function(rule) {
  return function(cells,jp) {
    var nextCells = rule(cells,jp);
    var newCells = difference(nextCells,cells);

    //filtering new cells with the same environment
    var filteredNewCells = filter(function(newCell) {
      return some(function(sisterCell) {
        return !isSeed(newCell.creator) && equal(newCell.env,sisterCell.env);
      },sisters(newCell,cells));
    },newCells);

    return difference(nextCells,filteredNewCells);
} };
```

The differentBindings rule designator returns a rule that only keeps a new cell if its bindings are different from its sisters or its creator is a seed. Finally, the rule composition to emulate the matching process of tracematches is:

```
tracematchMatching = keepSeed(pattern)(differentBindings(tracematchApplyReaction(alphabet)));
```

6.5.3.3 Advising Process

If the advice kind of a stateful aspect in tracematches is before or after, the advising process executes the advice for every match cell in undetermined order. Instead, if the advice kind is around, the advising process chains the advice executions, each one nesting the next one. Hence, we customize the advising process in the following way:

```
var tracematchesAdvising = function(matchCells, jp, advice) {
  var envs = getEnvs(matchCells);
  //randomize elements of envs

  //chain advice executions like AspectJ
  var chainedAdvices = chainAdvice(advice, envs, isAround());
  return chainedAdvices(jp);
}
```

Notice that it is not necessary to filter by environments that contain different bindings (like in HALO) because it is now filtered by the matching process.

6.5.4 Conclusion

Through the points of openness of OTM, we can instantiate different trace-based mechanisms. In addition, OTM enjoys the same expressiveness of ETM to define patterns. For example, any instantiation allows developers to define a pattern like $(a^v)^*$ that gathers one or more lists of bindings during its matching. The former pattern is not currently supported in most trace-based mechanisms. However, with two small modifications of the matching process of tracematches presented here, it is possible to express such a pattern. First, the binding gathered by a symbol that it is used with operators like * and $^+$ should become a list that contains this value instead of one value only (like ETM). Second, the comparison that determines if two symbols used with * and $^+$ should compare elements of the lists instead of only using the current tracematches comparison (“==”).

6.6 Evaluations

This section presents two evaluations of OTM. First, we extended a third-party Tetris game to support a new mode of game. Second, we run benchmarks in order to measure the overhead of the use of the pattern language and the matching process algorithm of OTM.

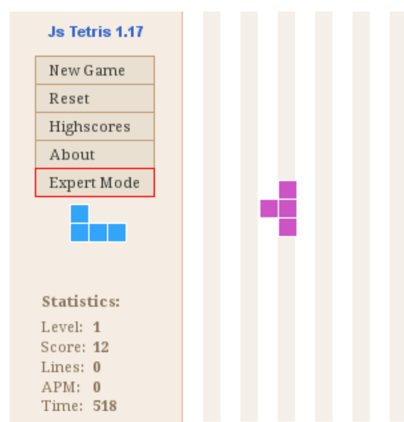

Figure 6.8: A Tetris game with a new mode: "Expert Mode".

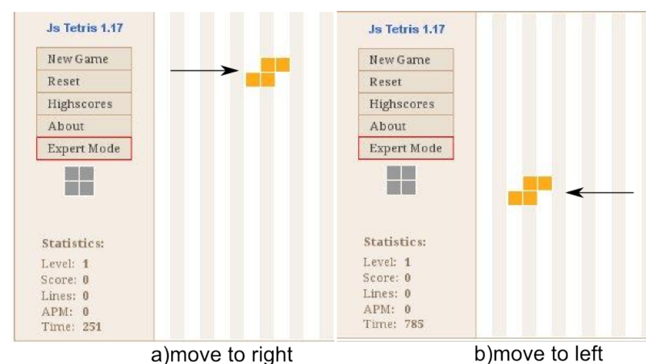

Figure 6.9: A sequence of two moves: right follows by left.

6.6.1 Adding an Expert Mode in Tetris

A large number of games for JavaScript have been made available on the Web by programmers[1]. As Figure 6.8 shows, we extended a Tetris game[2] to support a new mode of game: *expert mode*. This mode penalizes the indecision of a gamer to place a piece in an horizontal way; if the gamer changes the direction of the translation of a piece, the piece speeds up its fall (Figure 6.9).

```
var expertMode = {
    kind: AFTER,
    pattern: or(seq(call(left),call(right)),seq(call(right),call(left))),
    advice: function(jp) {
        tetris.puzzle.speed = 0; //"0" means the fastest
} };
```

The expertMode stateful aspect implements the new mode in this Tetris game. The pattern expertMode matches a move from left to right or vice versa. The associated advice just speeds up the fall of a piece. For every new piece, the stateful aspect must remove its possible matches because a gamer has a new opportunity to choose the direction. To achieve this goal, we customize the ETM matching process semantics (a single match at a time) with an except rule:

```
var except = function(rule) {
  return function(cells,jp) {
    return jp.isCall() && jp.fun == tetris.puzzle.reset? // new piece?
           [] : rules(cells,jp);
} };
```

[1] http://www.webresourcesdepot.com/25-amazing-javascript-games-some-fun-and-inspiration/
[2] http://www.gosu.pl/tetris/

```
expertMode.matching = customEtmMatching(except);
```

The except rule removes all cells if the reset method is called, which means a new piece is deployed. In conclusion, to add a new mode to the Tetris game with OTM, it is not necessary to modify the Web application.

6.6.2 Benchmarks

We ran performance tests of the JavaScript version OTM. The results of these tests were compared to tracematches. For this experiment, an Intel Core 2 Duo, 2.66 GHz with 2GB of RAM running Ubuntu 10.04 (kernel 2.6.32) was used. For the JavaScript version of OTM, we used the Firefox JavaScript interpreter (version 1.8.0). We used the *abc*[1] compiler (version 1.3.0) for tracematches.

The experiment uses the implementation of the *autosave* example (Section 3.3.5). The base code below together the aspect implementation is executed 500,000 times with a pattern that matches sequences of different lengths of editions, from 5 to 30 editions. In the experiment, we measure the average time used to execute the implementation. Two approaches are used for the implementation of the example: *a)* an AspectScript aspect, which keeps a counter of editions, and *b)* an OTM stateful aspect with the matching process semantics of ETM and tracematches. Finally, we execute the same experiment with an aspect of AspectJ and a stateful aspect of tracematches.

```
//PATTERN_LENGTH varies to 5, 10, 15, 20, 25, and 30 according to the experiment setting
//this code is executed 500,000 times in order to get an average.
start = getCurrentTime();
for (i = 0; i < PATTERN_LENGTH; ++i) {
  app.edit();
}
delta = getCurrentTime() - start;
```

The JavaScript version of OTM and tracematches are aspect language extensions of AspectScript and AspectJ respectively. Figures 6.10 and 6.11 show the increment of the overhead of OTM and tracematches over their aspect languages. The overhead of OTM is evidently less than tracematches, and the tracematch overhead quickly increases when the pattern is longer (which may be due to the index scheme used in long patterns [Bodden, 2012]). In addition,

[1] http://www.sable.mcgill.ca/abc/

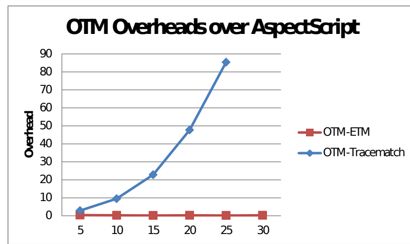

Figure 6.10: Overhead of OTM with two different matching semantics over AspectScript.

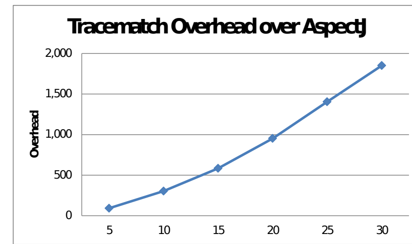

Figure 6.11: Overhead of tracematches over AspectJ.

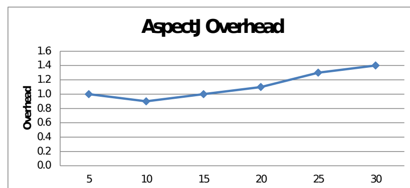

Figure 6.12: Overhead of AspectScript over JavaScript.

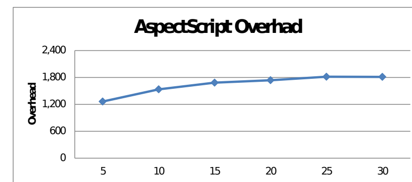

Figure 6.13: Overhead of AspectJ over Java.

Figure 6.10 that shows the choice of the semantics of matching process strongly affects the performance of OTM: the performance of ETM matching semantics is quite similar to an aspect of AspectScript; instead, the stateful aspect that uses the matching semantics of tracematches differs from the aspect implementation in an exponential manner.

Although figures 6.10 and 6.11 show a less overhead in OTM, these results do not mean that the JavaScript version of OTM has less overhead than tracematches for Java. Figures 6.12 and 6.13 shows the overhead of AspectScript is significantly greater than AspectJ. The AspectJ performance is very similar to Java (average of 1.3) instead of the AspectScript performance (average of 1,800).

Discussion. The current implementation of AspectScript is very slow compared to AspectJ. This so because AspectScript observes every join point of the program execution to know whether an aspect matches or not; meaning that AspectScript behaves similar to a monitoring system. This behavior helps to decrease the *new* overhead introduced by the implementation of OTM for JavaScript because our proposal is monitor system as well. It is important to mention that the choice of the matching process impacts on the OTM overhead. Instead, AspectJ does not observe every join point, meaning that tracematches has to introduce a significant new overhead to be implemented. In conclusion, OTM has a good performance over AspectScript because of the

current implementation of the aspect language. However, if OTM is implemented as an AspectJ extension, the OTM overhead would be significantly greater than tracematches.

6.7 Design Discussion

Every design of a system is motivated by a number of reasons. In this section, we discuss the reasons that have been considered in the design of OTM. In particular, we discuss our motivations to:

- propose a new algorithm to match program execution traces,
- choose the points of openness of OTM.

6.7.1 A New Algorithm to Match Traces

Different algorithms have been proposed to carry out the matching process. The simplest and naive matching process algorithm uses a database with the information of all join points occurred and queries to this database; the queries are really descriptions of patterns. Although this algorithm is very flexible to match traces, the definition of the pattern is completely tangled with the matching process semantics because there is no a separation between "what it must be matched" and "how it must be matched". In addition, this algorithm is impractical because the information of all join points has to be stored in the database and for every new join point the whole pattern is checked to know if it matches. As mentioned in Section 6.3.1, there are several algorithms for the matching process, however, the semantics of these algorithms are poorly flexible (see Chapter 3 for more details).

In an innovative manner, we use MatcherCells [Leger and Tanter, 2012], a self-replication algorithm to flexibly match traces. Self-replication algorithms are useful to define a wide range of semantics using the composition of small and simples rules. Therefore, the MatcherCells algorithm allows developers to flexibly define the matching process semantics. Like current matching process algorithms, this algorithm stores the information of a few join points and does not entirely check a pattern at each new join point. Unlike current algorithms, this algorithm does not impose restrictions on pattern definitions (*e.g.* tracematch patterns are only defined through regular expressions because of the use of a deterministic finite automaton to match).

6.7.2 Openness Points

A stateful aspect supports multiple matches of a pattern and simultaneous advice executions. These two features are defining characteristics of a stateful aspect. The well-chosen points to be open is focused on controlling both features through the customization of the matching and advising processes of a stateful aspect.

In OTM, it is possible to completely change the matching and advising processes of a stateful aspect. Although one may think the matching process must completely be coded every time that this process is customized, it is not really true because small pieces of code (*i.e.* rules) can be composed in a modular manner; in most cases, it is only necessary to code the particular behavior of a matching process. In the description of the advising process presented here, it must be completely coded every time that a new and different customization is needed. Initially, we explored the use of a rule-like technique to customize this process. For example, the designator rule and the rule composition that filter some match cells before executing the advices are:

```
var filterMatchCells = function(rule) {
  return function(matchCells,jp,advice) {
    var nextMatchCells = //filter match cells
    return rule(nextMatchCells,jp,advice);
} };

var selectiveAdviceExecutions = filterMatchCells(executeAdvices);
```

Conversely to the use of rules in a the definition of a matching process, the rules for advising are used at the end in order to delay advice executions. Based on our experience with OTM, the idea of using rules for the advising process was discarded because the advising process behavior is not so complex to divide it into small pieces of code.

6.8 Summary

This chapter presented OTM, a model of an open implementation of a trace-based mechanism that allows developers to flexibly customize the semantics of stateful aspects. Therefore, the limitations caused by fixed stateful aspects are addressed in OTM. As OTM follows the ETM guidelines and uses the MatcherCells algorithm, patterns, advices, and customizations are functions, bringing all benefits of the base language to express them (in particular, higher-order functions). The flexibility provided by the OTM model allows developers to define new semantic variants (Section 6.4) and emulate the semantics of existing trace-based mechanisms (Section 6.6). In

addition, this chapter presented a concrete and practical implementation of OTM for JavaScript with a typed description.

This chapter, together with the previous one, described the main contribution of our thesis: a model of an expressive and open trace-based mechanism. To the best of our knowledge, OTM is the first trace-based mechanism that supports the customizations of the core semantics of stateful aspects. The following two chapters describe different case studies of OTM.

Other examples. Apart from the examples presented here, the OTM Web site, http://pleiad.cl/otm, contains other running examples.

Part II: Case Studies

Chapter 7

Modularly and Flexibly Control Causality on the Web[1]

This chapter presents a case study of OTM: WeCa, a practical library to control causality issues on Web applications in a modular and flexible manner. In this case study, Web applications are context-aware systems that adapt their behavior when certain causal relations (contexts) are detected.

Ajax has allowed JavaScript programmers to create interactive, collaborative, and user-centered Web applications, known as Web 2.0 Applications. These Web applications behave as distributed systems because processors are user machines that are used to send and receive messages between one another. Unsurprisingly, these applications have to address the same causality issues present in distributed systems like the need *a)* to control the causality between messages sent and responses received and *b)* to react to distributed causal relations. JavaScript programmers overcome these issues using rudimentary and alternative techniques that largely ignore the distributed computing theory. In addition, these techniques are not very flexible and need to intrusively modify these Web applications. In this chapter, we study how causality issues affect these applications and present WeCa, a practical library that allows for modular and flexible control over these causality issues in Web applications. In contrast to current proposals, WeCa is based on (stateful) aspects, message ordering strategies, and vector clocks. We illustrate WeCa in action with several practical examples from the realm of Web applications. For instance, we analyze the flow of information in Web applications like Twitter using WeCa.

[1]The content of this chapter are based on our technical report "Modular and Flexible Causality Control on the Web" [Leger et al., 2011b]. This work is currently under revision.

7.1 Introduction

There is a strong trend towards the use of Web 2.0 Applications (WebApps for now) [O'Reilly, 2005], interactive, collaborative, and user-centered Web applications, like Twitter and Facebook. For the development of these applications, the JavaScript language and Ajax technologies [Garrett, 2005] are widely used. This is so because JavaScript, a dynamic prototype-based language with higher-order functions, is supported by most modern browsers and Ajax technologies allow WebApps to send and receive messages from a server or other applications[1] asynchronously. The latter feature converts WebApps into distributed systems because processors are user machines that are used to send and receive messages between one another. As a consequence, these applications have to address the causality issues of distributed systems like the need *a)* to control the causality between messages sent and responses received [Murty and Garg, 1997] and *b)* to react to distributed causal relations [Lamport, 1978].

The need to control the causality between messages sent and responses received arises when a WebApp retrieves various server responses in an arbitrary order. Instead, the need to react to distributed causal relations arises, for example, when it is necessary to analyze the flow of information at runtime that occurs among WebApps [Benevenuto et al., 2009]. Surprisingly, there is not much technical support that addresses the previous issues and even less modularly and flexibly. The current proposals [Farkas, 2011, Rico, 2011, Wikipedia, 2011] allow JavaScript programmers to overcome these issues using rudimentary techniques like explicit function wrappers and postmortem techniques like dynamic graphs. In addition, these techniques largely ignore the distributed computing theory and/or their inflexible uses end up scattered throughout many places in the code of a WebApp entangled with other concerns. In this paper, we present WeCa, a practical library that allows for modular and flexible control over causality on the Web.

In contrast to current proposals, WeCa is based on Aspect-Oriented Programming (AOP) [Kiczales et al., 1997b] and distributed computing concepts [Garg, 2002]. In particular, WeCa uses stateful aspects 3, Lamport's vector clocks [Lamport, 1978, Mattern, 1989] and message ordering strategies [Murty and Garg, 1997]. Our proposal allows for modular and flexible definition of *a)* message ordering strategies that control the causality between Ajax requests and server responses and *b)* monitors that react to distributed causal relations. Currently, the WeCa implementation uses AspectScript 4, an aspect-oriented extension of JavaScript, to

[1] Using a server as intermediary.

define aspects that enforce message ordering strategies in WebApps. In addition, the WeCa implementation uses OTM 6, an Open Trace-based Mechanism like trace-based mechanisms [Allan et al., 2005] for JavaScript, to define stateful aspects that react to distributed causal relations in WebApps.

The rest of this chapter is organized as follows. Section 7.2 presents and illustrates some causality issues on the Web through different WebApps. Section 7.3 introduces key concepts on which WeCa is based: (stateful) aspects, message ordering strategies, and vector clocks. Section 7.4 introduces WeCa, which combines the aforementioned concepts to address modularly and flexibly causality issues on the Web. Section 7.5 presents how our proposal addresses the causality issues described in Section 7.2. Section 7.6 presents different evaluations of WeCa. Section 7.7 discusses proposals about causality on the Web. Finally, Section 7.8 summaries.

Note. WeCa, along with the examples presented in this paper, is available online at http://pleiad.cl/weca.

7.2 Ajax & Web 2.0 Applications

Ajax [Garrett, 2005], a shorthand for *Asynchronous JavaScript and XML*, is a group of interrelated Web technologies used on the client-side to create interactive Web applications. Using Ajax, Web applications can send and retrieve data from a server or other applications asynchronously without reloading the current Web page. The following piece of code written in JavaScript shows a simple request to a server using Ajax:

```
var request = new XMLHttpRequest();
request.open("GET","server.com");
request.onreadystatechange = function() {
  if (this.readyState == 4)
    updateWebPage(this.responseText);
}
request.send(parameters);
```

The request object represents the *Ajax request* to the server. The open method configures the communication with the server, and the send method sets the parameters of the Ajax request and sends it. The server responds with an arbitrary delay. When the server responds, the onreadystatechange method is executed[1]. The responseText instance variable of request contains

[1]This method is actually executed every time that the value of readyState changes. We only want to take an action when the server response is available for the application, *i.e.* when readystate takes the value 4.

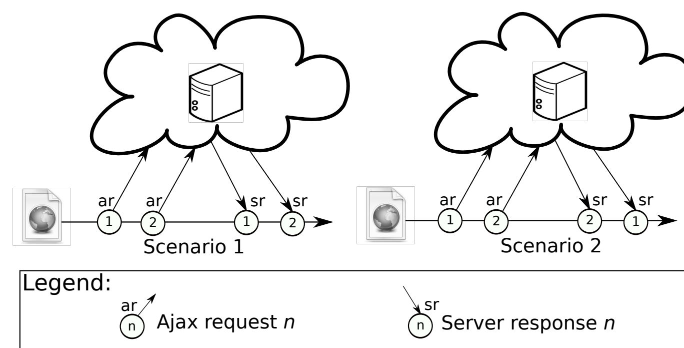

Figure 7.1: Two possible scenarios when a Web application sends two Ajax requests.

the *server response*, which can be used, for example, to update the Web page.

Using Ajax technologies, JavaScript programmers create interactive, collaborative, and user-centered Web Applications, known as Web 2.0 Applications [O'Reilly, 2005]. For example, Housing Maps, an application for finding a house for sale, is created from server responses that come from different sources. Another example is Twitter, an application for social network and microblogging, which allows users to send and receive messages. When a WebApp uses Ajax technologies, some needs arise, such as the need *a)* to control the causality between messages and responses received [Murty and Garg, 1997] and *b)* to react to distributed causal relations [Lamport, 1978].

7.2.1 Controlling Message Causality

A WebApp can send several Ajax requests to different servers. However, the application can retrieve and process the server responses in an arbitrary order; meaning that this application may behave nondeterministically due to the lack of control of the causality between Ajax requests and server responses. For example, Figure 7.1 shows that if a WebApp sends two Ajax requests to a server, two scenarios are possible: the server returns server response 1 followed by server response 2 or vice versa. Depending on the expected behavior of the WebApp, different strategies to control the message causality can be used. We now present three WebApps that require three different strategies:

A FIFO strategy for a mashup application. A mashup application is created from the combination of information retrieved from different servers, *e.g.* Housing Maps. Programmers

have to overcome the issue of creating an incorrect Web page due to an arbitrary (and unexpected) order of server responses. For instance, consider a Web page that is created with two Ajax requests. If the second server response is processed before the first server response, the Web page is created incorrectly. A *FIFO* strategy, which processes the server responses in the same order as the Ajax requests are sent, ensures that the Web page is always created correctly.

A Discard Late strategy for the visualization of news threads. WebApps typically update data from servers using Ajax (*e.g.* news, thread posts, and streaming medias). Programmers have to overcome the issue of processing obsolete data due to late server responses. For example, consider a Web page of news threads that shows the last updates about several news. These updates are retrieved from server responses. If some server responses are retrieved and displayed in an incorrect order, the reader could misunderstand these last updates of a news thread. A *Discard Late* strategy that discards the late (and obsolete) server responses always shows really the last updates of a news thread. Therefore, the last updates of a news thread are exposed in correct order for the reader[1].

A Discard Early strategy for the visualization of thread posts. Another issue in the visualization arises due to the processing of early server responses. For example, consider a forum Web page that is frequently updated with threads from a server using Ajax. Every thread is shown with its title and its first posts, and the reader can click on a button "read more" to see the whole discussion of a given thread. Every post that is below the thread title is loaded dynamically using an independent Ajax request[2]. If server responses, which contain the posts, are retrieved and displayed in an incorrect order, the beginning of the discussion of the thread could be misunderstood. A solution could be to wait for all late posts of every thread. However, this solution could seriously delay the updating of the forum Web page. A better solution is to discard the earlier posts in order to correctly show the first posts of the thread. The discarded posts can be fetched later when the reader clicks the given "read more" button. A *Discard Early* strategy, which discards early server responses, can ensure the beginning of the discussion of every thread is read in a correct order.

[1] At the WeCa website, http://pleiad.cl/weca, we use the Discard Late to show a visually more extractive example, which shows a streaming video in a correct flow.

[2] A thread commonly contains a (large) variable number of posts.

7.2.2 Reacting to Distributed Causal Relations

Social network applications like Twitter and Facebook are another kind of WebApp. Nowadays, these applications are widely used, making the analysis of their flow of information an active research topic [Benevenuto et al., 2009, Jiang et al., 2010, Wilson et al., 2009, Wasserman and Faust, 1994]. This flow of information is analyzed through the messages sent and received between users of these applications. Such an analysis is complex due to the need to observe and react to distributed causal relations that occur among user interactions. As an example of the analysis of the flow of information in WebApps, consider the calculation of the popularity of user *tweets*[1] in Twitter:

Tweet popularity. This feature in Twitter [Mashable, 2010] allows a user to know the popularity of every tweet published by him or her, which is measured by the number of *retweets*[2] of direct and indirect followers. For example, Figure 7.2 shows four Tweeter users: Toti, Dacha, Kuky, and Paul. Toti follows Dacha and Dacha follows Paul; Kuky follows nobody and nobody follows Kuky. The figure shows that Paul publishes a tweet and Dacha receives this tweet and retweets it. The figure also shows that Kuky publishes a tweet and nobody receives it. Based on the popularity measurement, the popularity of Paul's tweet is 1 and that of Kuky is 0. Although Kuky and Paul would have published the same tweet[3], the popularity of Kuky's tweet is 0 because his tweet did not cause any retweet. An analysis based on the distributed causal relations observed between tweets and retweets can determine how many users retweet a given tweet. For example, Paul's tweet caused Dacha's retweet.

7.2.3 State of the Practice

State of the practice provides JavaScript practitioners have at their disposal a number of JavaScript libraries and postmortem tools to solve the kind of issues we described previously.

Some lightweight JavaScript libraries [Farkas, 2011, Rico, 2011] are used to control the causality between Ajax requests and server responses. However, these libraries do not modularly control the message causality because programmers need to explicitly wrap every Ajax request in order

[1] Messages posted via Twitter containing 140 characters or fewer. This word is indistinctly used as a noun and a verb.

[2] Tweets reposted by another user. This word is indistinctly used as a noun and a verb.

[3] Two tweets are equal if both contain the same string or have the same url associated. In this paper, we say two tweets are equal if they contain the same string.

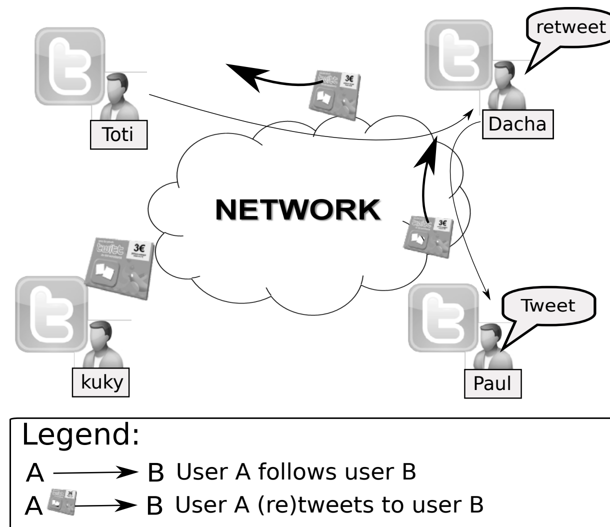

Figure 7.2: Retweeting a tweet.

to use these libraries. In addition, these libraries only provide a limited set of strategies that are not customizable. For example, the following piece of code that uses the AjaxManager library [Farkas, 2011] enforces a server response to follow the FIFO strategy. The FIFO strategy is enabled with a boolean value (queue). In addition, the Ajax request associated to the server response that must follow the FIFO strategy ("myFIFO") has to be rewritten manually.

```
//deploying the FIFO strategy
var fifo = managerAjax.create({
  queue: true, //this property means that FIFO is enabled
});

//An Ajax request that follows the previous FIFO strategy
fifo.add({
  success: function(serverResponse) {
    updateWebPage(serverResponse);
  },
  url: "server.com"
});
```

Postmortem tools based on dynamic graphs [Wikipedia, 2011] are used to observe distributed causal relations. As these tools are postmortem, they cannot be used to react to distributed causal relations at runtime.

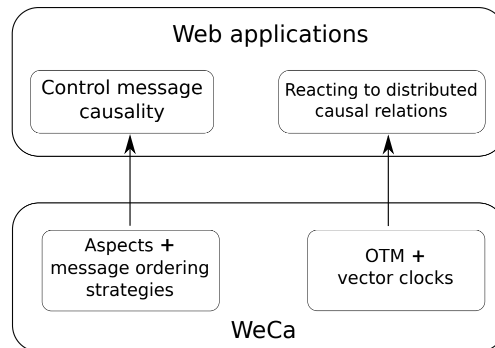

Figure 7.3: The solution proposed by WeCa.

7.2.4 WeCa Overview

WeCa is a practical library that allows for modular and flexible control over causality as required in the examples presented previously (see Figure 7.3). By *modular* we mean that the control over causality is addressed in a separate module at the code level, and by *flexible* we mean that the definition of the control over causality is customizable using the power of the base language. The runtime of WeCa observes every Ajax request with its server response to enforce a certain strategy to deal with server responses (*e.g.* Discard Early). In addition, this runtime observes every message sent and received between these applications to detect distributed causal relations at runtime in interactive communication of WebApps (*e.g.* the popularity of a tweet).

Figure 7.4 shows how WeCa works in a nutshell. Every WebApp has a WeCa runtime instance. Each instance can use an *aspect* [Kiczales et al., 1997b] to match every server response, and using *message ordering strategies* [Murty and Garg, 1997], the aspect enforces that (some) server responses follow a certain message ordering strategy. In addition, each instance can use *stateful aspects* [Douence et al., 2005] to match and react to distributed computations. These stateful aspects utilize *vector clocks* [Lamport, 1978, Mattern, 1989] to match and react to distributed computations that satisfy certain causal relations.

7.3 Distributed Computing

WeCa combines concepts from AOP and distributed computing [Garg, 2002]. In this section, we summarize the distributed computing concepts in order to clarify our proposal.

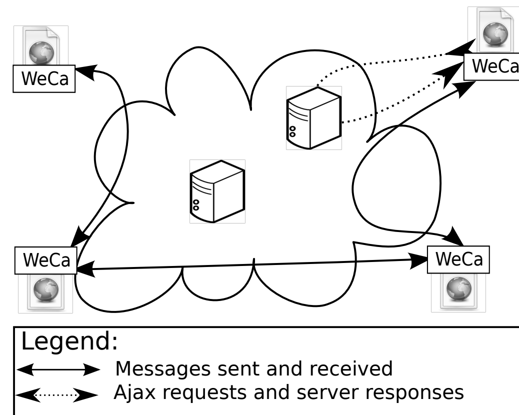

Figure 7.4: WeCa overview.

7.3.1 Distributed Computing

In distributed systems, processes communicate with each other using messages that are sent over the network. The sending and receiving of these messages as well as beginnings and ends of executions of functions are considered events of a distributed computation. Whereas we can observe a total order among events of a single process, there is no global clock (or perfectly synchronized local clocks) that allows us to observe a total order between events of different processes. Nevertheless, it is possible to observe a *partial order* between events if we use the *happened before* model proposed by Lamport [Lamport, 1978].

Aspects for distributed computations. In AOP, join points are execution points which correspond to events. Traditionally (stateful) aspects react to join points of a single process. To react to *distributed* traces of join points, they have to match join points of different processes; and to react to distributed causal relations, (stateful) aspects have to consider the causal relations among these join points. In this section, we explain distributed computing concepts using AOP terminology in order to understand how aspects can react to distributed causal relations. For example, we explain the happened before model using join points instead of events:

$$jp_1 \rightarrow jp_2$$

The happened before model defines a relation between two join points noted with an arrow. The arrow indicates a join point jp_1 causes another join point jp_2, meaning there is a causal

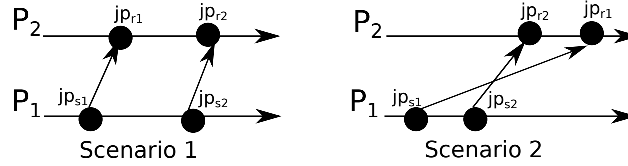

Figure 7.5: Two possible distributed computations for the same input.

relation from jp_1 to jp_2. For example, Figure 7.5 shows in both scenarios that the join point jp_{s1} causes the join points jp_{s2}, jp_{r1}, and jp_{r2} because the join point jp_{s1} *happened* before.

The happened before model makes it possible to address causality issues such as the need *a)* to control the causality between messages sent and responses received and *b)* to react to distributed casual relations.

7.3.1.1 Controlling Message Causality

Distributed systems are difficult to test because of their nondeterministic nature, that is, these systems may exhibit multiple behaviors for the same input. This nondeterminism is caused by an arbitrary ordering of messages received by processes in different distributed computations. Figure 7.5 shows that a process P_1 sends two messages, represented by jp_{s1} and jp_{s2} join points, to a process P_2. P_2 can receive these messages, represented by jp_{r1} and jp_{r2} join points, in two different orders arbitrarily.

Fortunately, message ordering strategies [Murty and Garg, 1997] can be used to control the message causality. We now detail some of these strategies:

FIFO. Any two messages from a process P_i to P_j are received in the same order as they are sent. In a formal manner, let jp_{s1} and jp_{s2} be any two join points sent and $jp_{s1} \rightarrow jp_{s2}$, then the jp_{r2} join point cannot be observed by any process before the jp_{r1} join point.

$$jp_{s1} \rightarrow jp_{s2} \Rightarrow \neg(jp_{r2} \rightarrow jp_{r1})$$

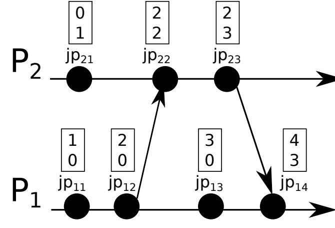

Figure 7.6: Join points of a distributed computation tagged with vector clocks.

Discard Late. Any two messages from a process P_i to P_j are received in the same order as they are sent or only the most recent message is received. In a formal manner, this strategy is just an extension of FIFO: if jp_{r2} is observed first, jp_{r1} cannot be observed by any process.

$$jp_{s1} \rightarrow jp_{s2} \Rightarrow \neg(jp_{r2} \rightarrow jp_{r1}) \vee \neg jp_{r1}$$

Discard Early. Any two messages from a process P_i to P_j are received in the same order as they are sent or only the oldest message is received. In a formal manner, this strategy is just an extension of FIFO: if jp_{r1} is observed first, jp_{r2} cannot be observed by any process.

$$jp_{s1} \rightarrow jp_{s2} \Rightarrow \neg(jp_{r2} \rightarrow jp_{r1}) \vee \neg jp_{r2}$$

7.3.1.2 Reacting to Distributed Causal Relations

To react to distributed causal relations, stateful aspects have to match distributed traces of join points and to consider causal relations between these join points. Sadly, the absence of a total order among these join points does not allow stateful aspects to consider causal relations.

Fortunately, the algorithm of Lamport's vector clocks [Lamport, 1978, Mattern, 1989] can allow stateful aspects to observe a partial order. We now explain a straightforward adaptation of this algorithm to allow stateful aspects to consider causal relations between join points of distributed traces.

To support vector clocks, every join point jp is tagged with an array of counters of size n, where n is the number of processes. This array represents a vector clock V, which is accessed by $V(jp)$, and is filled according to the following algorithm:

1. Initially, $V_i[k] = 0$ for $k = 1, ..., n$.

2. On each join point that does not represent the sending or receiving of a message, process P_i increases V_i as follows: $V_i[i] = V_i[i] + 1$.

3. On each join point jp that represents the sending of a message m, process P_i updates V_i as in *2)* and attaches the new vector clock to jp.

4. On each join point jp that represents the receiving of a message m, process P_i increases V_i as in *2)*. Then, P_i updates its current V_i as follows: $V_i = sup\{V_i, V(jp)\}$.

If all join points are tagged according to this algorithm, it is possible to define rules that verify if two join points satisfy distributed causal relations such as *causal* and *concurrent*:

Causal rule. This rule allows us to verify if a join point jp_1 *caused* another join point jp_2. For example, Figure 7.6 shows that jp_{12} caused jp_{22} and jp_{23}. In a formal manner, the causal rule is defined by the following relation:

$$jp_1 \rightarrow jp_2 \Longleftrightarrow V(jp_1) < V(jp_2) \tag{7.1}$$

Where:

$$V(jp_1) < V(jp_2) \Longleftrightarrow \forall k[V(jp_1)[k] \leq V(jp_2)[k]] \wedge \exists k'[V(jp_1)[k'] < V(jp_2)[k']]$$

Concurrent rule. This rule allows us to verify that the jp_1 and jp_2 join points are *concurrent*: jp_1 does not cause jp_2 and vice versa. For example, Figure 7.6 shows that jp_{22} and jp_{13} are concurrent. In a formal manner, the concurrent rule is defined by the following relation:

$$jp_1 \parallel jp_2 \Longleftrightarrow \neg(jp_1 \rightarrow jp_2) \wedge \neg(jp_2 \rightarrow jp_1) \tag{7.2}$$

Contribution for WeCa. This section showed the necessary concepts to work on causality. The implementation of causality in distributed systems is a crosscutting concern because it is necessary to intercept, modify, and (possibly) postpone the evaluation of every join point. For example, the FIFO strategy postpones the evaluation of early join points. For this reason, the use of (stateful) aspects of AspectScript and OTM allows WeCa to modularize these causality concerns.

7.4 WeCa

This section describes WeCa, a practical library that allows for modular and flexible control over causality on the Web. This library combines stateful aspects, message ordering strategies, and vector clocks to allow JavaScript programmers to modularly and flexibly define *a)* strategies that control the causality between Ajax requests and server responses and *b)* monitors that react to distributed causal relations.

Figure 7.4 shows that every WebApp has a WeCa runtime instance. Every instance observes join points generated on the application and other connected applications. If a programmer defines a message ordering strategy, the WeCa runtime enforces that server responses follow this strategy. In addition, if the programmer expresses a distributed causal relation pattern, the WeCa runtime frequently observes the distributed trace of join points to react whenever this trace is matched by such a pattern. We now explain how WeCa enforces message ordering strategies and reacts to distributed causal relations.

7.4.1 Controlling Message Causality

As mentioned in Section 7.2.1, a WebApp can send several Ajax requests to servers. However, this application can retrieve and process the server responses in an arbitrary order, therefore, this application behaves nondeterministically. Unlike that presented in Section 7.3.1.1, the problem does not arise between the messages sent by different processes, the problem arises when a WebApp retrieves server responses (Figure 7.1). Fortunately, message ordering strategies can also be used to control the causality between Ajax requests and server responses, and AspectScript aspects allow programmers to flexibly and modularly define these strategies. Using aspects and message ordering strategies, WeCa can be used, for example, to enforce server responses retrieved from a server following the Discard Early strategy (Section 7.2.1), which is provided as a library function in WeCa:

```
WeCa.deployStrategy(discardEarly,function(jp) {
  var request = jp.target;
  return request.url == "http://server.com";
});
```

In this piece of code, the deployStrategy method deploys the Discard Early strategy, which enforces server responses that come from http://server.com to follow this strategy. The two parameters of the deployStrategy method are JavaScript functions: the first represents the strategy and the second represents its scope. As message ordering strategies are JavaScript functions, the

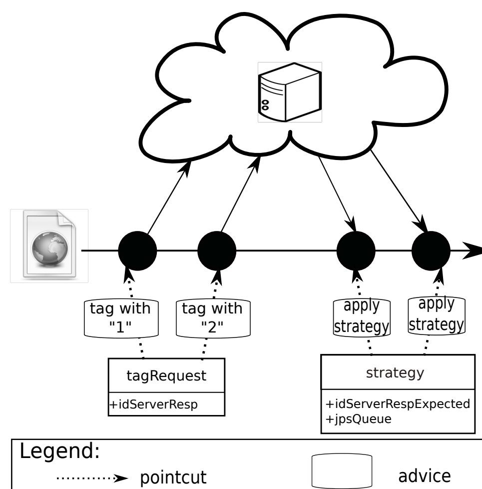

Figure 7.7: Two aspects to apply a message ordering strategy.

programmer can use the full power of the base language, in particular higher-order functions, to define a strategy. The second function of deployStrategy is parametrized by the join point that represents the execution of the onreadystatechange method, which processes the current server response. This function returns true if the server response must follow the strategy.

We now explain how WeCa implements and allows programmers to flexibly deploy these strategies.

Implementation details. Figure 7.7 shows how WeCa only needs two aspects to support message ordering strategies. The first aspect, tagRequest, matches calls to the send method, which represents an Ajax request, of a request object. The piece of advice of this aspect tags the request object with a fresh and incremental identifier idServerResp that is generated to identify the associated server response. The second aspect, strategy, matches the executions of the onreadystatechange method, which processes the server response of a request object. The piece of advice of this aspect is actually the message ordering strategy. This advice is defined by the programmer, who can use the idServerRespExpected and jpsQueue aspect instance variables to implement the strategy. The idServerRespExpected binding identifies the next server response expected, and jpsQueue is a queue of join points that contains methods to execute join point proceeds. As an example, the implementation of the Discard Early strategy is:

```
var discardEarly = function(jp,env) {
```

```
  var request = jp.target;
  if (this.idServerRespExpected >= request.idServerResp) {
    jp.proceed();
    this.idServerRespExpected = request.idServerResp + 1;
} };
```

The jp join point represents the execution of the onreadystatechange method that processes the current server response. The proceed method is only executed if the server response is equal or older than the server response expected, *i.e.* if the server response is not early. Apart from the Discard Early strategy, WeCa provides other message ordering strategies as library functions.

7.4.2 Reacting to Distributed Casual Relations

As mentioned in Section 7.2.2 the analysis of the flow of information of social network applications is an active research topic. This flow of information is analyzed through the messages sent and received between users of these applications, meaning that it is necessary to observe and react to distributed causal relations given by these messages.

Fortunately, Lamport's vector clocks can also be used to observe distributed causal relations in WebApps, and OTM stateful aspects allow programmers to react to these distributed causal relations in a flexible and modular manner. Using stateful aspects and vector clocks, WeCa can be used, for example, to deploy a stateful aspect that shows a message every time a follower retweets some tweet:

```
var callTweet = function(jp,env) {
  if (jp.isCall() && jp.fun == tweet) {
    var tweet = jp.args[0]; //1st argument to tweet
    return env.bind("tweet",tweet);
  }
  return false;
};

var notifyRetweet = function(jp,env) {
  if (jp.isCustom("notification-call") && jp.fun == retweet) {
    var retweet = jp.args[0]; //1st argument to retweet
    return retweet == env.tweet? env: false;
  }
  return false;
};
//necessary (sub)patterns to create the stateful aspect

var sAspNotifyRetweet = {
```

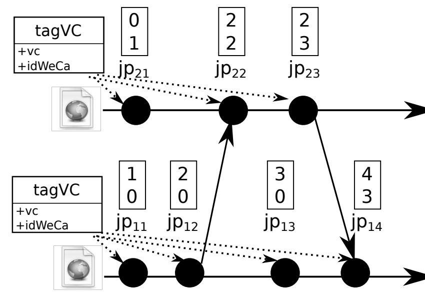

Figure 7.8: The tagVC aspect that tags every join points with a vector clock.

```
  pattern: causalSeq(callTweet, notifyRetweet),
  advice: function(jp, env){
    alert("One of your followers retweeted:"+env.tweet);
  },
  kind: AFTER
};

WeCa.OTM.deploy(sAspNotifyRetweet); //deployment
```

The sAspNotifyRetweet stateful aspect matches every time a tweet causes a retweet. The callTweet pattern returns an environment that contains the tweet, passed as parameter to the tweet function, if this pattern is matched with the call to tweet. The notifyRetweet pattern is matched with a *notification call join point*, a call join point that occurs in another connected application, of the call to the retweet function if both the tweet and retweet are equal. Finally, we use the causalSeq pattern designator to return a pattern that is used to match the two previous (sub)patterns if both satisfy the causal rule (Section 7.3.1.2).

We now explain how WeCa implements notification join points and patterns that are used to match distributed causal relations.

Implementation details. WeCa only needs two aspects to react to distributed causal relations. Figure 7.8 shows the first aspect, tagVC, which tags every join point with the corresponding vector clock. Figure 7.9 shows the second aspect, notify, which notifies other WeCa runtime instances for every join point generated on the computation of a connected WebApp. When a WeCa runtime instance receives the information about a remote join point, this instance generates a notification

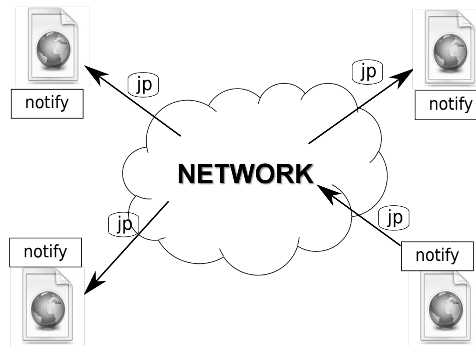

Figure 7.9: The notify aspect that notifies every join point to other applications.

join point of the corresponding kind[1], which can be matched by any stateful aspect. The context information of a notification join point includes the join point generated plus the WeCa instance identifier.

As we have shown in the previous piece of code, to react to distributed causal relations, WeCa uses OTM to define stateful aspects that react when they match distributed causal relations. In OTM, to define patterns that represent the match of distributed causal relations using the causal or concurrent rule (Section 7.3.1.2), it is necessary to have pattern designators like causalSeq to create patterns that consider causal relations. To achieve this, we only have to extend the seq pattern designator (Chapter 5, Section 5.4.1):

```
var seq = function(left, right) {
  return function(jp, env) {
    var result = left(jp, env);
    if (isEnv(result))
      return [function(jpNext, env) {
        if ( vectorClockCondition(jp,jpNext) )
          return right(jpNext, env);
        return false;}, result];
    return false;
} };
```

In Section 5.4.1, the seq pattern designator returns a pattern that if left matches this pattern directly returns right. Instead, this new seq returns a pattern that if left matches this pattern returns a function that first verifies if the jp and jpNext join points satisfy some causal relation before evaluating right. The relation causal is verified through the vectorClockCondition function.

[1] Like Ptolemy [Rajan and Leavens, 2008], AspectScript allows developers to explicitly trigger customized join points.

Based on the piece of code above, we can create pattern designators like causalSeq, concSeq, or seq using higher-order functions:

```
var makeSeq = function(vectorClockCondition) {
  return function seq(left, right) {
    //the piece of code above
} } };

var causalSeq = makeSeq(caused);
var concSeq = makeSeq(areConcurrent);
var seq = makeSeq(function(jp,jpNext) {return true;});
```

The caused function verifies whether jp causes jpNext or not, and the areConcurrent function verifies whether both join points are concurrent or not. The functions caused and areConcurrent are developed according to the relations 7.1 and 7.2 described in Section 7.3.1.2. Finally, if we need a seq pattern designator that does not consider any causal relation, the makeSeq function is called with a function that always returns true. These pattern designators are provided as library functions in WeCa.

7.4.3 Summary

WeCa uses only four aspects in order to work. These four aspects extend AspectScript and OTM to address causality issues on the Web. Two aspects are used to support the definition of flexible message ordering strategies, and two aspects are used to react to distributed causal relations. Using the power of the base language, developers can define message ordering strategies and patterns that are used to match distributed causal relations.

7.5 Revisiting Web 2.0 Applications

In Section 7.2, we have shown examples of how causality issues affect WebApps. This section presents how to address these issues using WeCa. First, we use WeCa to define appropriate message ordering strategies that address the different needs of WebApps described in Section 7.2.1. Second, we use WeCa to define a stateful aspect that determines the popularity of every tweet of a user (Section 7.2.2).

7.5.1 Controlling Message Causality

A WebApp can send several Ajax requests to different servers. However, this application can retrieve and process server responses in an arbitrary order. In Section 7.2.1, we presented three

WebApps that needed three different message ordering strategies. This section presents and describes these strategies, which are provided as library functions in WeCa. As mentioned in Section 7.4.1, these strategies are really advice of an aspect. In addition, we extend one of these strategies to show the usefulness of the flexibility provided by WeCa.

A FIFO strategy for a mashup application. Programmers have to overcome the issue of creating an incorrect Web page due to an arbitrary order of server responses. The FIFO message ordering strategy, which processes the server responses in the same order as Ajax requests are sent, can ensure the Web page is always created correctly. This strategy postpones the use of an early server response until it is the expected one. We now present the function that describes the FIFO strategy:

```
var FIFO = function(jp,env){
  var request = jp.target;
  if (this.idServerRespExpected != request.idServerResp){
    this.jpsQueue.push(jp);
  }
  else{
    jp.proceed();
    this.idServerRespExpected = request.idServerResp + 1;

    //executing jp proceeds that can be executed now
    this.idServerRespExpected =
      this.jpsQueue.execRecEarlyJPs(this.idServerRespExpected);
  }
}

WeCa.deployStrategy(FIFO);
```

As mentioned in Section 7.4.1, the `jp` join point represents the execution of the `onreadystatechange` function. If this server response is not the expected one, `jp` is added to a queue and its proceed execution is postponed. Instead, if this server response is the expected one, the join point proceed is executed and the queue tries executing proceeds of join points stored because some of them can be the expected ones now. This is so because if a join point proceed is executed inside of the queue, it can permit the execution of another join point proceed. As the scope is not specified in this deployment, the scope is global for this strategy, meaning that all server responses follow this strategy.

A Discard Late strategy for the visualization of news threads. In the visualization of news threads, programmers have to overcome the issue of using obsolete data due to late server responses. A Discard Late strategy, which discards late server responses, always show the last updates of a news thread. We now present the function that describes the Discard Late strategy:

```
var discardLate = function(jp,env) {
  var request = jp.target;
  if (this.idServerRespExpected <= request.idServerResp) {
    jp.proceed();
    this.idServerRespExpected = request.idServerResp + 1;
  }
};

WeCa.deployStrategy(discardLate,function(jp) {
  var request = jp.target;
  return request.url == "http://news.com";
});
```

If the server response is not late, the proceed of the jp join point is executed. If the server response is late, the join point proceed is not executed, meaning that the execution of the onreadystatechange method that processes this server response is discarded. Given the scope of the deployment, server responses that only come from http://news.com must follow this strategy.

A Discard Early strategy for the visualization of thread posts. In the visualization, programmers also have to overcome the issue of using early data due to early server responses. Remembering the forum Web page, which updates posts of the last threads with Ajax requests. If some server responses are retrieved and displayed in an incorrect order, the beginning of the discussion of a thread may be misunderstood. A *Discard Early* strategy, which discards early server responses, can ensure the reader better understands the beginning of the discussion of every thread. In Section 7.4.1, we presented the Discard Early strategy; we now present an extension of this strategy, which executes a callback function when it discards a server response.

```
var deShowReadMore = discardEarlyCallback(function callback(jp,env) {
   //show the "read more" button
});

WeCa.deployStrategy(deShowReadMore,function (jp) {
  var request = jp.target;
```

```
  return request.url == "http://forum.com/threadlist";
});
```

We use the callback function to show a button "read more" when a server response is discarded. The piece of code below shows that the evaluation of the discardEarlyCallback function returns a function that uses the Discard Early strategy and executes a callback function, when a server response is discarded.

```
var discardEarlyCallback = function(callback) {
  return function(jp,env) {
   //invoke the discard early strategy

    var request = jp.target;

    //is it discarded?
    if (this.idServerRespExpected < request.idServerResp)
      callback(jp,env);
} };
```

7.5.2 Reacting to Distributed Causal Relations

The analysis of the flow of information of WebApps is analyzed through the messages sent and received among users of these applications. In this section, we use a stateful aspect to calculate the popularity of tweets in Twitter (Section 7.2.2):

Tweet popularity. This feature allows a user to know the popularity of its tweets, which is measured by the number of retweets. As Figure 7.2 shows, if a user publishes a tweet that is retweeted by a follower, the popularity of this tweet is increased by one. An analysis based on the distributed causal relations between tweets and retweets can determine how many users retweeted a tweet. The following stateful aspect counts the retweets of a tweet until the user clicks a button "popularity":

```
var callTweet = //... as Section 7.4.2

var retweetCount = function (jp,env){
  var result = notifyRetweet(jp,env); //notifyRetweet as Section 7.4.2
  if (isEnv(result))
    return env.counter == undefined?
      env.bind("counter",0): env.bind("counter",env.counter + 1);
  return false;
```

```
};

var callPopularity = function(jp,env) {
  return jp.isCall() && jp.fun == clickPopularityButton;
}
//necessary (sub)patterns to create the stateful aspect

var sAspTweetPopularity = {
  patterns: causalSeq(callTweet,starUntil(retweetCount,callPopularity)),
  advice: function(jp,env){
    addPopularity(env.tweet,env.counter);
  },
  kind: AFTER,
};

WeCa.OTM.deploy(sAspTweetPopularity);
```

The sAspTweetPopularity stateful aspect is only an extension of the stateful aspect shown in Section 7.4.2. This new stateful aspect counts the number of retweets of a tweet and adds this counter to the analyzed tweet. The pattern begins the matching when the callTweet (sub)pattern matches and continues with matchings of the retweetCount (sub)pattern until the user clicks the "popularity" button. Every time retweetCount matches, the counter of retweet is increased by one. The starUntil pattern designator, which is provided as a library function in OTM, returns a pattern that can match the first (sub)pattern several times until the second (sub)pattern is matched.

Although this stateful aspect works, it can only count the popularity of one tweet. A solution to count the popularity of every tweet could be to have a stateful aspect for every tweet. However, this solution is not very efficient because Twitter users commonly publish the same tweet several times[1]. A better solution would be that the sAspTweetPopularity stateful aspect has a new potential match of its pattern for every different tweet. As mentioned in Chapter 6, OTM allows developers to customize the matching process of a stateful aspect. To customize this process, the matching property is added to the stateful aspect definition. The matching process can be customized through the composition of a set of rules, where each rule is a function and defines a part of the semantics of this process:

```
var sAspTweetPopularity = {
  //pattern, advice, and kind properties remain the same
  matching: keepSeed(pattern)(differentTweets(killCreators(applyReaction)));
```

[1] http://holykaw.alltop.com/the-art-of-the-repeat-tweet.

```
};
```

The composition of rules used to define the matching process of sAspTweetPopularity means that this stateful aspect has a new potential match of the pattern for every different tweet (more details in Chapter 6). The differentTweets rule designator returns a rule that removes the cells that have not different tweets:

```
var differentTweets = function(rule) {
  return function (potentialMatches,jp) {
    var nextCells = rule(cells,jp); //eg. the applyReaction rule
    var currentTweets = getTweets(nextCells);
    var newCells = difference(nextPotentialMatches,potentialMatches);

     filteredNewCells = filter(function(newCell) {
      return some(function(tweet) {
         return newCell.env.tweet == tweet;
      },currentTweets);
    },newPotentialMatches);

    return difference(nextCells,filteredNewCells);
} };
```

Tweets of every computer are obtained from the environment of bindings that each cell contains. Each tweet of the new cells is compared to current tweets to know whether this tweet is new or not. Finally, the new cells with new tweets are only returned.

7.6 Evaluation

This section presents four evaluations of our proposal. First, we evaluate the modular use of WeCa. Second, we evaluate the effectiveness to control the order of server responses. In the third and fourth evaluation, we measure the WeCa overhead.

7.6.1 Modular Use

Ajax Video Player [Shahin, 2006] is a JavaScript application that allows a Web page to show streaming of videos. This application sends an Ajax request to a server which responds with a buffer of frames. When the associated server response is processed the application sends another Ajax request to retrieve the next buffer; this cycle is repeated. Sending an Ajax request at a time introduces significant interruptions during the display of a video. We extended this application

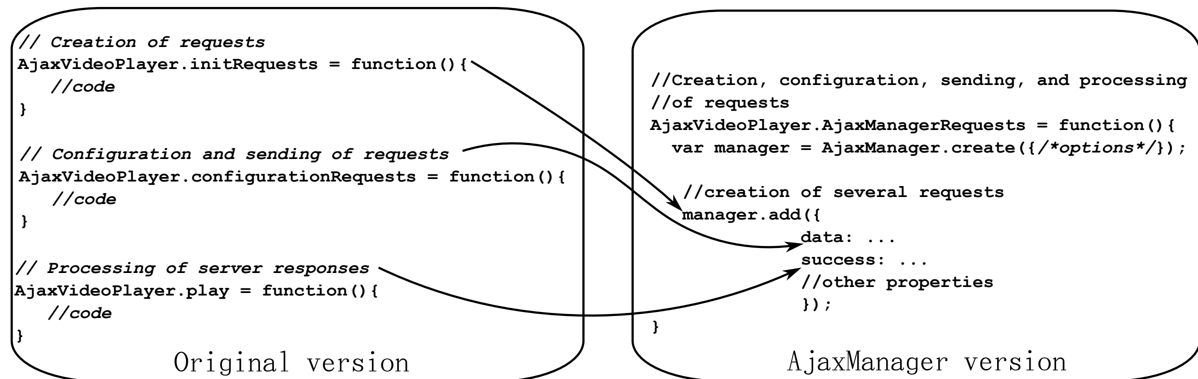

Figure 7.10: Modification of Ajax Video Player to support AjaxManager.

to send simultaneous Ajax requests in order to decrease these interruptions. Later, we use WeCa and AjaxManager [Farkas, 2011] to control the order of the server responses.

To work with WeCa, we only had to add the piece of code below, where the strategy variable is bound to a message ordering strategy:

```
WeCa.deployStrategy(strategy, function(jp) {
  return jp.target.url == "videoServer.php";
});
```

Figure 7.10 shows the necessary modifications in Ajax Video Player to support AjaxManager. We had to create an AjaxManager object (Section 7.2.3) and modify functions related to the creation, configuration, sending, and processing of Ajax Video Player requests in order to wrap them. As the figure also shows, different concerns related to Ajax requests must be mixed in only one piece of code to support AjaxManager. In conclusion, the difference between both tools is that the Ajax Video Player does not need to be modified to support WeCa.

7.6.2 Controlling Server Responses

As mentioned in the previous section, we extended Ajax Video Player to send simultaneous requests, which entail an unexpected order of server responses. Therefore, we use WeCa and AjaxManager to control the order of these responses.

We evaluate six different versions of Ajax Video Player: one request at a time, simultaneous requests, and four versions that use two different message ordering strategies of WeCa and AjaxManager. The application was tested with videos of different resolutions and lengths. The resolution of videos varies from 160x120 to 624x352 pixels and the length varies from 3 to 218

Versions/Evaluations	a) Interruptions	b) Wrong buffers	c) Discarded buffers
One request (original)	20.9%	0%	0%
Simultaneous requests	18.79%	18.29%	0%
WeCa Late	5.26%	0%	10.9%
WeCa FIFO	11.77%	0%	0%
AjaxManager Late	8.77%	0%	6.14%
AjaxManager FIFO	13.53%	7.39%	0%

Table 7.1: Evaluations of different versions of Ajax Video Player.

seconds (*i.e.* 03'38"). We measure the percentage of *a)* interruptions while a video is displaying, *b)* buffers that are incorrectly received (*i.e.* a buffer that is not the continuation of the previous one), and *c)* buffers that are discarded because they are received late. For this experiment, we say that an interruption is when a video is stopped until a new buffer is retrieved and processed.

As Table 7.1 shows, the version that sends one request at a time (original version) has more interruptions than other ones. When the original version is extended to support simultaneous requests, the interruptions decrease, but wrong buffers appear. The version that uses the Discard Late strategy of WeCa significantly decreases the interruptions, but some buffers are discarded. The version that uses the FIFO strategy of WeCa does not contain wrong and discarded buffers, but the interruptions are double of the previous version with WeCa. The AjaxManager versions behave similarly to WeCa, except for the FIFO strategy, which shows wrong buffers due to a bug in AjaxManager. To conclude, a version of Ajax Video Player that supports simultaneous requests improves the fluency of a streaming video; and WeCa or AjaxManager can be used to control the order of the buffers of frames.

7.6.3 WeCa Overhead

We ran tests of our library to evaluate the overhead of the control of message causality and reaction to distributed causal relations. For these tests, we used three machines: *a*, *b*, and *c*. The machine *a* is in Chile (Santiago) and is an Intel Core 2 Duo, 2.66 GHz with 2GB of RAM. The machines *b* and *c* are in USA (Newark) and are an Intel(r) Xeon (R), 2.27 GHz with 512MB of RAM. These machines are running on Ubuntu 10.04 with Firefox 12.

7.6.3.1 Overhead of Controlling Message Causality

We measured the overhead time of forcing each message ordering strategy presented in this chapter and compared these results to AjaxManager. To measure this overhead, we develop a

Strategy/Server kind	Local	Remote
WeCa FIFO	338.3%	2.9%
WeCa Late	315.7%	2.2%
WeCa Early	338.2%	2.5%
AjaxManager FIFO	14%	3.7 %
AjaxManager Late	0.5%	1.7 %
AjaxManager Early	Not supported	Not supported

Table 7.2: Benchmarks for the control of message causality.

Web application that sends 10,000 identical Ajax requests to a server application, and we use the WeCa message ordering strategies to order the associated server responses. To execute these tests, we first used the machine a for the Web and server application, therefore, the communication is local between both. Later, we used the machine a for the Web application and the machine b for the server application, meaning that the wide-area network is used. Finally, we compare these results to AjaxManager.

Table 7.2 shows a clear difference in the overhead between AjaxManager and WeCa when the Web and server application are in the same machine: 7.25% for AjaxManager and 330.3% for WeCa on average. This is so because the aspect used by WeCa observes and tries matching every join point of the application execution. This difference practically disappears when the server application is on a remote machine because of the latency of the wide-area network. The observed overhead is comparable to that produced by AjaxManager. These results indicate that WeCa is useful for most current Web applications, where the network latency overshadows the induced overhead.

7.6.3.2 Overhead of Reacting to Distributed Causal Relations

This section presents the overhead time introduced by WeCa to match (and react to) patterns that satisfy distributed causal relations. Concretely, we measured the overhead time of the insertion of vector clocks to join points plus the comparison between these vectors. For this experiment, we develop a Web application that sends empty messages to other Web applications. This application is deployed on the machines a and b. The messages are sent through a server application that is in the machine c. These Web applications contain a WeCa instance. In addition, the application of the machine a deploys a stateful aspect to non-simultaneously match 1,000 causal and concurrent sequences composed from 5 to 30 join points .

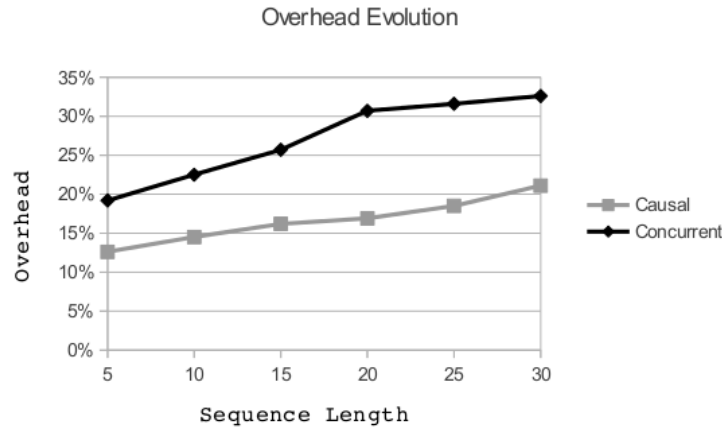

Figure 7.11: Benchmarks for the reaction to distributed causal relations.

Figure 7.11 shows the increment of the overhead of our library when the length of a sequence of join points enlarges. This is so because there are more insertions and comparisons of vector clocks when a sequence is longer. In addition, the figure shows that the overhead is greater when the concurrent relation is verified; this is because the concurrent rule requires one comparison more than the causal rule (Section 7.3.1.2). Note that as in the previous experiment, if we consider network latency, the observed overhead is negligible. Finally, we do not compare these results with other approaches because, to the best of our knowledge, WeCa is the only tool that reacts to distributed causal relations on the Web at runtime.

7.7 Causality on the Web

Although the relation between AOP and distributed computing to address the causality issues on the Web is a distinguishing contribution of this paper, there are already proposals that address these issues using different techniques. We now review JavaScript libraries to control the causality between Ajax requests and server responses and postmortem tools to observe distributed causal relations.

Libraries to control the message causality. A number of lightweight JavaScript libraries have been developed by programmers that rely on function wrappers. In the simplest case, Rico [Rico, 2011], a domain-specific library implicitly uses the FIFO message ordering strategy to sort server responses that are shown in an HTML table. A more elaborate case is

AjaxManager [Farkas, 2011][1], a general-purpose library that offers a fixed set of message ordering strategies. As mentioned in Section 7.2.3, every Ajax request of a WebApp must manually be written to follow a strategy in AjaxManager. WeCa uses the AOP interception to transparently enforce strategies, *i.e.* Ajax requests are not manually written. In addition, WeCa provides as library functions the strategies available in AjaxManager, and our proposal also allows programmers to customize these strategies to add new variants (*e.g.* the Discard Early strategy with callback shown in Section 7.5.2).

Frameworks to observe distributed causal relations. On the one hand, a large number of postmortem tools based on dynamic graphs are available to observe distributed causal relations in interactive WebApps [Wikipedia, 2011][2]. On the other hand, Kossinets *et al.* [Kossinets et al., 2008] use a modified version of vector clocks, which uses timestamp instead of counters, to analyze the minimum time required for information to spread from one user to another. However, these proposals are postmortem. Therefore, they cannot react to distributed causal relations at runtime like WeCa does.

7.8 Summary

This chapter presented WeCa, a practical library to control causality issues in WebApps. This library is a case study of OTM. Concretely, OTM is used to match distributed executions that satisfy certain causal relations.

Contrary to current proposals, WeCa uses general-purpose approaches instead of ad hoc solutions. The key element for this is the flexibility of AspectScript and OTM to define aspects and stateful aspects respectively. This flexibility makes it possible the extension of AspectScript and OTM using aspects that enable the supporting of message ordering strategies and vector clocks. To validate our proposal, we used WeCa with several practical examples from the realm of WebApps.

While many WebApps are highly interactive in nature and may not be so performance-sensitive, it is important to consider the WeCa performance. We plan to improve performance

[1] AjaxManager is an active project: https://github.com/aFarkas/Ajaxmanager/graphs/traffic.

[2] This reference refers to a list of these tools.

through the use of partial evaluation techniques for higher-order languages like JavaScript [Might et al., 2010] (Chapter 10 for more details).

Chapter 8

Supporting Adaptive Software for Primary Students[1]

This chapter presents another case study of OTM: ECOCAM, a framework to support adaptive software for primary school students. In this case study, ECOCAM allows developers to create interactive and visually attractive Web applications, which detect and react to steps that a student follows to solve mathematical operations like additions and subtractions. These Web applications are really context-aware systems that react to certain contexts (*i.e.* patterns of student steps).

The use of computer systems is widely-known to assist in the mathematics learning to primary school students [Kebritchi et al., 2010, Dynarski et al., 2007]. Concretely, these systems detect and react to patterns of interactions that a student follows to solve mathematical operations. However, the development of these systems is hardly modular due to crosscutting modifications needed by these systems to observe, analyze, and react to certain patterns of interactions. Therefore, these adaptive systems are hardly maintainable and reusable. In this chapter, we present ECOCAM, a framework to modularly develop context-aware Web applications for primary school students. These Web applications are used to assist students in the learning of mathematics. We illustrate ECOCAM through the development of a Web application that detects and reacts to the (lack of) use of a mental calculation strategy that primary students use to solve a set of addition problems.

[1]The content of this chapter is a summary and adaptation of our technical report "ECOCAM, a Context-Aware System To Promote Mental Calculation Strategies: A Design and Case Study" [Leger et al., 2011a].

8.1 Introduction

The education area in mathematics has widely benefited from computer science [Kebritchi et al., 2010, Dynarski et al., 2007]. Not surprisingly , the use of computer systems to assist in the learning of mathematics to primary school students is widely-known. These computer systems are really context-aware systems [Satyanarayanan, 2001] because they react to patterns of interactions that a student follows to solve mathematical operations (*e.g.* additions, subtractions, etc). In these context-aware systems, the *context* is a specific pattern of interactions that a student follows, and the *adaptation* is the assistance from the computer system to the student.

As mentioned in Chapter 3, the development of context-aware systems without a mechanism, like trace-based mechanisms, is hardly maintainable and reusable. In the mathematics learning, these systems must be intrusively modified to observe, analyze, and react to every interaction of a student. In this chapter, we propose ECOCAM, a framework to support adaptive software for primary school students. ECOCAM is used to modularly develop Web applications that assist the mathematics learning. This framework is illustrated through the development of a visually rich Web application that detects and reacts whether a student is using the *transfer* strategy [Gálvez et al., 2011] or not to solve an addition problem.

Mental calculation strategies. Primary school students commonly use a *universal strategy* to solve any addition problem: this strategy adds from the last digit of both numbers (*i.e.* units) up to the first digit, and if the addition of any pair of digits is greater than 9, the *carry over process* is used. This process translates one or more units to the next digits of the upper number. To solve some addition problems, the use of the universal strategy is thorny and error-prone. For example, the universal strategy forces students to use the carry over process in the addition $2999 + 22$. Instead, if the previous addition is altered to $3000 + 21$, the carry over process is not necessary anymore, therefore, the equation can be solved faster and more safely. When a student alters an addition problem with the goal of simplifying it, the student is utilizing a *mental calculation strategy* [Taton, 1953, Lethielleux, 2001].

The Web application presented here detects, records, and reacts to the (lack of) use of the transfer strategy over a set of addition problems solved by K-4 students. Moreover, if a student manifests difficulties solving these equations (through the number of bad answers and/or number of times that a student does not use the strategy), the Web application is adapted to assist the student.

The rest of this chapter is organized as follows. Section 8.2 introduces mental calculation strategies and requirements to react to them. Section 8.3 briefly presents ECOCAM. Section 8.4 presents as example a Web application developed using ECOCAM. Section 8.5 summaries this chapter.

8.2 Mental Calculation Strategies & System Requirements To React to Them

In this section, we summarize necessary concepts to introduce our proposal. We first introduce mental calculation strategies [Taton, 1953, Lethielleux, 2001] and later we describe the requirements of a computer system to react them.

8.2.1 Mental Calculation Strategies

Formerly, the goal of the introduction of arithmetic at the school was for students to acquire skills to quickly and exactly calculate mathematical operations like additions and subtractions. These skills would be useful in activities of selling and shopping [Taton, 1953]. To achieve this goal, students learned *universal strategies*: strategies used to solve any mathematical operation in a mechanical manner. The memorizing of some results (*e.g.* multiples of ten) speeds up the calculation process. This memorizing was named *mental calculation strategies*.

Nowadays, the evolution of technology of pocket artifacts and the discussion about benefits to acquire cognitive skills in mathematics suggests the use of mental calculation strategies over universal strategies. The use of mental calculation strategies is based on the idea of taking advantage of the configuration of a specific mathematical operation (*e.g.* multiples of 10). The use of these strategies promotes the understanding of mathematics and prevents the *mechanization*[1] of students to solve mathematical operations. We illustrate benefits of mental calculation strategies through the transfer strategy.

Transfer: A Mental Calculation Strategy for Additions. Various mental calculation strategies like *transfer* and *doubles* have been proposed to solve additions [Shrager and Siegler, 1998]. We now describe the *transfer* strategy because of its simplicity and usefulness. Consider the equation $4998 + 42$ and a student who only knows the universal strategy to solve it. This

[1]Solving a mathematical operation following a sequence of steps unawarely.

student uses the *carry over* process when the (sub)addition of $8 + 2$ returns 10. Later, the *carry over* is repeatedly used for every remaining digit until the most left location (in this case, 4). The use of the carry over process is thorny and error-prone. The transfer strategy moves one or more units between numbers of an addition problem. For example, this equation can be simplified if the student alters it to $5000 + 40$, which does not need the carry over process to be solved.

8.2.2 Computer System Requirements to React to Mental Calculation Strategies

Neither a teacher nor a computer system can observe the internal cognitive process of a student. Thereby, it is a hard task for a teacher to know whether a student is using a mental calculation strategy or not, given that the teacher needs to unobtrusively observe and analyze every step that the student makes to solve a mathematical operation. It is harder for a computer system because this system can only observe through its interactions with the student. These interactions represent the *student steps* used to solve an operation. We now describe requirements of a computer system to observe, analyze, and react to mental calculation strategies:

Observe students steps. Before solving a mathematical operation, a student *thinks* about a plan to solve it; this plan is composed of a sequence of steps. A user interface allows a computer system to observe the sequence of student steps because of the interaction with the student. Rich and intuitive graphical user interfaces can currently be useful to force students to give plenty information about their steps in an unobtrusive manner. However, a computer system must insert *sensors* on every interaction and (possibly) other internal modules in a crosscutting manner.

Analyze student steps. To analyze student steps, a computer system needs an *analyzer* that detects whether the current sequence of student steps represents a mental calculation strategy or not. To get student steps, this analyzer must communicate with every sensor in a crosscutting manner as well.

React to mental calculation strategies. A computer system can react to whether or not a student uses a mental calculation strategy (*e.g.* showing a tutorial). To react to sequences that represent mental calculation strategies, a computer system must execute a piece of code when a certain pattern of sequences is detected.

The previous three requirements can be modularly addressed by the ECOCAM framework through the use of OTM. The patterns of student steps that represent mental calculation strategies are really patterns of join points that must be matched, and the reaction to these strategies is the execution of a piece of advice in OTM.

8.3 ECOCAM

This section presents ECOCAM, a framework to support adaptive software for primary school students. This framework allows developers to build Web applications with rich graphical user interfaces. The graphical interface of these applications provides visual representations of numbers, named *number iconizations*[1]. The manual manipulation of number iconizations by students (*i.e.* student steps) can be modularly observed and analyzed by these Web applications. In addition, when a certain pattern of student steps is detected or not a piece of code can be executed.

8.3.1 Number Iconizations

The ECOCAM framework allows developers to build Web applications, which are specialized for primary school students. These applications can display visual elements named number iconizations. Figure 8.1 shows a number iconization that can be created with this framework. A number iconization is the visual representation of a number, which can be split into three kinds of rectangles: big, medium, and small. The small rectangles represent units, medium ones represent five units, and big ones represent tens. Like Figure 8.2, students can manipulate these rectangles; concretely, rectangles of a number iconization can be extracted, inserted into another one, and split into small amounts. In front of an addition or subtraction, a student manipulates two number iconizations in order to alter the operation. For example, Figure 8.3 shows an addition that is composed of two number iconizations, where both number iconizations are altered to create another addition. Although additions have different numbers, both are equivalent because the result of both is the same.

The manipulation of number iconizations in order to alter an operation allows these Web applications to unobtrusively observe student steps used.

[1] Children often learn the meaning of the numbers through icons [Wiese, 2003].

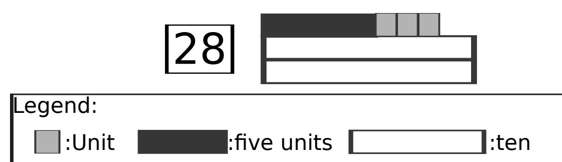

Figure 8.1: Icons provided by the ECOCAM framework.

Operation	**time *t***	**time *t* + *1***
extract		
insert		
split		

Figure 8.2: The different operations supported by number iconizations.

8.3.2 Analyzing Student Steps

ECOCAM allows developers to detect patterns of student steps through the analysis of history of manipulations over number iconizations. We now explain how ECOCAM supports the detection and the ability of reaction to these patterns.

8.3.2.1 Detecting Patterns of Student Steps

Developers can declare values that selectively observe every student step. The following three lines of code declare a value that represents certain student steps over a number iconization a:

```
1 var allA = ECOCAM.observe("a");
2 var tenA = ECOCAM.observe("a").tens();
3 var extractUnitA = ECOCAM.observe("a").units().extract();
```

The first line creates a value that represents all student steps of any rectangle of the a number iconization[1]. The second line creates a value that represents student steps over tens (big rectangles) of a. Finally, the third line creates a value that represents extractions of small

[1] A number iconization is identified by a string because it corresponds to an identifier of the Document Object Model (DOM) of a Web page.

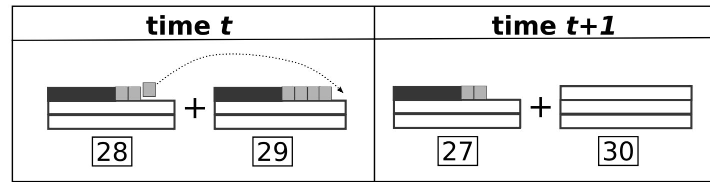

Figure 8.3: Manipulating number iconizations to alter an addition.

rectangles (units) from a. As the ECOCAM framework uses OTM, these values correspond to OTM patterns that match sequences of student steps (*i.e.* calls to methods technically). Thereby, these values can be composed to build other ones. For example:

```
var insertUnitB = ECOCAM.observe("b").units().insert();
var transferFromAtoB = seq(extractUnitA, insertUnitB);
```

The transferFromAtoB pattern detects a transfer of a unit from the number iconization a to b made by a student, who does not make any other student step between extractUnitA and insertUnitB. For example, in the following two patterns of student steps, transferFromAtoB matches the left pattern matches but does not match the right pattern.

$extractUnitA \rightarrow insertUnitB$ $\qquad$ $extractUnitA \rightarrow extractUnitA \rightarrow insertUnitB$

8.3.2.2 Reacting to Patterns of Student Steps

It is possible to execute a piece of code when a pattern of student steps is detected or not. For example, the following piece of code shows a message of whether or not a student transfers a unit from a to b:

```
ECOCAM.reactTo(transferFromAtoB, function(detected) {
  message = detected? "A unit was transferred from 'a' to 'b' " :
                     "A unit was not transferred from 'a' to 'b' or
                     the student took more steps than only transferring a unit";
  alert(message);
});
```

The reactTo method takes as parameters the pattern of student steps that should be detected and a reaction: a function that is executed whether the pattern of student steps is detected or not. This latter function receives a boolean that indicates whether the pattern was detected (more on this in Section 8.4.1.2).

OTM in ECOCAM. ECOCAM customizes the matching process semantics of OTM stateful

aspects to *a)* disable the multiple matches of the pattern and *b)* remove matches that do not *exactly* match the pattern of student steps (similar to the tracematch semantics presented in Chapter 6.5.3). In ECOCAM, this customization is useful to execute a piece of code at the moment that a pattern of student steps is matched or is no longer matched.

ECOCAM allows developers to build Web applications that unobtrusively *observe*, *analyze*, and *react* to patterns of student steps used to solve mathematical operations. The next section presents a Web application developed through ECOCAM, which reacts to the (lack of) use of the transfer mental calculation strategy.

8.4 Promoting Mental Calculation Strategies using ECOCAM

This section illustrates ECOCAM through the development of a Web application that detects and reacts to the (lack of) use of the transfer mental calculation strategy. This Web application was evaluated with a group of K-4 students (approximately 9 years old) that solved a set of additions.

Figure 8.4 shows two scenarios of the Web application developed using the ECOCAM framework. The left scenario shows that the Web application has a cloud that contains *1)* the equation that a student has to solve and *2)* the input to write the addition result. In addition, the Web application contains *3)* two number iconizations that represent the two numbers of the addition. The student can manipulate number iconizations to alter the addition or can immediately write the addition result. If the student alters the addition, this alteration is shown below the cloud. The right scenario shows that the student clicks *4)* the arrow button to join both number iconizations in order to show a summarized version of number iconizations.

The Web application presents a predefined set of additions to a student. These additions are considered complex for potential students: primary school students. For this reason, the manipulation of two number iconizations is useful for these students.

8.4.1 Detecting and Reacting to the Transfer Strategy

Although the set of additions are complex for potential students, these additions can be solved easier if they utilize the transfer mental calculation strategy. Using ECOCAM, the Web application detects and reacts to the (lack of) use of this strategy.

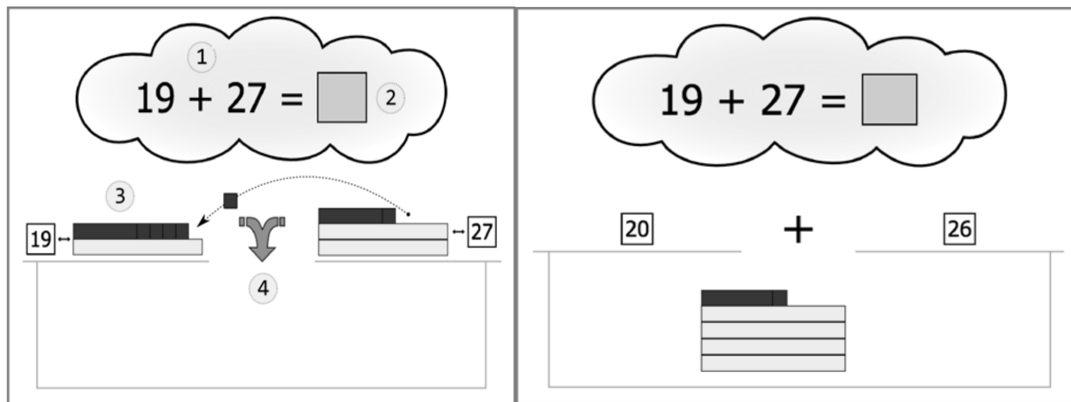

Figure 8.4: Two scenarios of the Web application developed using ECOCAM. The left scenario shows a student that is transferring a unit from 27 to 19. The right scenario shows the resultant addition after the transfer $(20 + 26)$.

8.4.1.1 Detecting the Transfer Strategy

Each addition of this Web application has associated a pattern of student steps that represents the appropriate use of the transfer strategy. For example, the transfer strategy for the addition $19 + 27$ (shown in Figure 8.4) is the extraction of a unit from the right number iconization followed by the insertion of a unit to the left number iconization. In ECOCAM, the previous sequence is expressed as:

```
var extractUnitRight = ECOCAM.observe("right").units().extract();
var insertUnitLeft = ECOCAM.observe("left").units().insert();
//the sequence associated with the addition 19 + 27
var transferFromRightToLeft = seq(extractUnitRight, insertUnitLeft);
```

The transferFromRightToLeft sequence represents the appropriate use of the transfer strategy for the addition $19+27$. This sequence only detects a translation of a unit between both number iconizations. The methods observe, units, and extract are functions provided by the ECOCAM framework. All these functions return functions that support other functions[1]. For example, the implementations of observe and units are:

```
var observe = function(id) {
  var pointcut = function(jp, env) {
      return /* jp corresponds to an event of the right number iconizations*/
  };

  pointcut.units = function() {
```

[1] A JavaScript function is also a JavaScript object.

```
    var left = this;
    var innerPointcut = function(jp) {
      return left(jp) && /* jp correspond to an event of an unit*/
    }
    innerPointcut.extract = //implementation

    return innerPointcut;
  };

  return pointcut;
}
```

Every time one of these functions calls to a method, a conjunction of two pointcuts is created. Thus, it is possible to match accurately the needed event.

8.4.1.2 Reacting to the Transfer Strategy

The Web application records the transfer strategy for every addition whether a student uses or not. This information is used to generate a report about the student. In addition, if the student provides evidence of the ignorance of the transfer strategy, the Web application triggers an assistance to the student. This assistance is an animation that shows how to solve some additions using this strategy. In ECOCAM, the reaction to the (lack of) use of the transfer strategy, needed by this Web application, is expressed as a JavaScript function:

```
var reaction = function(detected) {
  report.record(detected);
  if (/*condition to assist to the student*/)
    displayAnimation_HowToUseTransferStrategy();
};

ECOCAM.reactTo(transferFromRightToLeft, reaction);
```

In the reaction function, if the number of bad answers is greater than a number of times (*e.g.* three), this function displays the animation to the current student. In addition, this function uses a report object that records whether the pattern of student steps is detected or not, which is used to generate a report of the student about the use of transfer mental calculation strategy (Chapter 6.5.3)

In ECOCAM, a pattern represents a strategy. Therefore, the fail of matching of a pattern means that the strategy was not used. The matching semantics of OTM stateful aspects are customized to exactly match a pattern of adjacent join points (similar to the matching semantics of tracematches, Chapter 6.5.3), and if this unwanted join point among this pattern, the stateful

aspect abort the matching of the pattern and the reactTo method with the detected = false is executed.

8.4.2 Evaluation

The Web application developed using ECOCAM was evaluated in two different ways. First, we analyzed a large collection of data obtained from the use of this application. The goal of this first evaluation is to know the number of times that the assistance of the Web application was necessary (*i.e.* how many students used the transfer mental calculation strategy). Second, we carried out personal interviews with some students. The goal of this evaluation is to get feedback of its user interface provided by the ECOCAM Application Programming Interface (API).

Using data collection. The Web application was evaluated with 17 K-4 students. Each student solved 32 equations, which are considered complex for these students and the transfer strategy is the appropriate mental calculation strategy to solve them. As a consequence, 544 (= 17*32) answers were obtained, of which 433 (79.96%) were correct and 111 (20.4%) were incorrect. The assistance of the Web application was only displayed 13 times (2.4%).

Using interviews. Using personal interviews, we got feedback about the Web application and its user interface. Most students enjoyed solving additions problems and did not feel the intrusion of the Web application while they solved the addition problems.

We now mention two observations by students after using the application. First, Oscar, 10 years old, commented that the user interface reinforced the strategy that he learned by himself. Miranda, 9 years old, commented that the user interface did not support her own mental calculation strategy: she adds tens then units. Miranda's strategy is a typical strategy for most students, however, this strategy presents difficulties when units add a number greater than 9, and students end up using the universal strategy again.

8.5 Summary

This chapter presented ECOCAM, a framework to support adaptive Web applications for primary school students. This framework is another case study of OTM because it is used to develop context-aware Web applications that react to steps that a student follows to solve a mathematical

operation. In these Web applications, the *context* is a certain pattern of student steps, and the *adaptation* is the assistance from the Web application to the student.

ECOCAM is illustrated through the development of a Web application that reacts to the (lack of) use of the transfer mental calculation strategy to solve additions. This Web application has been evaluated with a group of K-4 students. These students correctly answered the majority of additions without using the Web application assistance. We plan to extend this Web application to react to *a)* other mental calculation strategies for additions (*e.g.* doubles[1]) and *b)* mental calculation strategies for subtractions.

[1]An addition like $15 + 16$ is altered to $15 + 15 + 1$, which is easier to solve because $15 + 15$ is an addition of doubles whose result is generally known.

Part III: Conclusions

Chapter 9

Contributions

This chapter briefly reviews the main contributions of this thesis work. Our contribution can be divided in four different topics:

- **AspectScript**. An expressive aspect language for the Web. (Chapter 4).
- **ETM**. A model of an expressive trace-based mechanism (Chapter 5).
- **OTM**. A model of an open implementation of a trace-based mechanism (Chapter 6).
- **MatcherCells**. A self-replication algorithm to flexibly match traces (Chapter 6).
- **Case studies**. Two case studies where OTM is used (chapters 7 and 8).

The following sections summarize these contributions.

9.1 AspectScript

Motivation. JavaScript is widely used to build increasingly complex Web applications. Not surprisingly, these applications need to address crosscutting concerns. Therefore support for aspect-oriented programming is crucial to preserve proper modularity. However, there is no aspect-oriented extension of JavaScript that fully embraces the characterizing features of that language: dynamic prototype-based programming with higher-order functions. In Chapter 4, we presented AspectScript [Toledo et al., 2010], a full-fledged AOP extension of JavaScript that adopts higher-order programming and dynamicity as its core design principles. In AspectScript, pointcuts and advices are standard JavaScript functions, bringing the benefits of higher-order programming

patterns to define aspects. We illustrated AspectScript by means of several practical extensions to a Facebook-like Web application, a representative Web 2.0 application.

Technical details. AspectScript is a first concrete and expressive aspect-oriented extension for JavaScript. This aspect language fully embraces the characteristic features of JavaScript by supporting higher-order pointcuts, advices, and aspects [Dutchyn et al., 2006], a full-fledged join point model, customizable quantified events [Rajan and Leavens, 2008], and dynamic aspect deployment with expressive scoping [Tanter, 2008b]. Aspect reentrancy [Tanter, 2008a] is avoided without causing any burden on programmers using execution levels [Tanter, 2010]. This combination of features is unique in the current design space of aspect languages. To date, AspectScript still has important challenges like portability, performance, and debugging support.

Relation to the thesis. The expressiveness of this aspect language (in particular, pointcuts and advices) is the base on top of which we provide an implementation of our model, an open implementation of an trace-based mechanism (Chapter 6). Pointcuts of AspectScript can encompass the *matching process* of a stateful aspect and advices of AspectScript can encompass the *advising process*. Nevertheless, it is important to mention that AspectScript is only a powerful means to achieve practical results of our thesis work (and other proposals [Toledo and Tanter, 2011a, Toledo and Tanter, 2011b]).

9.2 ETM

Motivation. Real-world applications bring together common issues related to security flaws, application errors, and crosscutting concerns. Different trace-based mechanisms [Allan et al., 2005, Goldsmith et al., 2005, Herzeel et al., 2006, Martin et al., 2005, Ostermann et al., 2005, Eugster and Jayaram, 2009] with distinctive features have been proposed to solve these particular issues. Most existing trace-based mechanisms are limited to express the pattern that should match a certain program execution trace. For example, in tracematches [Allan et al., 2005], patterns *a)* are limitedly expressed by regular expressions, *b)* cannot gather and use bindings out of the context of the current join point, and *c)* are not reusable. In Chapter 5, we presented a model of an Expressive Trace-based Mechanism (ETM) [Leger and Tanter, 2010b], which allows developers to flexibly express and reuse stateful aspects, in particular patterns. In addition, we

presented a complete and practical implementation of ETM for JavaScript, which allows us to show its usefulness through better modularization support in building Web applications. Besides, we used Typed Racket [Tobin-Hochstadt and Felleisen, 2008] to formalize the core semantics of ETM as a typed functional program.

Technical details. ETM is a model of a trace-based mechanism that allows developers to flexibly express stateful aspects. Various limitations of existing trace-based mechanisms are addressed thanks to the flexibility provided to define stateful aspects. In ETM, patterns and advices are first-class functions; therefore, patterns and advices can be reused and composed. In addition, developers, through an explicit environment, can flexibly gather and use bindings during the matching of a pattern. Finally, a pattern in ETM can alter its definition at runtime to adapt to the execution context.

Relation to the thesis. From a conceptual and implementation viewpoint, ETM is the first step towards achieving our thesis goal, by offering flexibility in defining patterns. Nevertheless, the processes of matching and advising of stateful aspects in ETM are still hidden and cannot be customized by developers, ETM is extended to become an Open Trace-based Mechanism (OTM), which is the main contribution of our work (Chapter 6).

9.3 OTM

Motivation. ETM, like existing trace-based mechanisms, has different features and semantics, which are fixed for developers. As a consequence, developers must "code around" existing trace-based mechanisms in contortive ways because they cannot *a)* take advantage of the specific features and semantics of existing mechanisms and *b)* devise new semantic variants to satisfy specific needs. In Chapter 6, we presented a model of an Open implementation of a Trace-based Mechanism (OTM) [Leger and Tanter, 2010a]. The core semantic elements of OTM are open to customization. In addition, we presented a complete and practical implementation of OTM for JavaScript, which is currently an extension of ETM. This implementation allows us to *a)* emulate the semantics of specific trace-based mechanisms like Alpha and tracematches and *b)* show how new semantic variants are useful to develop modular (Web) applications. As for ETM, we used Typed Racket to formalize the core of OTM.

Technical details. OTM is a model of an open trace-based mechanism that allows developers to flexibly express stateful aspects and customize their core semantics. Stateful aspect semantics of existing trace-based mechanisms and new possible variants can be implemented in OTM. All customizations of stateful aspects are carried out by functions and are local to every stateful aspect. In particular, developers can customize the processes of matching and advising of a stateful aspect. The matching process is used to control multiple matches of a pattern, and the advising process is used to schedule the multiple executions of an advice.

Relation to the thesis. As mentioned previously, OTM is the main contribution of this thesis work. To the best of our knowledge, our proposal is the first open implementation of a trace-based mechanism that supports the customization of the core semantics of stateful aspects.

9.4 MatcherCells

Motivation. Trace-based mechanisms use a matching process to carry out the matching of a program execution trace. To the best of our knowledge, matching processes of existing trace-based mechanisms are not flexible, meaning that the matching semantics of stateful aspects is fixed. For instance, most matching processes allow multiple matches of a pattern when the associated context information is different. Obtaining a different matching semantics requires ad hoc modifications of the aspects, if at all possible. In order to allow flexible customization of the matching semantics of a given aspect, Chapter 6 presents a self-replication algorithm [Neumann, 1966], named MatcherCells [Leger and Tanter, 2012]. Through the composition of simple reaction rules, MatcherCells makes it possible to express a range of matching semantics, per stateful aspect.

Technical details. MatcherCells is a self-replication algorithm to flexibly to match program execution traces. Self-replication algorithms are inspired by cellular behavior. Concretely, these algorithms emulate the reactions of biological *cells* to a trace of *reagents*. The reaction of a cell to a reagent can be the creation of an identical copy of itself with a small variation, nothing, death, or some of these combinations. These reactions are defined by a set of rules that govern the evolution of cells. In MatcherCells, a cell contains the pattern should be matched and its

evolution depends on the set of rules defined for a stateful aspect, and reagents correspond to join points generated by a program execution. By composing of simple rules, one can flexibly express diverse matching semantics for a stateful aspect.

Relation to the thesis. As OTM goal is to allow developers to customize the core semantics of a stateful aspect, in particular the matching semantics, we need a matching algorithm that can be flexibly customized. Unlike existing algorithms for matching, MatcherCells allow developers to customize it.

9.5 Case Studies

We used OTM in two case studies: WeCa [Leger et al., 2011b] and ECOCAM [Leger et al., 2011a]. Both case studies are scenarios where a context-aware system uses OTM to detect contexts and adapt its behavior in a modular and flexible manner.

9.5.1 WeCa

Ajax allows JavaScript developers to create interactive, collaborative, and user-centered Web applications. These Web applications behave as distributed systems because processors are user machines that are used to send and receive messages between one another. Unsurprisingly, these applications have to address the same causality issues present in distributed systems like the need *a)* to control the causality between messages sent and responses received and *b)* to react to distributed causal relations.

To react to (or control) causal relations in Web applications, it is necessary for these applications to observe a distributed program execution, which is composed of various Web application executions, and analyze causal relations among events. Whenever a certain distributed program execution that satisfies a specific causal relation is observed, the Web application(s) must change their behavior to react to this causal relation. JavaScript developers currently use techniques that are not very flexible and need to intrusively modify Web applications in order to react to distributed causal relations.

OTM is used to develop WeCa, a practical library that allows for modular and flexible control over these causality issues in Web applications. In contrast to current proposals, WeCa is based on stateful aspects, message ordering strategies [Murty and Garg, 1997], and vector clocks [Lamport, 1978]. Thanks to the expressiveness of OTM stateful aspects, they can be extended to detect

distributed causal relations (*i.e.* contexts) and adapt the behavior of Web applications. We illustrated WeCa in action with several practical examples from the realm of Web applications. For instance, we analyzed the flow of information in a Web application similar to Twitter using WeCa.

9.5.2 ECOCAM

Primary school students commonly use the *universal strategy* to solve any addition problem[1]. However, some addition problems can be solved faster and safer if these students take advantage of the particular numbers that compose an addition. For example, in the addition $2999 + 22$, the universal strategy forces to students to use the *carry over* process, which is a thorny and prone-error process. Instead, this addition can be altered to $3000 + 21$, which does not need the carry over process. This kind of alterations are known as mental calculation strategies [Taton, 1953]. Investigating what mental calculation strategies the students can learn before they internalize the universal strategy is a valuable research topic [Shrager and Siegler, 1998].

Trying to detect the mental calculation strategy that a student uses to solve an addition problem is a hard task for a computer system because it needs to expose and analyze every step that the student follows. If a user interface of a computer system permits every step that a student follows to be gathered, it is necessary to analyze these steps in order to detect the mental calculation strategy used. The analyzing process is crosscutting because it is necessary to modify, at least, every method of the system that permits the interaction with the student. In addition, mental calculation strategies are detected by an analysis of patterns of steps that a student follows to solve an addition problem [Gálvez et al., 2011], meaning that the computer system needs a powerful matching mechanism to detect these strategies.

OTM is used to develop ECOCAM, a framework to support adaptive software for primary school students. Using this framework, we create a context-aware system that modularly detects the mental calculation strategies that these students use to solve addition problems. In this context-aware system, there are stateful aspects that are constantly observing and analyzing steps that a student follows. Whenever there is a matching of a stateful aspect, it means that a mental calculation strategy is detected. We used this context-aware system to analyze behavior of several K-4 students to solve a predefined set of addition problems [Leger et al., 2011a].

[1] This strategy adds from the last digit of both numbers (*i.e.* units) up to the first digit. If the addition of any pair of digits is greater than 9, the carry over process is used. This process translates a unit to the next digit of the upper number.

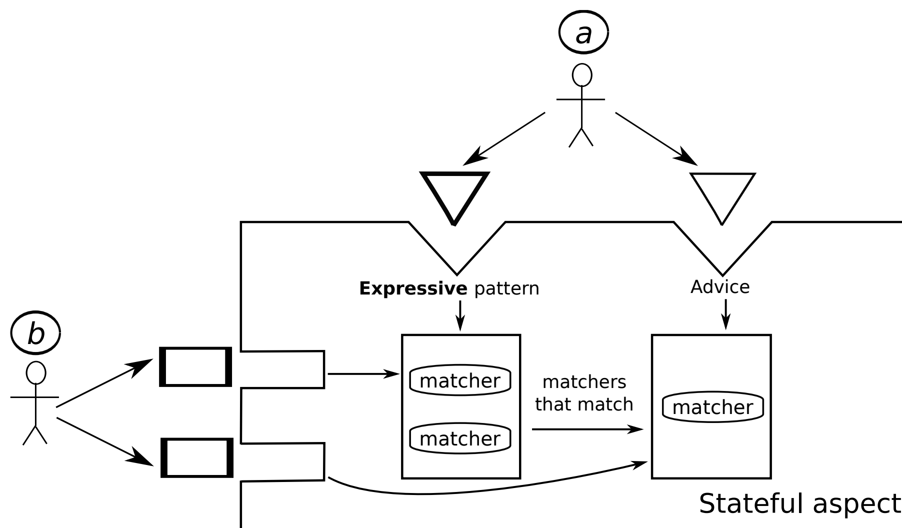

Figure 9.1: The big picture of our work. Contributions are in **bold**.

9.6 Thesis Work in a Few Words

Figure 9.1 represents the big picture of our thesis work, which shows the progress in the research area of programming languages, in particular trace-based mechanisms. Before our work, trace-based mechanisms allowed developers to limitedly express stateful aspects. For example, developers could not express adaptive patterns and/or patterns that gather lists of bindings during the matching process (Chapter 5). In addition, developers could not customize the core semantics of stateful aspects, for example, simultaneous advice executions could not be scheduled (Chapter 6). This thesis work allows developers *a)* to flexibly express stateful aspects, in particular patterns, and *b)* to customize stateful aspect semantics like the life-cycle of matchers, abstractions that match a pattern, and the scheduling of simultaneous advice executions.

Chapter 10

Perspectives

The contributions of this thesis work directly open perspectives for future work. Apart from the perspectives discussed here, which originate from our contributions, a number of related research directions are worth exploring. This chapter briefly discusses the most important ones.

The first two sections discuss formal research of semantics and performance for OTM. The last section proposes a new case study: conscientious software. Some of these research proposals represent on-going work.

10.1 Formal Semantics for OTM

We have provided a practical implementation of our model for JavaScript, a dynamic and prototyped-based programing language. Instantiating our model in other languages like Perl, Python, Smalltalk, and Ruby, should not be very different since these languages follow similar design rationale. Instantiating our model in languages like Java would be similar as well, however, this instantiation would result less flexible because Java does not support high-order functions and the current pattern model of OTM depends on functions that are created at runtime[1].

It would be worthwhile to study the formal semantics of OTM. Although we use Typed Racket to provide a typed functional description of our model (Appendix B) (like Masuhara *et al.* in [Masuhara et al., 2003] propose for the pointcut-advice model), this artifact is not sufficient to demonstrate properties of our model. For example, this artifact is not sufficient to demonstrate that the semantics of OTM for tracematches proposed in Chapter 6 really correspond to the semantics of tracematches [Allan et al., 2005].

[1] An implementation of OTM for object-oriented languages Like Java should replace function abstractions with objects.

10.2 OTM Performance

A recurrent topic in the body of literature of trace-based mechanisms is performance [Allan et al., 2005, Herzeel et al., 2006, Ostermann et al., 2005]. The contributions of our model are focused on identifying and customizing the core elements of a trace-based mechanism rather than performance. Nevertheless, certain customized semantics of a stateful aspect in OTM can notably improve performance in some scenarios. For example, disabling multiple instances of a pattern can improve performance where it is only necessary that one pattern matches at the same time. Given that we identify core elements of a trace-based mechanism, it seems a good idea to conduct some research about how to locally improve performance of every element of a trace-based mechanism. Bodden *et al.* in [Bodden et al., 2010, Bodden et al., 2005, Bodden, 2010] have studied several proposals in this line of research, *e.g. dependency state machines*.

10.3 OTM in Conscientious Software

The auto-maintenance of software is an active research topic [George et al., 2003, Desmky and Rinard, 2003]. In particular, Gabriel and Goldman [Gabriel and Goldman, 2006] propose the term *conscientious software* to define software that is responsible for itself and its features. The authors argue that the software should be separate functional work from its own maintenance. The maintenance should be carried out by small systems that actively monitor the program execution, continually perform self-testing, catch errors and automatically recover/learn from them, etc.

In [Fleissner and Baniassad, 2007], the authors present the design of an architecture of aspects for conscientious software. However, this architecture presents two drawbacks. First, the language to identify events is based on a pointcut language, meaning that developers can define aspects that only react to the current event. Second, the definition of aspects cannot be updated at runtime, therefore, unpredictable conditions or situations cannot be incorporated later. OTM is a powerful trace-based mechanism that supports the definition of expressive stateful aspects that modularly observe and react to a program execution. The dynamic nature of OTM allows stateful aspects to adapt their patterns and advices at runtime to satisfy unforeseen needs of the software. Determining the correct adaptation of a stateful aspect can be carried out using artificial intelligence, in particular genetic programming, cellular automata, swarm intelligence, evolutionary self-replicating algorithm, etc. We feel that it would be worthwhile to study how OTM can be applied in conscientious software.

Appendix A

Summary of the OTM API

This appendix summaries the API of OTM to define a stateful aspect. A stateful aspect is an object with at least three properties:

pattern:	$JoinPoint \times Env \rightarrow False \bigcup Env \bigcup Pattern \times Env$
advice:	$JoinPoint \times Env \rightarrow Value$
kind:	$Constant$

A pattern returns false if it does not match the current join point, Env if it matches the trace of join points, and a pair of pattern-environment if it advances in its matching. An advice takes as parameters the last join point matched by the pattern and an environment of bindings gathered during the matching. The advice returns a value of the base language. Finally, the kind value indicates if the advice is before, around, or after (more in Chapter 5).

To customize the semantics of a stateful aspect, two optional properties, which are functions, can be aggregated to the stateful aspect object:

matching:	$List < Cell > \times JoinPoint \rightarrow List < Cell \bigcup MatchCell >$
advising:	$List < MatchCell > \times JoinPoint \times Advice \rightarrow Value$

The matching function is executed for every join point. The list of cells of the matching parameter represents the current possible matches of the pattern. The join point passed as parameter effects this list of cells. This function returns the list of cells for the next join point. The OTM distribution provides a set of predefined matching functions. The advising function is executed every time that there is a match of the pattern. The list of match cells represents the matches of

the pattern. The developer decides how the advice is executed for each match cell. The function returns the same kind of value that the advice returns. See Chapter 6 for more details.

Appendix B

A Typed Functional Description of OTM

This appendix shows a typed function description of OTM (Chapter 6) using Typed Racket [Tobin-Hochstadt and Felleisen, 2008]. First, we present the customization needed to emulate ETM, which is the OTM default semantics. Then, we present the modules of Pattern, Cell, Rules, and OTM. The full description of OTM is available at http://pleiad.cl/otm.

```
; The ETM matching semantics is single match at a time
(: etm-matching (Pattern -> Rule))
(define (etm-matching pattern)
  ((add-seed pattern)(kill-creators (apply-reaction))))

; In ETM, an advice is only executed
(: etm-advising Advising)
(define (etm-advising match-cells jp advice)
  (advice jp (Cell-env (first match-cells))))

; Deploy for aspects with the ETM semantics
(: etm-deploy (Pattern Advice -> Aspect))
(define (etm-deploy pattern advice)
  (deploy pattern advice (etm-matching pattern) etm-advising))
```

Figure B.1: The module of the ETM semantics: the default semantics of OTM.

```
; In an aspect language, a pointcut can return an environment if
it matches, otherwise false.
(define-type Match Env)
(define-predicate Match? Match)

(define-type Fail #f)
(define-predicate Fail? Fail)

(define-type PcResult (U Match Fail))

; A pattern can return the same of a pointcut (U #f Env). In ad-
dition, a pattern can return another kind of value: (Pairof Pat-
tern and Env), which represents the *advance* in the matching of
the pattern. In a few words, Pattern: JPCall Env -> (U PcResult
(Pairof Pattern Env))

; It is equivalent to (U Match Fail Advance)
(define-type PatResult (Rec SM (U PcResult (Adv SM))))
; Internal types of a pattern
(define-type (Pat A) (JPCall Env -> A))  ; Pattern
(define-type (Adv A) (Pairof (Pat A) Env)) ; Advance
; Exported types of a pattern
(define-type Pattern (Pat PatResult))
(define-predicate Pattern? Procedure)

(define-type Advance (Adv PatResult))
(define-predicate Advance? (Pairof Procedure Env))

; call: a pattern that *only* matches a call join point (aka
poincut).
(: call (Value -> Pattern))
(define (call id-fun)
  (λ (jp env)
    (if (eq? id-fun (JPCall-fun jp))  env #f)))

; seq: pattern that matches the pattern left followed by right.
(: seq (Pattern Pattern -> Pattern))
(define (seq left right)
  (λ (jp env)
    (let ([result (left jp env)])
      (cond
        ((Match? result) (cons right result)) ; left matches
        ((Advance? result) (cons (seq (car result) right) (cdr result)))
        (else #f))))) ; otherwise
```

Figure B.2: The pattern module and two pattern designators.

```
(define-struct: Cell
 ([pat : (U Pattern Null)] ; Null for match-cells
  [env : Env]
  [creator : (U Cell Null)] ; Null for seed))

; Creation of a seed
(: seed (Pattern -> Cell))
(define (seed pattern)
 (make-Cell pattern (new-Env) null))

; The reaction a cell. This implementation is not part of matcher-
cells
(: react (Cell JPCall (Env Pattern Cell -> Env) -> Cell))
(define (react cell jp creation)
 (let*
     ([next (Cell-pat cell)]
      [pattern (if (Pattern? next) next (error "a match-cell"))]
      [result (pattern jp (Cell-env cell))])
   (cond
     ; cases of the result of a cell
     [ (Advance? result)
       (Cell (car result) (creation (cdr result) (car result) cell) cell)]
     [ (Match? result)
       (Cell null result cell)]
     [ else
       cell])))  ; the same cell
```

Figure B.3: The Cell module.

```
; The Rule type and its predicate
(define-type Rule ((Listof Cell) JPCall -> (Listof Cell)))
(define-predicate Rule? Procedure)

; Rule Designators

; It applies the reaction of a cells and allows developers to cus-
tomize the creation of a cell
(: apply-reaction : (case->
                     (-> Rule)
                     ((Env Pattern Cell -> Env) -> Rule)))
(define (apply-reaction [#{creation : (Env Pattern Cell -> Env)} (λ (env pat creator) env)])
  (λ: ([cells : (Listof Cell)] [jp : JPCall])
      (remove-duplicates (append (map (λ: ([cell : Cell])
                                    (react cell jp creation)) cells)  cells))))

; It kills all cells that created a new cell in an iteration
(: kill-creators (Rule -> Rule))
(define (kill-creators rule)
  (λ (cells jp)
    (let ([next-cells (rule cells jp)])
      (difference next-cells (creators next-cells cells)))))

; It adds a seed if there is no cells or only match cells
(: add-seed (Pattern -> (Rule -> Rule)))
(define (add-seed Pattern)
  (λ (rule)
    (λ (cells jp)
      (let ([next-cells (rule cells jp)])
        (if (or (= (length next-cells) 0) (only-match-cells? next-
cells))
            (cons (seed pattern) next-cells)
            next-cells))))))

; It always keeps a seed in the solution of cells
(: keep-seed (Pattern -> (Rule -> Rule)))
(define (keep-seed pattern)
  (λ (rule)
    (λ (cells jp)
      (let ([next-cells (rule cells jp)])
        (if (> (count-seed next-cells) 0)
            next-cells
            (cons (seed pattern) next-cells))))))
```

Figure B.4: Some rules designators.

```
; Stateful Aspect structure
(define-struct: Aspect
  (
   [pattern : Pattern]
   [advice : Advice]
   ; [kind : Symbol] ;;all stateful aspects are around
   [matching : Rule]
   [advising : Advising]
   [cells : (Listof Cell)]))


; Type of advice and its advising process
(define-type Advice (JPCall Env -> Value))
(define-type Advising ((Listof Cell) JPCall Advice -> Value))

; It deploys a stateful aspect with custom semantics
(: deploy (Pattern Rule Advising -> Aspect))
(define (deploy pattern advice matching advising)
  (make-Aspect pattern advice matching advising (list (seed pattern))))

; The weave of a stateful aspect. Here, the matching and advising
process are applied
(: weave (Aspect JPCall -> (Pairof Aspect Value)))
(define (weave asp jp)
  (let*
      ([temp-cells ((Aspect-matching asp) (Aspect-cells asp) jp)]
       [match-cells (filter match-cell? temp-cells)])
    (cons (update-aspect asp (filter no-match-cell? temp-cells))
          (if (> (length match-cells) 0)
              ((Aspect-advising asp) match-cells jp (Aspect-
advice asp))
              (proceed jp)))))
```

Figure B.5: The OTM module.

References

[Ajaxpect, 2011] Ajaxpect (2011). Ajaxpect. A JavaScript framework for aspect-oriented programming. http://code.google.com/p/ajaxpect/. 29, 52

[Allan et al., 2005] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Adding trace matching with free variables to AspectJ. In [OOPSLA 2005, 2005], pages 345–364. ACM SIGPLAN Notices, 40(11). 2, 6, 7, 15, 17, 18, 20, 32, 58, 72, 73, 75, 76, 80, 88, 103, 144, 150, 151

[AOSD 2010, 2010] AOSD 2010 (2010). *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France. ACM Press. 165

[Aracic et al., 2006] Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. (2006). An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag. 11, 29, 36, 43

[AspectJS, 2011a] AspectJS (2011a). AspectJS. A function-call framework in JavaScript. http://www.aspectjs.com/. 29, 52

[AspectJS, 2011b] AspectJS (2011b). AspectJS. A JavaScript framework for aspect-oriented programming. http://zer0.free.fr/aspectjs/. 52

[Augusto, 2007] Augusto, J. C. (2007). Ambient intelligence: The confluence of pervasive computing and artificial intelligence. In Schuster, A. J., editor, *Intelligent Computing Everywhere*, pages 213–234. Springer London. 13

[Augusto and Nugent, 2006] Augusto, J. C. and Nugent, C. D. (2006). The role of artificial intelligence. In Carbonell, J. G. and Siekmann, J., editors, *Designing Smart Homes*, Lecture Notes in Computer Science. Springer. 13

[Barendregt, 1984] Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. North-Holland. 63

[Benavides Navarro et al., 2008] Benavides Navarro, L. D., Douence, R., and Südholt, M. (2008). Debugging and testing middleware with aspect-based control-flow and causal patterns. In *Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference*, volume 5346 of *Lecture Notes in Computer Science*, pages 183–202, Leuven, Belgium. Springer-Verlag. 19

[Benerecetti et al., 2001] Benerecetti, M., Bouquet, P., and Bonifacio, M. (2001). Distributed context-aware systems. *Human-Computer Interaction*, 16:213–228. 13

[Benevenuto et al., 2009] Benevenuto, F., Rodrigues, T., Cha, M., and Almeida, V. (2009). Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 49–62, New York, NY, USA. ACM. 102, 106

[Bodden, 2010] Bodden, E. (2010). Specifying and exploiting advice-execution ordering using dependency state machines. In *International Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*. 151

[Bodden, 2012] Bodden, E. (2012). Personal Communication. July 10, 2012. 95

[Bodden et al., 2005] Bodden, E., Chen, F., and Rosu, G. (2005). Dependent advice: a general approach to optimizing history-based aspects. In [OOPSLA 2005, 2005], pages 3–14. ACM SIGPLAN Notices, 40(11). 151

[Bodden et al., 2006] Bodden, E., Forster, F., and Steimann, F. (2006). Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition. 45, 46

[Bodden et al., 2010] Bodden, E., Lam, P., and Hendren, L. (2010). Clara: a framework for statically evaluating finite-state runtime monitors. In *1st International Conference on Runtime Verification (RV)*, volume 6418 of *LNCS*, pages 74–88. Springer. 151

[Cerny, 2011] Cerny (2011). Cerny. A javascript framework for method-call interception. http://www.cerny-online.com/cerny.js/. 29, 52

[Chen and Roşu, 2003] Chen, F. and Roşu, G. (2003). Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127. 20

[Chen and Roşu, 2007] Chen, F. and Roşu, G. (2007). Mop: An efficient and generic runtime verification framework. In [OOPSLA 2007, 2007], pages 569–588. ACM SIGPLAN Notices, 42(10). 20, 90

[Clifton et al., 2000] Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. (2000). MultiJava: Modular open classes and symmetric multiple dispatch in java. In *Proceedings of the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, pages 130–145, Minneapolis, Minnesota, USA. ACM Press. ACM SIGPLAN Notices, 35(11). 11

[Costanza and Hirschfeld, 2005] Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming – an overview of ContextL. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA. 1

[Dantas et al., 2008] Dantas, D. S., Walker, D., Washburn, G., and Weirich, S. (2008). AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14. 53

[Darlington, 2000] Darlington, K. (2000). *The essence of expert systems*. Essence of computing series. Prentice Hall. 18, 80

[Desmky and Rinard, 2003] Desmky, B. and Rinard, M. (2003). Automatic detection and repair of errors in data structures. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 78–95, New York, NY, USA. 151

[Dey and Abowd, 2000] Dey, A. K. and Abowd, G. D. (2000). Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands. 2

[Dijkstra, 1968] Dijkstra, E. W. (1968). The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346. 1

[Douence et al., 2004] Douence, R., Fradet, P., and Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects. In Lieberherr, K., editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150, Lancaster, UK. ACM Press. 32

[Douence et al., 2005] Douence, R., Fradet, P., and Südholt, M. (2005). Trace-based aspects. In Filman, R. E., Elrad, T., Clarke, S., and Akşit, M., editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston. 14, 17, 108

[Douence et al., 2001] Douence, R., Motelet, O., and Südholt, M. (2001). A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 170–186, London, UK. Springer-Verlag. 14

[Dutchyn et al., 2006] Dutchyn, C., Tucker, D. B., and Krishnamurthi, S. (2006). Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239. 11, 29, 40, 42, 43, 50, 53, 144

[Dynarski et al., 2007] Dynarski, M., Agodini, R., Heaviside, S., Novak, T., Carey, N., Campuzano, L., Means, B., Murphy, R., Penuel, W., Javitz, H., Emery, D., and Sussex, W. (2007). *Effectiveness of reading and mathematics software products findings from the first student cohort : report*. DIANE Publishing. 130, 131

[Elrad et al., 2001] Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming. *Communications of the ACM*, 44(10). 29

[Eugster and Jayaram, 2009] Eugster, P. and Jayaram, K. (2009). EventJava: An extension of java for event correlation. In Drossopoulou, S., editor, *Proceedings of the 23rd European Conference on Object-oriented Programming (ECOOP 2009)*, number 5653 in Lecture Notes in Computer Science, Genova, Italy. Springer-Verlag. 2, 6, 7, 18, 72, 75, 80, 144

[Farkas, 2011] Farkas, A. (2011). A JavaScript library to block, abort, queue, and synchronize Ajax requests. 102, 106, 107, 124, 128

[Fleissner and Baniassad, 2007] Fleissner, S. and Baniassad, E. L. A. (2007). Epi-aspects: aspect-oriented conscientious software. In [OOPSLA 2007, 2007], pages 659–674. ACM SIGPLAN Notices, 42(10). 151

[Forgy, 1982] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/ many object pattern match problem. *Artificial Intelligence*, 19:17–37. 18, 80, 89

[Gabriel and Goldman, 2006] Gabriel, R. P. and Goldman, R. (2006). Conscientious software. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 433–450, Portland, Oregon, USA. ACM Press. ACM SIGPLAN Notices, 41(10). 151

[Gálvez et al., 2011] Gálvez, G., Cosmelli, D., Cubillos, L., Leger, P., Mena, A., Tanter, É., Flores, X., Luci, G., Montoya, S., and Soto Andrade, J. (2011). Estrategias cognitivas para el calculo mental. *Revista Latinoamericana de Investigación en Matemática Educativa*, 14(1):9–40. 131, 148

[Garg, 2002] Garg, Ph.D., V. K. (2002). *Elements of distributed computing*. John Wiley & Sons, Inc., New York, NY, USA. 102, 108

[Garrett, 2005] Garrett, J. J. (2005). Ajax: A new approach to Web applications. 34, 102, 103

[Gassanenko, 1993] Gassanenko, M. (1993). Context-oriented programming: Evolution of vocabularies. In *Proceedings of the euroFORTH conference (93)*, Marianske Lazne, Czech Republic. 1

[George et al., 2003] George, S., Evans, D., and Marchette, S. (2003). A biological programming model for self-healing. In *Proceedings of the 2003 ACM workshop on Survivable and Self-Regenerative Systems*, SSRS '03, pages 72–81, New York, NY, USA. ACM. 151

[Ghezzi et al., 2011] Ghezzi, C., Pradella, M., and Salvaneschi, G. (2011). An evaluation of the adaptation capabilities in programming languages. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 50–59, New York, NY, USA. ACM. 14

[Goldsmith et al., 2005] Goldsmith, S. F., O'Callahan, R., and Aiken, A. (2005). Relational queries over program traces. In [OOPSLA 2005, 2005], pages 385–402. ACM SIGPLAN Notices, 40(11). 2, 6, 7, 15, 19, 72, 75, 144

[Hayes, 2003] Hayes, B. (2003). The post-OOP paradigm. *American Scientist*, 91(2):106–110. 1

[Herzeel et al., 2006] Herzeel, C., Gybels, K., and Costanza, P. (2006). A temporal logic language for context awareness in pointcuts. In Thomas, D., editor, *Workshop on Revival of Dynamic Languages*, number 4067 in Lecture Notes in Computer Science, Nantes, France. Springer-Verlag. 2, 6, 7, 15, 18, 72, 75, 80, 88, 89, 144, 151

[Hirschfeld et al., 2008] Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology*, 7(3). 1

[Holzer et al., 2011] Holzer, A., Ziarek, L., Jayaram, K., and Eugster, P. (2011). Putting events in context: aspects for event-based distributed programming. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, pages 241–252, Porto de Galinhas, Brazil. ACM Press. 84

[Humax, 2011] Humax (2011). Humax. A JavaScript framework for aspect-oriented programming. http://humax.sourceforge.net/. 29, 52

[Inostroza et al., 2011] Inostroza, M., Tanter, É., and Bodden, E. (2011). Join point interfaces for modular reasoning in aspect-oriented programs. In *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), New Ideas track*, pages 508–511, Szeged, Hungary. ACM Press. 39

[Jiang et al., 2010] Jiang, J., Wilson, C., Wang, X., Huang, P., Sha, W., Dai, Y., and Zhao, B. Y. (2010). Understanding latent interactions in online social networks. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 369–382, New York, NY, USA. ACM. 106

[jQuery, 2011] jQuery (2011). jQuery. A JavaScript library to manage event handling, animating, and Ajax interactions for the Web development. http://jquery.com/. 50, 52

[Kebritchi et al., 2010] Kebritchi, M., Hirumi, A., and Bai, H. (2010). The effects of modern mathematics computer games on mathematics achievement and class motivation. *Computer in Education*, 55:427–443. 130, 131

[Kiczales, 1992] Kiczales, G. (1992). Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA 92 Workshop on Reflection and Metalevel Architectures*. Akinori Yonezawa and Brian C. Smith, editors. 23

[Kiczales et al., 1993] Kiczales, G., Ashley, J. M., Rodriguez, L., Vahdat, A., and Bobrow, D. G. (1993). Metaobject protocols: Why we want them and what else they can do. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101–118. MIT Press. 4

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary. Springer-Verlag. 11, 12, 29, 39, 58, 73, 91

[Kiczales et al., 1996] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., and Mendhekar, A. (1996). Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al. 56

[Kiczales et al., 1997a] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., and Murphy, G. (1997a). Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)*, pages 481–490, Boston, Massachusetts, USA. ACM Press. 3, 23, 24, 73, 78

[Kiczales et al., 1997b] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997b). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland. Springer-Verlag. 2, 9, 10, 11, 102, 108

[Kossinets et al., 2008] Kossinets, G., Kleinberg, J., and Watts, D. (2008). The structure of information pathways in a social communication network. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 435–443, New York, NY, USA. ACM. 128

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565. 71, 102, 104, 108, 109, 111, 147

[Leger et al., 2011a] Leger, P., Gálvez, G., Cubillos, L., Cosmelli, D., Inostroza, M., Tanter, É., Luci, G., and Andrade, J. S. (2011a). Ecocam, un sistema computacional adaptable al contexto para promover estrategias de cálculo mental: un diseño y estudio de casos. Technical Report TR/DCC-2011-11, University of Chile. 130, 147, 148

[Leger and Tanter, 2010a] Leger, P. and Tanter, É. (2010a). An open trace-based mechanism. In Aldrich, J. and Massa, R., editors, *Proceedings of the 14th Brazilian Symposium on Programming Languages (SBLP 2010)*, pages 123–138, Salvador - Bahia, Brazil. 71, 145

[Leger and Tanter, 2010b] Leger, P. and Tanter, É. (2010b). Towards an open trace-baced mechanism. In Leavens, G. T., Katz, S., and Mezini, M., editors, *Proceedings of the Ninth Workshop on Foundations of Aspect-Oriented Languages (FOAL 2010)*, pages 25–30, Rennes and Saint Malo, France. 55, 144

[Leger and Tanter, 2012] Leger, P. and Tanter, É. (2012). A self-replication algorithm to flexibly match execution traces. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, pages 27–32, Potsdam, Germany. ACM Press. 80, 97, 146

[Leger et al., 2011b] Leger, P., Tanter, É., and Douence, R. (2011b). Modular and flexible causality control on the web. Technical Report TR/DCC-2011-9, University of Chile. 101, 147

[Lerner et al., 2010] Lerner, B. S., Venter, H., and Grossman, D. (2010). Supporting dynamic, third-party code customizations in javascript using aspects. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, volume 45, pages 361–376. ACM. 41

[Leroy et al., 2011] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. (2011). Language Objective Caml: Caml supports functional, imperative, and object-oriented programming styles. http://caml.inria.fr/. 53

[Lethielleux, 2001] Lethielleux, C. (2001). *Le calcul mental*. Bordas Bordas. 131, 132

[Lieberherr and Silva-Lepe, 1994] Lieberherr, K. and Silva-Lepe, I. (1994). Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101. 11

[Luckham, 2001] Luckham, D. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 4

[Maes, 1987] Maes, P. (1987). *Computional reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium. 24

[Martin et al., 2005] Martin, M., Livshits, B., and Lam, M. S. (2005). Finding application errors and security flaws using PQL: a program query language. In [OOPSLA 2005, 2005], pages 365–383. ACM SIGPLAN Notices, 40(11). 2, 6, 7, 15, 19, 72, 75, 144

[Mashable, 2010] Mashable (2010). Website for news in social and digital media, technology and web culture. 106

[Masuhara et al., 2002] Masuhara, H., Kiczales, G., and Dutchyn, C. (2002). Compilation semantics of aspect-oriented programs. In *AOSD Workshop on Foundations of Aspect-Oriented Languages*, pages 17–26. 10

[Masuhara et al., 2003] Masuhara, H., Kiczales, G., and Dutchyn, C. (2003). A compilation and optimization model for aspect-oriented programs. In Hedin, G., editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag. 11, 41, 62, 150

[Masuhara et al., 2005] Masuhara, H., Tatsuzawa, H., and Yonezawa, A. (2005). Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 320–330, Tallinn, Estonia. ACM. 53

[Mattern, 1989] Mattern, F. (1989). Virtual time and global states of distributed systems. In et al., C. M., editor, *Proceding Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier. 102, 108, 111

[McEachen and Alexander, 2005] McEachen, N. and Alexander, R. T. (2005). Distributing classes with woven concerns – an exploration of potential fault scenarios. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 192–200, Chicago, Illinois, USA. ACM Press. 43

[Meredith et al., 2011] Meredith, P. O., Jin, D., Griffith, D., Chen, F., and Roşu, G. (2011). An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*. To appear. 2, 20, 72, 90

[Meyerovich et al., 2009] Meyerovich, L., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2009)*, Orlando, Florida, USA. ACM Press. 35

[Mezini and Ostermann, 2004] Mezini, M. and Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th Symposium on Foundations of Software Engineering (FSE-12)*, pages 127–136, Newport Beach, CA, USA. 42

[Might et al., 2010] Might, M., Smaragdakis, Y., and Van Horn, D. (2010). Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 305–315, New York, NY, USA. ACM. 129

[Milner et al., 1990] Milner, R., Tofte, M., and Harper, R. (1990). *The definition of Standard ML*. MIT Press, Cambridge, MA, USA. 53

[Murty and Garg, 1997] Murty, V. and Garg, V. (1997). Characterization of message ordering specifications and protocols. In *17th IEEE International Conference on Distributed Computing Systems (ICDCS '97)*, pages 492–499, Los Alamitos, CA, USA. IEEE Computer Society. 102, 104, 108, 110, 147

[Navarro et al., 2006] Navarro, L. D. B., Südholt, M., Vanderperren, W., De Fraine, B., and Suvée, D. (2006). Explicitly distributed aop using awed. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 51–62, Bonn, Germany. ACM Press. 19

[Neumann, 1966] Neumann, J. V. (1966). *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA. 80, 146

[Núñez et al., 2009] Núñez, A., Noyé, J., and Gasiūnas, V. (2009). Declarative definition of contexts with polymorphic events. In *International Workshop on Context-Oriented Programming*, pages 2:1–2:6, New York, NY, USA. ACM. 39

[OOPSLA 2005, 2005] OOPSLA 2005 (2005). *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA. ACM Press. ACM SIGPLAN Notices, 40(11). 159, 161, 163

[OOPSLA 2007, 2007] OOPSLA 2007 (2007). *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, Montreal, Canada. ACM Press. ACM SIGPLAN Notices, 42(10). 160, 161

[O'Reilly, 2005] O'Reilly, T. (2005). What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications & Strategies*. 102, 104

[Ossher and Tarr, 2001] Ossher, H. L. and Tarr, P. L. (2001). Multi-dimensional separation of concerns and the hyperspace approach. In Akşit, M., editor, *Software Architectures and Component Technology*, volume 648 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer. 11

[Ostermann et al., 2005] Ostermann, K., Mezini, M., and Bockisch, C. (2005). Expressive pointcuts for increased modularity. In Black, A. P., editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag. 2, 6, 7, 15, 18, 72, 75, 80, 144, 151

[Parnas, 1972] Parnas, D. (1972). On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058. 1

[Postel and Reynolds, 1985] Postel, J. and Reynolds, J. (1985). File transfer protocol (ftp). request for comments 959. 19

[Prototype, 2011] Prototype (2011). Prototype. A JavaScript library that aims to ease development of dynamic Web applications. http://www.prototypejs.org/. 29, 52

[Rajan and Leavens, 2008] Rajan, H. and Leavens, G. T. (2008). Ptolemy: A language with quantified, typed events. In Vitek, J., editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 155–179, Paphos, Cyprus. Springer-Verlag. 11, 29, 38, 39, 40, 117, 144

[Rajan and Sullivan, 2003] Rajan, H. and Sullivan, K. (2003). Eos: Instance-level aspects for integrated system design. In *Proceedings of ESEC/FSE 2003*, pages 297–306, Helsinki, Finland. 42

[Rao, 1991] Rao, R. (1991). Implementational reflection in Silica. In America, P., editor, *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP 91)*, volume 512 of *Lecture Notes in Computer Science*, pages 251–266, Geneva, Switzerland. Springer-Verlag. xii, 24

[Razek et al., 2003] Razek, M. A., Frasson, C., and Kaltenbach, M. (2003). A context-based information agent for supporting education on the web. In *Proceedings of the 2003 international conference on Computational science and its applications: PartI*, ICCSA'03, pages 170–179. Springer-Verlag. 13

[Rico, 2011] Rico (2011). A JavaScript library for rich internet applications. 102, 106, 127

[Satyanarayanan, 2001] Satyanarayanan, M. (2001). Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10 –17. 1, 13, 131

[Schilit et al., 1994] Schilit, B., Adams, N., and Want, R. (1994). Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90, Washington, DC, USA. IEEE Computer Society. 12

[Shahin, 2006] Shahin (2006). Ajax Video Player, streaming over HTTP with Javascript and Ajax. 123

[Shrager and Siegler, 1998] Shrager, J. and Siegler, R. S. (1998). Scads: a model of children's strategy choices and strategy discoveries. *Psychological Science*, 9:405. 132, 148

[Stamey et al., 2005] Stamey, J., Saunders, B., and Blanchard, S. (2005). The aspect-oriented Web. In *Proceedings of the 23rd annual international Conference on Design of Communication: documenting & designing for pervasive information*, pages 89–95, Coventry, United Kingdom. ACM. 29, 52

[Strang and Linnhoff-popien, 2004] Strang, T. and Linnhoff-popien, C. (2004). A context modeling survey. 5

[Tanter, 2008a] Tanter, É. (2008a). Controlling aspect reentrancy. *Journal of Universal Computer Science*, 14(21):3498–3516. Best Paper Award of the Brazilian Symposium on Programming Languages (SBLP 2008). 29, 45, 144

[Tanter, 2008b] Tanter, É. (2008b). Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium. ACM Press. 29, 37, 43, 44, 66, 144

[Tanter, 2009] Tanter, É. (2009). Beyond static and dynamic scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*, pages 3–14, Orlando, FL, USA. ACM Press. 29, 37, 44

[Tanter, 2010] Tanter, É. (2010). Execution levels for aspect-oriented programming. In [AOSD 2010, 2010], pages 37–48. Best Paper Award. 45, 46, 144

[Tanter et al., 2009] Tanter, É., Fabry, J., Douence, R., Noyé, J., and Südholt, M. (2009). Expressive scoping of distributed aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 27–38, Charlottesville, Virginia, USA. ACM Press. 44

[Tanter et al., 2006] Tanter, É., Gybels, K., Denker, M., and Bergel, A. (2006). Context-aware aspects. In Löwe, W. and Südholt, M., editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 227–242, Vienna, Austria. Springer-Verlag. 1, 2

[Taton, 1953] Taton, R. (1953). *Le calcul mental*. Que sais-je. Presses Universitaires de France. 131, 132, 148

[Tobin-Hochstadt and Felleisen, 2008] Tobin-Hochstadt, S. and Felleisen, M. (2008). The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 395–406, San Francisco, CA, USA. ACM Press. 3, 6, 55, 61, 71, 145, 154

[Toledo et al., 2010] Toledo, R., Leger, P., and Tanter, É. (2010). AspectScript: Expressive aspects for the Web. In [AOSD 2010, 2010], pages 13–24. 28, 143

[Toledo and Tanter, 2011a] Toledo, R. and Tanter, É. (2011a). Access control in JavaScript. *IEEE Software*, 28(5):76–84. 144

[Toledo and Tanter, 2011b] Toledo, R. and Tanter, É. (2011b). Secure and modular access control with aspects. Technical Report TR/DCC-2011-7, University of Chile. 144

[Tucker and Krishnamurthi, 2003] Tucker, D. B. and Krishnamurthi, S. (2003). Pointcuts and advice in higher-order languages. In Akşit, M., editor, *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 158–167, Boston, MA, USA. ACM Press. 29

[Walker and Viggers, 2004] Walker, R. J. and Viggers, K. (2004). Implementing protocols via declarative event patterns. *SIGSOFT Softw. Eng. Notes*, 29(6):159–169. 19

[Washizaki et al., 2009] Washizaki, H., Kubo, A., Mizumachi, T., Eguchi, K., Fukazawa, Y., Yoshioka, N., Kanuka, H., Kodaka, T., Sugimoto, N., Nagai, Y., and Yamamoto, R. (2009). AOJS: aspect-oriented JavaScript programming framework for web development. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 31–36, Charlottesville, Virginia, USA. ACM. 29, 52

[Wasserman and Faust, 1994] Wasserman, S. and Faust, K. (1994). *Social network analysis: methods and applications.* Structural analysis in the social sciences, 8. Cambridge University Press, 1 edition. 106

[Wiese, 2003] Wiese, H. (2003). *Numbers, Language, and the Human Mind.* Cambridge University Press. 134

[Wikipedia, 2011] Wikipedia (2011). Social network analysis software. 102, 107, 128

[Wilson et al., 2009] Wilson, C., Boe, B., Sala, A., Puttaswamy, K. P., and Zhao, B. Y. (2009). User interactions in social networks and their implications. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 205–218, New York, NY, USA. ACM. 106

[Wirth, 1974] Wirth, N. (1974). On the design of programming languages. *Information Processing 74*. 23

[Zita et al., 2002] Zita, K. H., Phelps, J., and Geib, C. W. (2002). An open agent architecture for assisting elder independence. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 578–586. ACM. 13