An Efficient and Flexible Implementation of Aspect-Oriented Languages

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Diplom-Informatiker Christoph-Matthias Bockisch

geboren in Bielefeld

Referentin:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. dr. ir. Mehmet Akşit
Datum der Einreichung:	14. Mai 2008
Datum der mündlichen Prüfung:	1. Juli 2008

Erscheinungsjahr 2009

Darmstadt D17

Abstract

Compilers for modern *object-oriented programming languages* generate code in a platform independent intermediate language [LY99, CLI06] preserving the concepts of the source language; for example, classes, fields, methods, and virtual or static dispatch can be directly identified within the intermediate code. To execute this intermediate code, state-of-the-art implementations of virtual machines perform *just-in-time* (JIT) compilation [DS84, ACL+99, Ayc03] of the intermediate language; i.e., the virtual instructions in the intermediate code are compiled to native machine code at runtime. In this step, a declarative representation of source language concepts in the intermediate language facilitates highly efficient adaptive and speculative optimization of the running program which may not be possible otherwise.

In contrast, constructs of *aspect-oriented languages*—which improve the separation of concerns—are commonly realized by compiling them to conventional intermediate language instructions or by driving transformations of the intermediate code, which is called weaving. This way the aspect-oriented constructs' semantics is not preserved in a declarative manner at the intermediate language level. This representational gap between aspect-oriented concepts in the source code and in the intermediate code hinders high performance optimizations and weakens features of software engineering processes like debugging support or the continuity property of incremental compilation: modifying an aspect in the source code potentially requires re-weaving multiple other modules [SDR08, RDHN06].

To leverage language implementation techniques for aspect-oriented languages, this thesis proposes the Aspect-Language Implementation Architecture (ALIA) which prescribes—amongst others—the existence of an intermediate representation preserving the aspect-oriented constructs of the source program. A central component of this architecture is an extensible and flexible meta-model of aspect-oriented concepts which acts as an interface between front-ends (usually a compiler) and back-ends (usually a virtual machine) of aspect-oriented language implementations. The architecture

and the meta-model are embodied for Java-based aspect-oriented languages in the *Framework for Implementing Aspect Languages* (FIAL) respectively the *Language-Independent Aspect Meta-Model* (LIAM) which is part of the framework. FIAL generically implements the work flows required from an execution environment when executing aspects provided in terms of LIAM. In addition to the first-class intermediate representation of aspect-oriented concepts, ALIA—and the FIAL framework as its incarnation—treat the points of interaction between aspects and other modules—so-called join points—as being late-bound to an implementation. In analogy to the object-oriented terminology for late-bound methods, the join points are called *virtual* in ALIA. Together, the first-class representation of aspect-oriented concepts in the intermediate representation as well as treating join points as being virtual facilitate the implementation of new and effective optimizations for aspect-oriented programs.

Three different instantiations of the FIAL framework are presented in this thesis, showcasing the feasibility of integrating language back-ends with different characteristics with the framework. One integration supports static aspect deployment and produces results similar to conventional aspect weavers; the woven code is executable on any standard Java virtual machine. Two instantiations are fully dynamic, where one is realized as a portable plug-in for standard Java virtual machines and the other one, called STEAMLOOM$^{\text{ALIA}}$, is realized as a deep integration into a specific virtual machine, the Jikes Research Virtual Machine [AAB+05]. While the latter instantiation is not portable, it exhibits an outstanding performance.

Virtual join point dispatch is a generalization of virtual method dispatch. Thus, well established and elaborate optimization techniques from the field of virtual method dispatch are re-used with slight adaptations in STEAMLOOM$^{\text{ALIA}}$. These optimizations for aspect-oriented concepts go beyond the generation of optimal bytecode. Especially strikingly, the power of such optimizations is shown in this thesis by the examples of the **cflow** dynamic property [ACH+05b], which may be necessary to evaluate during virtual join point dispatch, and dynamic aspect deployment [MO03]—i.e., the selective modification of specific join points' dispatch.

In order to evaluate the optimization techniques developed in this thesis, a means for benchmarking has been developed in terms of macro-benchmarks; i.e., real-world applications are executed. These benchmarks show that for both concepts the implementation presented here is at least circa twice as fast as state-of-the-art implementations performing static optimizations of the generated bytecode; in many cases this thesis's optimizations even reach a speed-up of two orders of magnitude for the **cflow** implementation and even four orders of magnitude for the dynamic deployment.

The intermediate representation in terms of LIAM models is general enough to express the constructs of multiple aspect-oriented languages. Therefore, optimizations of features common to different languages are available to applications written in all of them. To proof that the abstractions provided by LIAM are sufficient to act as intermediate language for multiple aspect-oriented source languages, an automated translation from source code to LIAM models has been realized for three very different and popular aspect-oriented languages: AspectJ [KHH+01], JAsCo [VSV+05] and Compose* [BA01, dRHH+08]. In addition, the feasibility of translating from CaesarJ [AGMO06] to LIAM models is shown by discussion. The use of an extensible meta-model as intermediate representation furthermore simplifies the definition of new aspect-oriented language concepts as is shown in terms of a tutorial-style example of designing a domain specific extension to the Java language in this thesis.

Zusammenfassung

Compiler für moderne *Objekt-orientierte Programmiersprachen* generieren Code in einer Plattform-unabhängigen Intermediate-Sprache [LY99, CLI06], die die Konzepte der Quell-Sprache erhält; zum Beispiel können Klassen, Felder, Methoden und virtueller oder statischer Methodendispatch direkt im Intermediate-Code identifiziert werden. Um diesen Intermediate-Code auszuführen, führen aktuelle Implementierungen von virtuellen Maschinen sogenannte *just-in-time* (JIT) Compilierung der Intermediate-Sprache durch [DS84, ACL+99, Ayc03]; das bedeutet, dass die virtuellen Instruktionen im Intermediate-Code zur Laufzeit zu nativem Maschinencode compiliert werden. In diesem Schritt ermöglicht eine deklarative Repräsentation der Konzepte aus der Quell-Sprache in der Intermediate-Sprache höchst effiziente adaptive und spekulative Optimierungen des laufenden Programms, die sonst nicht möglich wären.

Im Gegensatz hierzu werden die Konstrukte von *Aspekt-orientierten Programmiersprachen* – die zu einer besseren "Separation of Concerns" führen – üblicherweise dadurch realisiert, dass sie zu herkömmlichen Intermediate-Instruktionen compiliert werden oder dass sie Transformationen im Intermediate-Code bewirken; dies wird "Weben" genannt. Auf diese Weise bleibt die Semantik Aspekt-orientierter Konstrukte auf der Ebene der Intermediate-Sprache nicht deklarativ erhalten. Durch diese Kluft in der Darstellung Aspekt-orientierter Konzepte im Quell-Code und im Intermediate-Code werden hoch-performante Optimierungen behindert und Eigenschaften eines Software-Entwicklungsprozesses geschwächt, wie Unterstützung beim Debuggen oder die Continuity-Eigenschaft von inkremeteller Compilierung: Eine Modifikation an einem Aspekt im Quell-Code zieht möglicherweise das erneute Weben mehrerer anderer Module nach sich [SDR08, RDHN06].

Um die Sprach-Implementierungstechniken für Aspekt-orientierte Programmiersprachen zu verbessern, wird in dieser Arbeit die Architektur für Aspekt-Sprachen-Implementierungen (englisch Aspect-Language Implementation Architecture, ALIA) vorgeschlagen, welche unter anderem vorschreibt, dass eine Intermediate-Repräsentation existiert, die die Aspekt-orientierten

Konstrukte aus dem Quell-Programm erhält. Eine zentrale Komponente dieser Architektur ist das erweiterbare und flexible Meta-Model von Aspekt-orientierten Konzepten, das als Schnittstelle zwischen Front-Ends (üblicherweise ein Compiler) und Back-Ends (üblicherweise eine virtuelle Maschine) von Aspekt-orientierten Sprachimplementierungen dient. Die Architektur und das Meta-Model sind für Java-basierte Sprachen in dem Framework zum Implementieren von Aspekt-Sprachen (englisch Framework for Implementing Aspect Languages, FIAL) beziehungsweise dem Sprachunabhängigen Aspekt-Meta-Model (englisch Language-Independent Aspect Meta-Model, LIAM), das Teil des Frameworks ist, verkörpert. FIAL implementiert generisch Arbeitsabläufe, die von einer Ausführungsumgebung benötigt werden, welche Aspekte ausführt, die mit LIAM definiert sind. Zusätzlich zu der deklarativen Intermediate-Repräsentation von Aspekt-orientierten Konzepten behandelt ALIA – und FIAL als dessen Verkörperung – Join-Points – also Punkte an denen Interaktion zwischen Aspekten und anderen Modulen stattfindet – so, dass sie an eine Implementierung spät-gebunden werden. Analog zu der Objekt-orientierten Terminologie für spät-gebundene Methoden werden Join-Points in ALIA als *virtuell* bezeichnet. Die deklarative Repräsentation von Aspekt-orientierten Konzepten in der Intermediate-Repräsentation und das Behandeln von Join-Points als virtuell ermöglichen es, neue und effektive Optimierungen für Aspekt-orientierte Programme zu entwickeln.

In dieser Arbeit werden drei Instantiierungen von FIAL vorgestellt, die herausstellen, dass eine Integration von Sprach-Back-Ends mit unterschiedlichen Eigenschaften mit dem Framework möglich sind. Eine der Integrierungen unterstützt statisches Aspekt-Deployment und erzeugt Ergebnisse, die vergleichbar mit denen herkömmlicher Weber sind; der gewobene Code kann auf jeder standard-Java Virtual Machine ausgeführt werden. Zwei Instantiierungen sind vollständig dynamisch, wobei eine als portables Plug-in für standard-Java Virtual Machines realisiert ist und die andere als tiefgehende Integration in eine spezielle virtuelle Maschine, die Jikes Research Virtual Machine [AAB+05]. Während STEAMLOOM$^{\text{ALIA}}$, die letztere Instantiierung, nicht portabel ist, ist sie besonders leistungsstark.

Der Dispatch von virtuellen Join-Points ist eine Verallgemeinerung des Dispatchs von virtuellen Methoden. Daher werden in STEAMLOOM$^{\text{ALIA}}$ etablierte und ausgefeilte Optimierungstechniken aus dem Bereich des virtuellen Methodendispatchs wiederverwendet. Diese Optimierungen für Aspekt-orientierte Konzepte gehen über die Erzeugung von optimalem Bytecode hinaus. Die Mächtigkeit solcher Optimierungen wird in dieser Arbeit besonders offensichtlich an den Beispielen der dynamischen Eigenschaft **cflow** [ACH+05b], die bei der Auswertung eines virtuellen Join-Point-Dispatchs benötigt werden

kann, und dynamischem Aspekt-Deployment [MO03] – das heißt die selektive Modifikation des Dispatchs von bestimmten Join-Points.

Um diese Optimierungstechniken zu evaluieren, wurde in dieser Arbeit ein Benchmark-Verfahren entwickelt, in dem Makro-Benchmarks, also reale Applikationen, verwendet werden. Diese Benchmarks zeigen, dass für beide Konzepte die hier präsentierte Implementierung mindestens etwa doppelt so schnell ist wie aktuelle Implementierungen, die den generierten Bytecode statisch optimieren; in vielen Fällen erreichen die in dieser Arbeit vorgestellten Optimierungen sogar eine Beschleunigung von zwei Größenordnungen für die **cflow**-Implementierung und sogar vier Größenordnungen für das dynamische Deployment.

Die Verwendung von LIAM-Modellen als Intermediate-Repräsentation ist allgemein genug, um Aspekt-orientierte Konstrukte mehrerer Sprachen auszudrücken. Daher profitieren alle Sprachen davon, wenn für gemeinsame Konzepte Optimierungen implementiert werden. Um zu zeigen, dass die Abstraktionen von LIAM ausreichend sind, um als Intermediate-Sprache für mehrere Aspekt-Orientierte Quell-Sprachen zu dienen, wurden automatische Übersetzer von Quell-Code zu LIAM-Modellen für drei sehr verschiedene aber populäre Aspekt-orientierte Sprachen entwickelt: AspectJ [KHH+01], JAsCo [VSV+05] und Compose* [BA01, dRHH+08]. Zusätzlich wird die Machbarkeit, von CaesarJ [AGMO06] zu LIAM-Modellen zu übersetzen, diskutiert. Die Verwendung eines erweiterbaren Meta-Modells als Intermediate-Repräsentation vereinfacht weiterhin die Definition von neuen Aspekt-orientierten Sprach-Konzepten, wie in dieser Arbeit durch ein Tutorial-artiges Beispiel gezeigt wird, in dem eine Domänen-spezifische Erweiterung der Sprache Java entwickelt wird.

Contents

1 Overview **1**
- 1.1 Context of this Thesis . . . 1
- 1.2 Introduction . . . 4
  - 1.2.1 Virtual Join Points . . . 5
  - 1.2.2 Aspect-Oriented Language Implementation Architecture 7
  - 1.2.3 VM Integration of AO Concepts . . . 11
  - 1.2.4 Benchmarks . . . 12
- 1.3 Summary of Contributions . . . 13
- 1.4 Structure of this Thesis . . . 16

2 Virtual Join Points **19**
- 2.1 Background . . . 20
  - 2.1.1 Pointcut-and-advice Languages . . . 20
  - 2.1.2 The Java Bytecode Format . . . 23
- 2.2 Binding of Join Points . . . 26
- 2.3 Virtual Join Points . . . 29
- 2.4 Prototype Implementation of Envelopes . . . 34
  - 2.4.1 Generating Envelopes . . . 34
  - 2.4.2 Weaving in Envelopes . . . 35
  - 2.4.3 Limitations of the Prototype . . . 37
  - 2.4.4 Dynamic Weaving in Envelopes . . . 39
- 2.5 Evaluation of Weaving Approaches . . . 40
  - 2.5.1 Evaluated Approaches . . . 40
  - 2.5.2 Benchmarks . . . 41

3 The Aspect Language Implementation Architecture **45**
- 3.1 Common Features of Aspect-Oriented Languages . . . 46
  - 3.1.1 AspectJ . . . 47
  - 3.1.2 CaesarJ . . . 52
  - 3.1.3 JAsCo . . . 53
  - 3.1.4 Compose* . . . 54
  - 3.1.5 Summary of language features . . . 56

3.2 The Language Independent Aspect Meta-Model 56
3.3 The Framework for Implementing Aspect Languages 62
3.3.1 FIAL's Execution Model 63
3.3.2 Work Flows 64
3.3.3 Weaving Directives 65
3.4 Language Mappings 68
3.4.1 AspectJ Mapping 70
3.4.2 JAsCo Mapping 72
3.4.3 Compose* Mapping 74
3.4.4 Discussion 78
3.5 Execution Environments 79
3.5.1 Envelope-Based Reference Implementation 80
3.5.2 Static Weaver 82
3.6 AO Language Design and Implementation with FIAL 83
3.6.1 Realizing the Sample Language in FIAL 86

4 Optimizing AO Concepts in the Virtual Machine 93
4.1 Motivation for Dynamic Optimizations of AO Concepts 94
4.2 Dynamic Optimizations of OO Concepts 95
4.3 Optimized Dynamic Aspect Deployment 102
4.3.1 Eager versus Lazy Envelope Call Insertion 104
4.3.2 Speculative Inlining Techniques for Envelopes 107
4.3.3 Special Language Features 111
4.4 Control Flow Quantification 112
4.4.1 Current Implementation of Control Flow Quantification 113
4.4.2 Control Flow Guards 116
4.4.3 Implementation of Control Flow Guards 120

5 Evaluating Dynamic Optimizations of AO Concepts 125
5.1 Benchmarks for Dynamic Features of AO Languages 126
5.2 Evaluating Optimized Dynamic Deployment 127
5.2.1 Approaches Participating in the Evaluation 127
5.2.2 Alternative Configurations of Envelope-Aware Jikes . . 129
5.2.3 Results of Deployment Evaluation 132
5.3 Evaluating Control Flow Quantification 135
5.3.1 Implementations of **cflow** in the `ajc` and `abc` Compilers 138
5.3.2 Micro-Benchmarks 140
5.3.3 Benchmarks Based on SPEC JVM98 142
5.3.4 `abc` Benchmark Suite 143

6 Related Work 147

6.1 Virtual Join Points 147
6.1.1 Prototype- and Delegation-Based Execution Model . . 147
6.1.2 Reflection-Based Execution Model 149
6.2 Meta-Models for Aspect-Oriented Concepts 149
6.2.1 Metaspin and JAMI 150
6.2.2 Reflex 150
6.2.3 Aspect Sand Box 151
6.3 Intermediate Languages and Execution Environments 151
6.3.1 Nu 152
6.3.2 Lightweight VM Support for AspectJ 153

7 Conclusions and Future Work **155**
7.1 Conclusions 155
7.2 Future Work 158
7.2.1 Optimizing the Dispatch Function 158
7.2.2 Virtual Machine as FIAL Instantiation 160
7.2.3 Static Crosscutting 161
7.2.4 IDE Support for Programs Executed on FIAL 162

Scientific Career **165**

Bibliography **167**

List of Figures

1.1 The most important LIAM entities. 8
1.2 Overview of different usages of FIAL. 9

2.1 Accessing a database from the client as well as the server. . . 22
2.2 The Java Virtual Machine class file format. 24
2.3 Virtual join point dispatch. 31

3.1 Compilation and execution of AO programs in ALIA. 47
3.2 Schema of how filters are applied in Compose*. 55
3.3 Entities of the Language Independent Meta-Model. 57
3.4 Example of a model in LIAM. 58
3.5 Runtime object model of JoinPoint for the ThisJoinPointContext. 59
3.6 Object model of ThisJoinPointContext context in LIAM. 60
3.7 Interaction between FIAL and its instantiation. 63
3.8 Object diagram of a general advice unit. 64
3.9 Evaluation strategy of a dispatch function as a BDD. 66
3.10 Object diagram of a BoundAction. 67
3.11 Data structure for representing the order of advice. 67
3.12 Object diagram of an ActionOrderElement data structure. 68
3.13 Linking of pointcut-and-advice and advised locations in AJDT. 79
3.14 Instruction sequence generated for ordered BoundActions. . . . 82
3.15 Class diagram of the decorator pattern's structure. 84
3.16 Class managing dependencies of decoratees and decorators. . . 87
3.17 Structure of the aspect for enforcing the decorator pattern. . . 87
3.18 Advice unit for updating the decorator-decoratee mapping. . . 88
3.19 Advice unit for forwarding calls on decorated objects. 89

4.1 Class with a monomorphic method. 97
4.2 Class with a polymorphic method. 99
4.3 Inline sequence without and with envelopes. 106
4.4 Diagram of the code splitting algorithm using merge points. . 109
4.5 Stack frame layout for baseline-compiled methods. 121

5.1 Schema of executing benchmarks with dynamic deployment. . 132

List of Tables

2.1 Results of the measurements of static weaving. 43
2.2 Results of the measurements of dynamic weaving. 44

5.1 Percent overhead of using eager envelopes in Jikes RVM. . . . 130
5.2 Time for deploying an aspect on SPEC JVM98 benchmarks. . 134
5.3 Steady-state performance in the SPEC JVM98 benchmark. . . 136
5.4 Start-up performance in the SPEC JVM98 benchmarks. 137
5.5 Start-up performance after deployment of an aspect. 137
5.6 Overhead in percent measured by the micro-benchmarks. . . . 141
5.7 Overhead in SPEC JVM98 benchmarks. 143
5.8 Overhead in the `figure` and `quicksort` benchmarks. 145

List of Listings

1.1 Pointcut-and-advice for validating client SQL queries. 3
1.2 Residual dispatch code for a join point shadow. 4
1.3 Pseudo code generated for a join point shadow by ERIN. . . . 10
2.1 Example Services class of the base program. 21
2.2 Aspect checking database accesses from untrusted contexts. . . 21
2.3 A pointcut-and-advice using **args** as dynamic property. 23
2.4 Bytecode instructions calling a method. 25
2.5 Bytecode instructions accessing the method's arguments. . . . 25
2.6 Compilation and weaving result of AspectJ. 26
2.7 Pointcut-and-advice referring to dynamic properties. 30
2.8 Compilation and weaving result in the envelopes approach. . . 32
2.9 Bytecode of a plain envelope. 34
2.10 Bytecode of an envelope with residual dispatch. 36
2.11 Envelope with residual dispatch for two pointcut-and-advice. . 36
2.12 Difference between **call** and **execution** in AspectJ. 38
3.1 Code containing different kinds of join point shadows. 48
3.2 Pointcut-and-advice using pointcut parameters. 49
3.3 AspectJ code using a precedence declaration. 50
3.4 Output of executing the sample program in Listing 3.3. 51
3.5 AspectJ code declaring strategy for aspect instantiation. . . . 52
3.6 CaesarJ code performing thread-local aspect deployment. . . . 52
3.7 JAsCo code declaring precedence of pointcut-and-advice. . . . 53
3.8 Aspect in AspectJ syntax. 56
3.9 Woven code for class Services and aspect QueryInjectionChecker. . 70
3.10 Example of logging implemented in the JAsCo language. . . . 72
3.11 Java class with annotations representing an instantiated **hook**. 73
3.12 Bypassing the decorator. 84
3.13 Proper use of the decorator pattern. 85
3.14 Proper use of the decorator pattern in the sample language. . 86
3.15 Implementation of DecoratorInstantiationStrategy. 90
3.16 Implementation if DecoratedDynamicProperty. 90
3.17 Implementation if FromDecoratorDynamicProperty. 91

4.1 Code calling a virtual method. 97
4.2 Guarded inlined method. 98
4.3 Guarded inlined method with code splitting. 100
4.4 Speculative inlining with on-stack-replacement. 102
4.5 A pointcut *or* clause containing a **cflow** pointcut designator. . 113
4.6 Implementation of Fibonacci numbers with an aspect. 118
4.7 Woven pseudo code using **cflow** word. 119
5.1 Aspect used by the modified SPEC JVM98 benchmark suite. . 133
5.2 Micro-measurement harness. 139
5.3 Pointcut-and-advice for figure benchmark. 144
5.4 Pointcut-and-advice for quicksort benchmark. 145

Preface

For a PhD assistant, writing papers and getting them accepted for publication is always a challenge. You tend to improve the practical work until the submission deadline is just around the corner, which, while the practical work matures, leaves you with little time for actually writing the paper. This has not been different for me when writing the *big paper*—with the difference that the *submission deadline* is at least partly negotiable which requires a larger amount of self-control. At this place, I would like to thank the *chairs of my program committee*—Prof. Dr.-Ing. Mira Mezini, my supervisor, and Prof. dr. ir. Mehmet Akşit, my second examiner—for accepting the *big paper*, but even more for their support of my work.

In 2000 a course taught by Mira was among one of my first chosen courses and obviously influenced my decision to make my Diploma in her group and afterwards stay as a PhD assistant. During my whole time in her group, she always guided and advised me; but similarly important she also gave me freedom to evolve freely. Not to forget, co-authoring papers with her taught me to take a bird's eye view and to write well structured scientific texts. From her I also learned that re-ordering a paper's sections last minute to improve the structure can actually work. Mehmet accompanies my work since 2006 when I joined the AOSD-Europe project where his group is one of the partners. I often discussed my work with Mehmet and his group which has widened-up my work's focus; funnily, they usually had to follow my presentations twice for some reason or the other. I also like to thank Mehmet for the very encouraging feedback on my PhD thesis.

Like in all my other publications, there are several people who helped me in achieving the results written down in the *big paper*. First of all, I like to thank all my colleagues. Each of them has provided valuable comments in discussions; some of them even in more practical ways. Michael Haupt was the adviser of my Diploma thesis which formed the basis of further joint work and finally my PhD thesis. Together with Sebastian Kanthak I realized a very important corner stone of my work in the short time that he was also part of Mira's group. Collaboration with Andreas Sewe, whose Diploma

thesis was an important conceptual contribution to my work, kept to be fruitful after he joined Mira's group as PhD assistant. Furthermore, Michael Eichberg and Ralf Mitschke contributed to my work in early stages. Thanks also to Thorsten Schäfer and Shadi Rifai—who accompanied me especially closely—for the motivational *Lumpy competition*.

My very special thank goes to Matthew Arnold from the IBM T. J. Watson Research Center who collaborated with me on the implementation of my key contributions in the field of virtual machine integration; his insights into the Jikes RVM were invaluable for their success. But I also want to thank all those students who contributed to my work by participating in practical courses or by means of their theses, who are: Florian Breuing, Sarah Ereth, Hristo Indzhov, Jannik Jochem, Markus Maus, Suraj Mukhi, Dennis Müller, Heiko Paulheim, Nico Rottstädt, and Nathan Wasser.

The AOSD-Europe Network of Excellence (European Union grant no. FP6-2003-IST-2-004349) has funded part of this work. To all the partners of the network, I also want to say thank you for being a forum of presenting and discussing my work.

Although, I keep mentioning similarities between writing a paper and writing the PhD thesis, there are some significant differences. It's much more difficult to tell a coherent story over 200 pages than over 20 pages. You also have to invest a large effort for a comparatively long period. While these difficulties taught me some very interesting lessons, they also necessitated help from several other people. To start with, I want to thank my friends whom I could rarely meet towards the end of writing my thesis—but nevertheless they still turned-up celebrating together with me after the exam. A special thank must go to Gudrun Jörs for helping me in many ways: she took the load of organizing many of the needs of the PhD exam and other things off from me, she was always available for a chat when I needed diversion, and she regularly provided me with delicious cake which kept my blood glucose level in the range required by a brain writing a PhD thesis. I also like to thank my family, especially my parents Marianne and Andreas and my sister Sonja, for their support, for proof-reading the thesis—even though it is written in English—and for providing valuable comments. For all the above, for constant encouragement and motivation, and for her adorable patience I especially thank my wonderful wife Sabrina to whom I dedicate my work and this thesis.

Chapter 1

Overview

1.1 Context of this Thesis

This thesis is concerned with the implementation of aspect-oriented programming languages. Programming language implementations are usually split into a *front-end* and a *back-end*. A front-end, e.g., a compiler for the language, translates the program into a representation suitable for execution, whereby it resolves implicit content like unqualified names used in the program and performs static error checking. The back-end, also called *execution environment* for a programming language, executes the front-end's output. The execution environments considered in this thesis are a combination of libraries and a virtual machine (VM) that manages the execution of applications written in that language. Managing the execution means to provide services like memory management [HB05] that otherwise had to be explicitly programmed, or dynamic optimizations [AHR02] of the executed application. The management is facilitated because the front-ends of most modern language implementations generate output in terms of an *intermediate language* of virtual instructions which are similar to assembler but still reflect the concepts of the source language.

Current implementations of virtual machines perform *just-in-time* (JIT) compilation [DS84, ACL+99, Ayc03] of the intermediate language. That means that the virtual instructions of the intermediate language are compiled to native machine code at runtime. It is called "just-in-time" because the compilation of a method is performed at the latest possible point in time, i.e., just before it has to be executed. Management of the executed application by the virtual machine is facilitated by the JIT compiler which inserts calls to the virtual machine's services into the machine code.

This thesis aims at advancing the state-of-the-art in the area of *execution environments* for *aspect-oriented* (AO) programming languages. Although, there are different flavors of aspect-oriented programming (AOP), in this thesis, dedicated support is only developed for the *pointcut-and-advice* flavor, also referred to as *PA languages* [MKD03]. A *pointcut* is an expression that evaluates to a set of *join points*, which are points in time during the execution of a program; pointcuts can be associated with *advice*—a piece of code—which means that the advice is to be executed when a join point selected by the associated pointcut is reached. The term "aspect-oriented language" is used in this thesis synonymously with the term "PA language" where not noted otherwise.

The current major aspect-oriented languages [KHH+01, AGMO06, BA01, VSV+05] are realized as extensions of conventional programming languages. That means that those concerns of the program that can be well modularized with the conventional language's means are written in the so-called *base language*. *Aspects* are an additional kind of module containing pointcut-and-advice as a means to modularize the remaining program parts. The advice code usually also consists of statements from the base language.

Much of the research in aspect-oriented languages is performed by extending the Java programming language, which is also true for the most popular [Ker] AOP language AspectJ [Aspa]. Thus, Java *bytecode*—the intermediate language of the Java Virtual Machine (JVM)—is generated by AO language front-ends. If not noted otherwise, execution environments discussed in this thesis are for the Java language or extensions thereof. In examples throughout this thesis, Java is used as the base language and AspectJ as the aspect-oriented language. Code generation is discussed in terms of Java bytecode; often examples of generated code are presented in Java source code rather than bytecode for reasons of simplicity. Discussing code generation and the implementation of optimizations in terms of Java bytecode and JVMs is no limitation. Virtual machines for other modern languages and their intermediate languages use very similar concepts [CLI06] and can be extended in similar ways.

The most prevalent implementation strategies of PA languages share the following conceptual work flow [HH04, DKM02, MKD02]. First, the aspect-oriented program is parsed to retrieve pointcuts and advice from its aspect modules. Next, the pointcuts are partially evaluated which results in a set of *join point shadows* which are instructions in the base program. Finally, the remainder or the partial evaluation—called the *residual dispatch*—drives the generation of code that is inserted at the join point shadow instructions. The latter two steps are generally referred to as the *weaving* phase. Weaving

is usually performed at compile-time or class loading-time [HM04b, Aspa, SVJ03, Bon03].

The residual dispatch at a join point shadow is represented in bytecode as instructions that evaluate the dynamic properties referred to by pointcuts that are projected onto the shadow, i.e., those sub-expressions that remained from the partial evaluation. The residual dispatch also comprises instructions that execute the advice functionality of those pointcut-and-advice of which the dynamic properties evaluated to **true**.

For illustration, consider the example in Listing 1.1, which is further elaborated in Section 2.1.1. In a client-server application, a database access layer is executed on the server. Through this layer, SQL queries are performed, which can either be triggered by the client or by the server. The server is trusted to only perform legal SQL queries, but the client may as well perform a query-injection attack; thus queries that originate from the client have to be validated first. Assume there is a method Servlet.doPost which is executed for every client request.

```
1 before() :
2         call(ResultSet Connection.query(String sql))
3         && cflow(void Servlet.doPost(Request, Response)) {
4     // validate the SQL string
5     // if it is invalid, throw an exception
6 }
```

Listing 1.1: Pointcut-and-advice for validating client SQL queries.

The pointcut in Listing 1.1 (lines 2–3) first selects all calls to the query method on a database Connection (line 2). Next, it specifies that only those calls are selected which occur in the control flow of the Severvlet.doPost method (line 3).

Considering this example, the join point shadows for this pointcut are all call sites of the method Connection.query. The residual dispatch at these join point shadows has to test at runtime whether the call is performed in the control flow of Servlet.doPost, i.e., whether Servlet.doPost is on the call stack. Listing 1.2 shows the code generated for the residual dispatch of a join point shadow affected by the pointcut-and-advice in Listing 1.1.

Line 3, Listing 1.2 represents the join point shadow; lines 1–2 constitute the residual dispatch, whereby in line 1 the dynamic part of the pointcut is evaluated and in line 2 the advice is executed if the dynamic part evaluates to **true**.

```
1 if(<cflow−test>)
2   <execute advice>
3 connection.query(sql);
```

Listing 1.2: Residual dispatch code for a join point shadow.

In this thesis the term *dynamic AOP* refers to PA languages with the following features.

1. Join points are selected based on *dynamic properties of the runtime context* in which they occur.

2. It is possible to *deploy* and *undeploy* aspects arbitrarily at runtime; only when an aspect is deployed its pointcut-and-advice take effect.

1.2 Introduction

Aspect-oriented concepts are expressed by special language constructs in the source code. Their explicit representation, however, vanishes after the weaving phase. Instead of being well modularized as in the source code, aspects are merged with code of the base language's modules in the intermediate representation. For example, the pseudo expression <cflow−test> in Listing 1.2 is comprised of several general purpose instructions of the base language.

In contrast, object-oriented concepts like classes, methods, fields and even polymorphism are also reflected in the intermediate representation. In Java bytecode [LY99], polymorphic method calls can be immediately identified because they are represented by the special **invokevirtual** instruction rather than multiple general-purpose instructions encoding the resolution logic.

One sort of problems is that this weaving approach weakens features of the software engineering process like debugging support, or the continuity property of incremental compilation: modifying an aspect in the source code potentially requires re-weaving multiple other modules [SDR08, RDHN06]. Another sort of problems of the representational gap between aspect-oriented concepts in the source code and in the intermediate code is related to language implementations. It is a very difficult task for the just-in-time compiler of a virtual machine to recognize the intention of a sequence of instructions generated by the weaver, e.g., for checking whether a control flow is active[1].

[1]Recently, Golbeck et al. [GDN+08] have implemented an approach for recognizing the intention of such instruction sequences that realize aspect-oriented concepts and use this information to drive optimizations at runtime. Their work, however requires the instructions to be annotated with meta-information. This approach, its strengths and weaknesses compared to the one from this thesis are discussed in Section 6.3.2.

On the contrary, a first-class representation of quantifications over the control flow can be exploited by the virtual machine's just-in-time compiler to perform optimizations during native code generation.

This thesis proposes the Aspect-Language Implementation Architecture (ALIA) which prescribes the existence of an intermediate representation preserving the aspect-oriented constructs of the source program. A central part of this architecture is the extensible and flexible meta-model of aspect-oriented concepts which acts as an interface between front-ends and back-ends of AO language implementations.

The architecture is embodied in the Framework for Implementing Aspect Languages (FIAL) and the Language-Independent Aspect Meta-Model (LIAM) which is part of the framework. FIAL defines the work flows required of an execution environment when executing aspects provided in terms of LIAM. Both FIAL and LIAM are tailored towards execution environments that weave aspects at the code level, e.g., they require to distinguish between static and dynamic properties of join points to which a pointcut can refer.

In addition to the first-class intermediate representation of aspect-oriented concepts, ALIA and the FIAL framework as its incarnation treat join points as being late-bound to an implementation. In analogy to the OO terminology for late-bound methods, join points are called *virtual* in ALIA. The first-class representation of AO concepts in the intermediate representation available to the execution environment as well as treating join points as being virtual facilitate the implementation of new and effective optimizations for aspect-oriented programs independently from the concrete high-level-language they are written in. Because the intermediate representation is general enough to allow multiple AO languages to be mapped onto it, optimizations of features common to multiple AO languages are available to applications written in all these languages. The use of an extensible meta-model as intermediate representation furthermore simplifies the definition of new AO language concepts.

1.2.1 Virtual Join Points

The architecture proposed in this thesis evolves around the concept of virtual join points which are points in the program execution to which meaning is late bound at runtime. The different meanings, called join point implementations, are the join point's execution with any possible combination of advice; the simplest possible join point implementation is executing the join point unadvised when no pointcut-and-advice is applicable at runtime. In the proposed architecture, join point shadows are conceived as *virtual invocation sites* of join points. The so-called *dispatch function* of a join point shadow is

evaluated at runtime in order to determine the join point implementation to be executed.

The correspondences between virtual join points and virtual methods are as follows. A join point shadow is the equivalent of a virtual method's call site; join point implementations, i.e., combinations of actions that have to be executed at runtime when a join point shadow is reached, correspond to the virtual method implementations that are a method call's potential targets. The adequate action combination is determined by evaluating the join point shadow's dispatch function, similar to performing look-up of a virtual method implementation.

The virtual method *look-up* [ES90], in fact, is just an optimization of a dispatch function that determines the adequate target method. This function takes the receiver object as input and maps each possible type to a concrete method implementation which is, in turn, the dispatch function's result. Because the domain of this function is highly structured and very constrained, an optimized implementation strategy, namely using a look-up table, is possible.

When a class is loaded dynamically, the dispatch function for virtual methods may change: after class loading, it may have additional possible input and result values. Dynamic aspect deployment, which affects the dispatch functions of those join point shadows that are matched by the aspect's pointcuts, is the counterpart of dynamic class loading.

In this thesis, a new approach for weaving pointcut-and-advice is developed introducing a layer of indirection into program code. All potential join point shadow instructions are replaced with a call to so-called *envelope* methods which become the manifestation of join point shadows. As a result join point shadows become explicit structures in the code, isolated from the base code.

The envelopes approach improves the performance of aspect weaving; the way envelope methods are generated ensures that they have a simple control flow structure which reduces the complexity of aspect weaving as compared to weaving aspects in arbitrary code of the base program. This makes the envelopes approach especially suitable for application in execution environments with support for runtime weaving. In such an environment, an efficient weaving approach is important because delays caused by dynamic aspect deployment decrease the overall performance of an application.

1.2.2 Aspect-Oriented Language Implementation Architecture

The Aspect-Oriented Language Implementation Architecture (ALIA) separates the implementation of front-ends and back-ends for aspect-oriented programming languages. In the architecture, compilers do not weave aspects into the compiled program but generate a first-class model of them. The execution environment is required to execute the program as defined in the model; therefore, it must understand the model and it must manage several activities like deploying or undeploying aspects, and dynamic class loading. As a result of dynamic class loading new code gets loaded at runtime which contains join point shadows potentially affected by aspects that are currently deployed or will be deployed in the future.

The Framework for Implementing Aspect Languages (FIAL) supports this architecture; one essential part of it is the Language Independent Aspect Meta-Model (LIAM) for pointcut-and-advice flavor aspect languages. The following core components of aspects in PA languages are represented by LIAM entities.

- Pointcuts select join points by referring to their *static and dynamic properties*.
- Pointcuts can *bind values from a selected join point's context* to variables.
- Pointcuts can be associated with *actions constituting the advice functionality* to be executed at join points selected by the pointcut.
- Advice *functionality* can be specified declaratively.
- Advice functionality can also be provided by means of a method that has to be invoked at matching join points; this method is called the *advice method* throughout this thesis. For a pointcut-and-advice a strategy can be specified that is used for *implicitly instantiating the receiver of advice methods*.
- It is possible to control the *execution order of advice at shared join points*. The term shared join point is used to denote a join point at which multiple advice have to be executed.
- Aspects are *logical groups of pointcut-and-advice that are to be deployed together*.

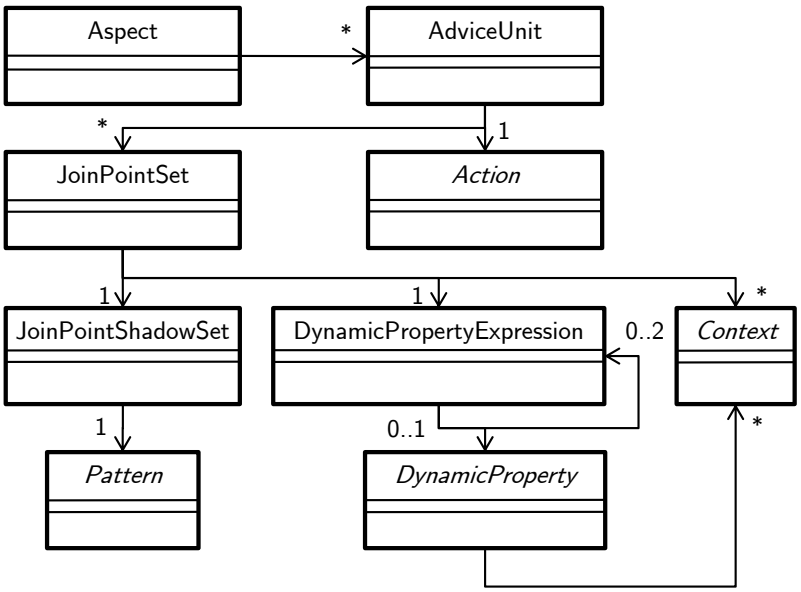


Figure 1.1: The most important LIAM entities.

- Aspects can be *dynamically deployed and undeployed*. It is possible to deploy an aspect only in a certain scope, e.g., a specific thread.

The meta-entities in LIAM are realized as abstract classes that capture the relationships among the corresponding concepts. Figure 1.1 shows the most important ones of LIAM's classes. Classes in italics are abstract and model concepts that can be refined. For example, DynamicProperty models the concept of a property that must be satisfied in the dynamic context of a join point. Non-abstract classes represent logical groupings rather than concepts and, thus, cannot be refined; an example is the class Aspect that represents a logical group of AdviceUnits, the LIAM equivalent to pointcut-and-advice. The role of each class is discussed in detail in Section 3.2.

Figure 1.2 shows how the FIAL framework is used by language front-ends and back-ends. The front-end (index 1), e.g., a compiler, generates two kinds of artifacts when it processes a program: a model based on the concrete sub-classes of the LIAM meta-entities for the aspect-oriented parts of the program (index 2), and intermediate code for those parts of the program that are expressible in the base language (index 3).

In order to be able to express an aspect written in some AO language in terms of LIAM, the meta-model has to be concretized first, as depicted by the realization relationship (index 4) in Figure 1.2. This is achieved by providing sub-classes that implement the specific semantics of the concepts in the corresponding language. In this thesis, existing relevant AO

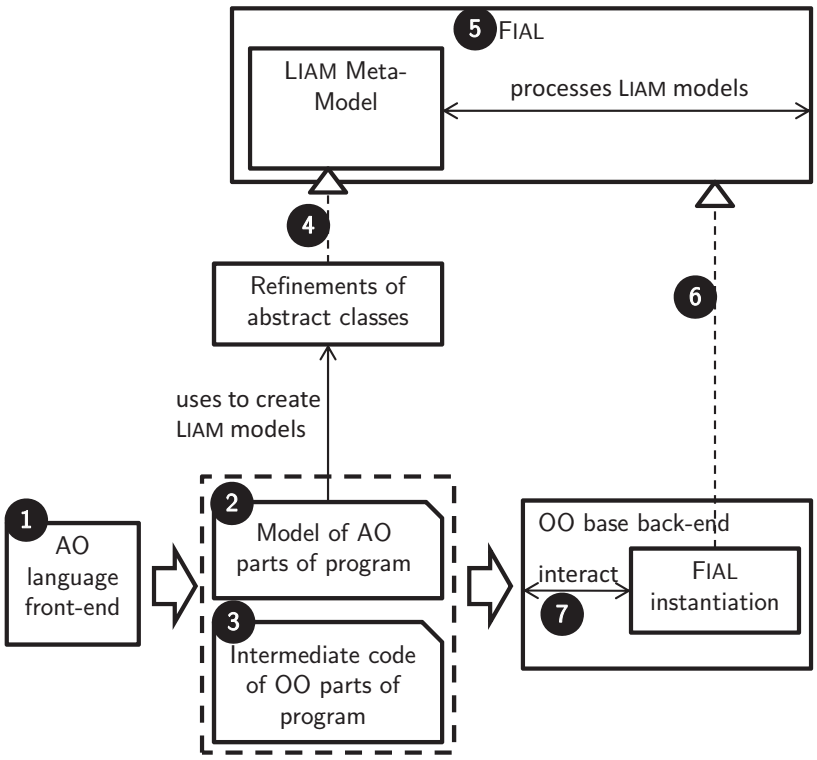


Figure 1.2: Overview of different usages of FIAL.

languages are investigated; their concepts are mapped to LIAM and provided as default implementations of its abstract classes. For example, a class CFlowDynamicProperty has been implemented as a sub-class of DynamicProperty to realize the semantics of the **cflow** pointcut designator of AspectJ.

FIAL (index 5 in Figure 1.2) is a framework providing support for several common tasks of execution environments for AO languages. Execution environments for OO languages manage meta-objects, e.g., of classes, methods and fields in the executed program. FIAL provides support for additionally managing meta-objects of the join point shadows in the program. It also generically implements common work flows for handling join point shadows, leaving some functionality abstract which needs to be realized by FIAL's instantiations.

It is intended that a FIAL instantiation (index 6, Figure 1.2) is realized as an extension of an existing execution environment for the base language. The instantiation realizes the abstract functionality by interacting with the base execution environment depicted by the arrow at index 7. The most important functionality is to perform code generation for join point shadows for which a declarative description is provided by FIAL. Further, the FIAL instantiation also must be able to re-create the code of a join point shadow that has changed due to deployment or undeployment of an aspect, and it must notify the FIAL framework of class loading.

An Envelope-based Reference ImplementatioN (ERIN) of FIAL has been developed in this thesis that implements a default code generation strategy for each LIAM meta-entity. The default code generation is based on the requirement that concrete sub-classes of LIAM classes provide a method realizing their meaning, e.g., by using reflection. For the pointcut-and-advice in Listing 1.1, ERIN generates the code in Listing 1.3 for the affected join point shadows. The isSatisfied method called on the dynamicProperty object is the method realizing the dynamic property's meaning.

```
1 if(dynamicProperty.isSatisfied())
2   //execute advice
3 connection.query(sql);
```

Listing 1.3: Pseudo code generated for a join point shadow by ERIN.

ERIN is realized as an extension to standard Java 6 virtual machines implemented as a Java agent. The agent intercepts class loading and uses the Class Redefinition [Ins] facility of JVMs in order to manipulate the code of join point shadow instructions and to re-create the code at (un)deployment.

When generating code for a join point shadow according the weaving specifications provided by FIAL, each entity can be handled separately. Thus, it

is possible for execution environments to separately optimize the code generation for single entities. ERIN exploits this facility and provides dedicated code generation strategies for some selected concrete entities as a proof-of-concept.

To show that FIAL is a suitable back-end for a variety of aspect-oriented programming languages, the languages AspectJ, JAsCo and Compose* are mapped to LIAM. Although these languages are very different in their source-level concepts, they could successfully be mapped to LIAM. Concepts provided by the languages are mapped to concrete LIAM entities and many of them are shared among the different mappings. For each language, a component has been developed that automatically translates a language-dependent intermediate representation of aspects into a LIAM model.

FIAL's capability of decoupling the definition of new language constructs from an optimizing implementation is shown by realizing a new sample language. This exercise shows that a language designer can provide an implementation, e.g., of the abstract class DynamicProperty, and implement its meaning as a normal Java method. The implementer of an optimizing execution environment is later provided with a precise specification of the new concept's semantics in terms of Java code. He/she can implement an optimizing code generation strategy for the new concept and can test it against the provided implementation.

1.2.3 VM Integration of AO Concepts

Integrating support for aspect-oriented concepts as instantiation of FIAL deeper into a virtual machine opens up additional optimization opportunities that go beyond the generation of optimal bytecode. For example, direct memory access, which is important for implementing optimized data structures for meta-information, cannot be performed by bytecode instructions.

A deep integration of the concepts of dynamic deployment and of the **cflow** dynamic property of join points into the virtual machine is realized in this thesis. Both optimizations outperform current approaches that are limited by the potential of bytecode instructions. The optimization of dynamic deployment is based on the concept of virtual join points and exploits similarities to the concept of virtual methods: established and highly effective optimizations of virtual method dispatch are adapted. They make use of techniques for de-virtualizing methods so that the indirection of virtual method calls can be saved in the native code generated by the JIT compiler [Mel99].

For non-virtual methods, it is obvious that the JIT compiler can fully resolve the called method and the overhead of method look-up can be saved.

In this case, the just-in-time compiler additionally can copy the native code for that target method directly into the call site—an approach called *inlining* the callee into the caller. Besides saving the look-up, inlining also facilitates further subsequent optimizations: after inlining, the JIT compiler can process the caller's and the callee's code together and can jointly optimize it.

Inlining can also be applied to virtual methods [AFG+05], if there is only one possible target method at a virtual method call site at JIT compile-time—this kind of optimization is called *speculative optimization*. The compiler makes the assumption that the method stays the only possible target in the future and inlines the currently single possible target method. In addition, the JIT compiler takes care that the inlining can be undone when the assumption is invalidated later due to dynamic class loading.

There is a range of different implementations for undoing speculative optimizations. In this thesis different implementations of speculative inlining are investigated with respect to their applicability to optimize virtual join point dispatch in the presence of dynamic deployment.

Besides efficient support for dynamic deployment, other aspect-oriented concepts also benefit from a deep integration into the virtual machine. In this thesis, a new optimization has been developed that allows for fast queries over the current control flow as required for the **cflow** dynamic property of join points. For this optimization, further components of the virtual machine are adapted in addition to the just-in-time compiler, including, the layout of the call stack and the virtual machine's thread system.

1.2.4 Benchmarks

With the speculative inlining of join point shadows, and the modifications to the call stack layout and the thread system for fast checks of the **cflow** dynamic property, two particularly powerful optimizations are developed in this thesis. In order to speed-up deployment respectively querying the control flow, both optimizations require supporting infrastructure. To rate the effectiveness of both optimizations and to show that the presence of the additional infrastructure has only minimal impact—if any—on the overall performance, the optimizations are thoroughly benchmarked.

Two kinds of benchmarks are performed. First, micro-benchmarks measure the impact on specific operations that are either affected by the optimized concept or by the required infrastructure. Second, macro-benchmarks measure the optimizations' impact on large-scale applications.

No standard benchmarks exist for dynamic aspect-oriented execution environments and no reasonably large aspect-oriented applications are available that make frequent use of dynamic AO concepts like dynamic deployment

and **cflow**. Therefore, a new benchmarking approach has been developed in this thesis. In this approach, the established SPEC JVM98 benchmark suite [SPE] for Java virtual machines has been extended with aspects that stress the usage of dynamic deployment and **cflow**. Although the used aspects themselves do not contribute to the applications' functionality, the benchmarking approach allows to investigate how the aspect-oriented concepts impact the performance in different real-world situations. Hence, this benchmarking approach is a valuable supplement to micro-benchmarks which only evaluate the performance of AO concepts separately from the application in which they are applied.

1.3 Summary of Contributions

This thesis makes three main contributions. First, an architecture for implementing aspect-oriented languages based on late-binding join point shadows is proposed. A framework embodying the architecture has been developed that supports the implementation of execution environments as well as language front-ends. As a proofs-of-concept one full-fledged execution environment has been developed as an instantiation of the framework, and relevant aspect-oriented programming languages have been mapped to the framework's meta-model of aspects. The feasibility of integrating further execution environments and languages with the framework is discussed. The framework supports designers and implementers of new aspect-oriented language concepts because design and optimizing implementation are separated.

Second, optimizing implementations of dynamic aspect deployment and the **cflow** dynamic property of join points have been developed. The optimizations are facilitated by a deep integration with the virtual machine and benefit from the fact that concepts like deployment and referring to the active control flow are first-class as stipulated by the developed architecture. The optimized concepts exhibit a performance that cannot be met by conventional AO language implementations operating on bytecode.

Third, means for benchmarking the dynamic aspect-oriented features of aspect deployment and support for the **cflow** dynamic property have been developed. The developed benchmarks are macro-benchmarks, i.e., they execute real-world applications.

The optimizations presented here drive another more conceptual contribution of this thesis. The efficiency improvements that result from the integration with the virtual machine emphasize an advantage of aspect-oriented quantification mechanisms, e.g., **cflow** and deployment, that has not been discussed so far. As yet, increased modularity has been the main argument

for AOP. While this is certainly the key benefit of AOP, this thesis shows that AOP also has the potential to make programs more efficient as compared to object-oriented programs. The idea is that by preserving aspect-oriented concepts explicit also at execution time, the possibility of applying sophisticated optimizations is opened-up that are out of reach for conventional language implementations.

The publications by the author of this thesis related to these contributions are listed at the beginning of each chapter. A complete list of scientific publications by the author is as follows.

- Publications in conference proceedings:

  - Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006
  - Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient Control Flow Quantification. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006
  - Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based Weaving for Faster Aspect Compilers. In *Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. GI, 2005
  - Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 2005
  - Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005
  - Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2004

- Publications in workshop proceedings:
  - Andreas Sewe, Christoph Bockisch, and Mira Mezini. Redundancy-free Residual Dispatch. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages*, 2008
  - Christoph Bockisch and Mira Mezini. A Flexible Architecture For Pointcut-Advice Language Implementations. In *Proceedings of the Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*. ACM Press, 2007
  - Christoph Bockisch, Michael Haupt, and Mira Mezini. Dynamic Virtual Join Point Dispatch. In *Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006
  - Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over Dynamic Properties of Program Execution. In *Proceeding of the Dynamic Aspects Workshop*, 2005
- Technical reports:
  - Christoph Bockisch, Andrew Jackson, and David Cousins. Second Review of Atelier Content and Performance. Technical Report AOSD-Europe-TUD-10, Technische Universität Darmstadt, 2008
  - Christoph Bockisch, Andreas Sewe, Mira Mezini, Arjan de Roo, Wilke Havinga, Lodewijk Bergmans, and Kris de Schutter. Modeling of Representative AO Languages on Top of the Reference Model. Technical Report AOSD-Europe-TUD-9, Technische Universität Darmstadt, 2008
  - Christoph Bockisch, Mira Mezini, Kris Gybels, and Johan Fabry. Initial Definition of the Aspect Language Reference Model and Prototype Implementation Adhering to the Language Implementation Toolkit Architecture. Technical Report AOSD-Europe-TUD-7, Technische Universität Darmstadt, 2007
  - Christoph Bockisch, Mira Mezini, Wilke Havinga, Lodewijk Bergmans, and Kris Gybels. Reference Model Implementation. Technical Report AOSD-Europe-TUD-8, Technische Universität Darmstadt, 2007
  - Andrew Jackson, Siobhán Clarke, Matt Chapman, and Christoph Bockisch. Deliver Preliminary Support for Next-Priority Use Cases. Technical Report AOSD-Europe-IBM-80, IBM UK, 2007

- Christoph Bockisch and Michael Haupt. Taxonomy of Implementation Techniques in Relation to the Aspects of the Meta-Model. Technical Report AOSD-Europe-TUD-6, Technische Universität Darmstadt, 2006
- Johan Brichau, Mira Mezini, Jacques Noyé, Wilke Havinga, Lodewijk Bergmans, Vaidas Gasiunas, Christoph Bockisch, Johan Fabry, and Theo D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe-VUB-12, Vrije Universiteit Brussel, 2006
- Andrew Jackson, Siobhán Clarke, Matt Chapman, Andy Dean, and Christoph Bockisch. Deliver Preliminary Support For Top Priority Use Cases. Technical Report AOSD-Europe-IBM-64, IBM UK, 2006
- Johan Brichau, Michael Haupt, Nicholas Leidenfrost, Awais Rashid, Lodewijk Bergmans, Tom Staijen, Istvan Anis Charfi, Christoph Bockisch, Ivica Aracic, Vaidas Gasiunas, Klaus Ostermann, Lionel Seinturier, Renaud Pawlak, Mario Südholt, Jacques Noyé, Davy Suvee, Maja D'Hondt, Peter Ebraert, Wim Vanderperren, Shiu Lun Tsang Monica Pinto, Lidia Fuentes, Eddy Truyen, Adriaan Moors, Maarten Bynens, Wouter Joosen, Shmuel Katz, Adrian Coyler, Helen Hawkins, Andy Clement, and Olaf Spinczyk. Report describing survey of aspect languages and models. Technical Report AOSD-Europe-VUB-01, Vrije Universiteit Brussel, 2005
- Michael Haupt, Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Towards Aspect-Aware Execution Models. Technical Report TUD-ST-2003-01, Technische Universität Darmstadt, 2003

1.4 Structure of this Thesis

The remainder of this thesis is structured as follows. Chapter 2 discusses early and late binding of join points. The pointcut-and-advice flavor of AOP languages as well as background information on Java bytecode and bytecode weaving are presented in Section 2.1. Section 2.2 discusses early and late binding of join point shadows. In Section 2.3 the concepts of late bound, or virtual, methods and late bound join points, analogously called virtual join points, are discussed; the approach of inserting envelope methods into base code to make join point shadows explicit structures is presented in this section as a step towards the semantics of virtual join points. A prototypical implementation of the envelopes approach is presented in Section 2.4 and

its effectiveness in making the weaving process simpler and, therefore, more efficient is evaluated in Section 2.5.

Chapter 3 presents the proposed architecture for aspect language implementations. Section 3.1 discusses features of current aspect-oriented languages that are supported by the meta-model developed in this thesis and presented in Section 3.2; the developed framework for aspect-oriented execution environments, of which the meta-model is a part, is presented in Section 3.3. Section 3.4 presents mappings from the established aspect-oriented languages AspectJ, CaesarJ, Compose* and JAsCo to the meta-model, and Section 3.5 presents execution environments implemented as an instantiation of the framework, including the full featured reference implementation based on the envelopes approach and Java Class Redefinition. The chapter ends with a case study of separately developing the design and an optimizing implementation of a new aspect-oriented language in Section 3.6.

Optimizations of AO concepts based on virtual machine integration are presented in Chapter 4. The approach of providing dedicated virtual machines for AO languages is motivated in Section 4.1. Background information on established dynamic optimizations applied in modern virtual machines is given in Section 4.2. Section 4.3 discusses the optimizing implementation of dynamic aspect deployment; optimized support for **cflow** is presented in Section 4.4.

Chapter 5 presents a macro-benchmarking approach for dynamic aspect-oriented language concepts in Section 5.1. In Section 5.2 the results of benchmarking the prototype of the dynamic deployment optimization are presented and discussed. Section 5.3 presents and discusses the benchmark results for the **cflow** prototype. In both sections, the benchmarks are also performed for related approaches and the results are compared.

Additional related approaches that are not used in comparisons throughout the thesis are discussed in Chapter 6. Chapter 7 concludes this thesis in Section 7.1 and presents some areas of future and on-going work in Section 7.2.

Chapter 2

Virtual Join Points

The notion of virtual join points follows an intuitive definition of the semantics of pointcut-and-advice which is, however, not mirrored by current implementations of AO languages. In this chapter, virtual join points are discussed in analogy to virtual methods in order to determine commonalities between both concepts. The commonalities suggest that techniques for virtual method call optimization can be applied to aspect-oriented programs as well. They also give rise to requirements how to treat virtual join points in the intermediate representation of an application.

A step is made toward this "direct semantics" of AO languages by introducing the concept of envelopes which are methods that dissect join point shadows from the base code, thereby facilitating to handle them separately. This opens-up opportunities for optimizing join point dispatch and speeds-up the weaving of aspects into applications; this is especially important when aspects are deployed at runtime.

Parts of this chapter have been published in the following papers.

1. *Christoph Bockisch, Michael Haupt, and Mira Mezini. Dynamic Virtual Join Point Dispatch. In* Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006*
2. *Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based Weaving for Faster Aspect Compilers. In* Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. GI, 2005*
3. *Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In* Proceedings of the European Conference on Object-Oriented Programming*, 2005*
4. *Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over Dynamic Properties of Program Execution. In* Proceeding of the Dynamic Aspects Workshop*, 2005*

2.1 Background

Aspect-oriented programming languages of the pointcut-and-advice flavor (PA flavor) introduce four main concepts. *Join points* are points in the execution of a module that are exposed for composition. The set of events that can be join points are determined by the *base language* in which the program is written. *Pointcuts* are expressions that quantify over join points in aspect-oriented programs. They can be associated with an *advice* that influences the execution of the selected join points and *aspects* are modules containing pointcut-and-advice.

In this section, the pointcut-and-advice flavor and its concepts are characterized in detail. Most of the current AO languages are realized as extensions of the object-oriented programming language Java. Therefore, aspect-oriented concepts are usually woven into Java bytecode. The bytecode format and some of its implications on the implementation of an aspect weaver are presented in the second sub-section.

2.1.1 Pointcut-and-advice Languages

For illustration of the PA flavor of AO languages, the example from Section 1.2.1 is elaborated here. The context of this example is a client-server application using Java Servlets with a data access layer executed on the server. For every request made by the client, the method Servlet.doPost is called whereby its Request argument contains data passed by the client.

The server-side application distinguishes between two different contexts of trust from which database accesses, further-on denoted by the method Connection.execQuery, may happen. Internal requests to the database are trusted, but requests originating from the client are not trusted. Listing 2.1 shows an excerpt of the Services class with two service methods that execute an SQL query on the database; the query is passed as a String parameter. Using parameters originating from untrusted sites in SQL statements bears the risk of an SQL command injection attack where unexpected parameter values are used to alter the effect of the SQL query, e.g., in order to gain unauthorized access to data. To avoid security leaks, SQL statements should be checked whether they are secure before they may execute if they stem from an untrusted context. In this example, calls to Connection.execQuery are untrusted if they are in the control flow of the Servlet.doPost method, i.e., the call is directly performed by Servlet.doPost or by any transitively called method.

```
1  class Services {
2    Connection connection;
3
4    void service1(String sql) {
5      //...
6      connection.execQuery(sql);
7    }
8
9    void service2(String sql) {
10     //...
11     connection.execQuery(sql);
12   }
13 }
```

Listing 2.1: Example Services class of the base program.

In Listing 2.2 an aspect is presented that captures all untrusted calls to Connection.execQuery to check the SQL statement. First, the aspect defines *when* to check SQL statements by specifying the pointcut in lines 3–5: it selects join points which are calls to the execQuery method in the control flow of method Servlet.doPost. The sequence diagram in Figure 2.1 shows a possible execution that uses the Services class from Listing 2.1. The selected method calls are marked with index a. Marked with index b are those calls to execQuery that are not in the control flow of doPost and, thus, not selected by the pointcut. Second, the aspect defines *what* should happen at the join points selected by the pointcut, in terms of *advice*, a piece of code (lines 6–9) associated with the pointcut.

```
1  aspect QueryInjectionChecker {
2    before(String sql):
3      call(ResultSet Connection.execQuery(String))
4        && args(sql)
5        && cflow(void Servlet.doPost(Request, Response))
6    {
7      if (isAttack(sql))
8        throw new SQLInjectionException();
9    }
10 }
```

Listing 2.2: Aspect checking database accesses from untrusted contexts.

The pointcut given in the example consists of several sub-expressions which demonstrate the different responsibilities of AspectJ pointcut designators.

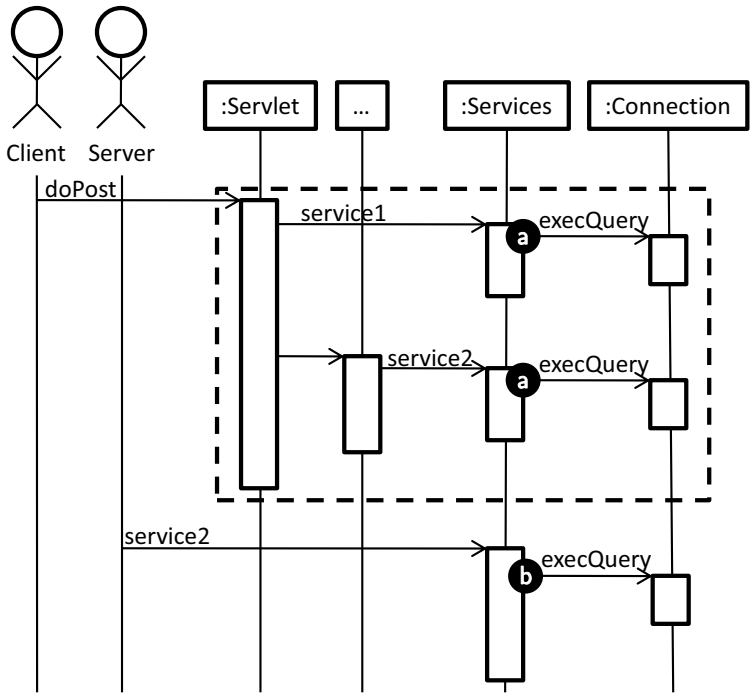


Figure 2.1: Accessing a database from the client as well as the server.

- Pointcuts can select join points based on their *static properties*. This is illustrated by the **call** sub-expression in line 3, Listing 2.2, which selects all calls to methods whose signature matches the expression in the parentheses.

- Pointcuts can also select join points based on their *dynamic properties*. In Listing 2.2, the **cflow** sub-expression (line 5) select join points which are in the control flow of the Servlet.doPost method as illustrated by the dashed box in the sequence from Figure 2.1. The box marks the control flow of the Servlet.doPost method in that sequence and every join point that occurs inside this box is in the control flow of Servlet.doPost.

- In the AspectJ language, pointcuts can, furthermore, *reify values* from the join point's context. This means that values from the context are passed as an arguments to the associated advice. The **args** sub-expression in line 4 (Listing 2.2) reifies the argument of the method call to the identifier sql by which the advice can access it to check the SQL query for command injection.

Additionally to binding a context value, the **args** pointcut designator also imposes a constraint on selected join points, more specifically, on the dynamic

type of the argument value to bind. This can be useful because the **call** pointcut designator selects join points only in terms of the signature of the called method. To give an example, consider the case where the execQuery method takes a CharSequence as argument, but SQL command validation should only be performed for String values, e.g., because values originating from the client are always of type String. This situation is expressed by the pointcut-and-advice in Listing 2.3 where the **call** pointcut designator's pattern selects calls to the method execQuery(CharSequence) (line 2). The parameter sql (line 1) to which the **args** designator refers (line 3), however, has the type String.

```
1 before(String sql):
2   call(ResultSet Connection.execQuery(CharSequence))
3     && args(sql)
4     && cflow(void FacadeServlet.doPost(Request, Response)) {
5   if (isAttack(sql))
6     throw new SQLInjectionException();
7 }
```

Listing 2.3: A pointcut-and-advice using **args** as dynamic property.

2.1.2 The Java Bytecode Format

Current AO language implementations as well as some of the prototypes developed in this thesis perform weaving at the bytecode level. Thus, the Java bytecode format [LY99] has some implications on the design of weavers.

The bytecode for each class is stored in a separate file, and each file has several sections—as shown in Figure 2.2—that reflect the different entities in the class's source code of which only the relevant ones are discussed here.

First, a class file stores the so-called *constant pool* which contains the symbolic names of methods, fields, and types as well as literals used within the class. Subsequently, those values are only referred to by means of their index within the constant pool. Next, type information for the class is stored, comprising the class's modifiers and the fully qualified names of the class, its super class and its interfaces. This is followed by two sections holding the definition of all declared fields, respectively methods. Finally, each entity can have an arbitrary number of bytecode attributes which can be used by the compiler to store additional information for that entity, e.g., debugging information like a mapping from bytecode instructions to lines in the source code.

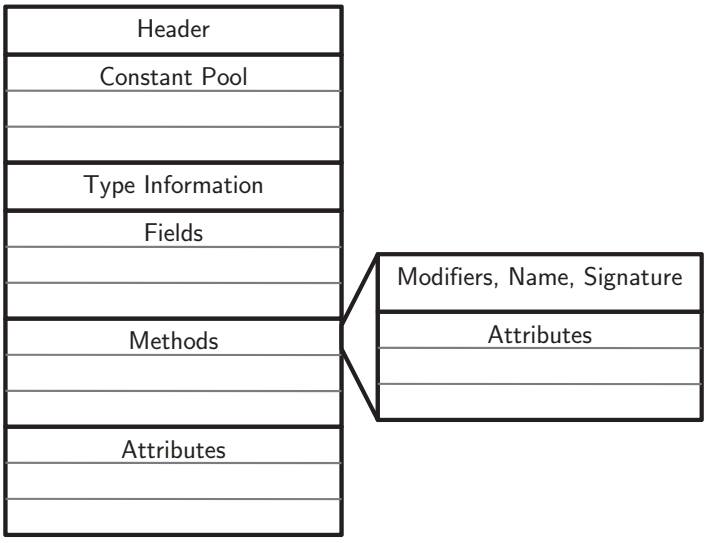


Figure 2.2: The Java Virtual Machine class file format.

A method definition consists of structural information: modifiers, the method's name, and the fully qualified names of the argument types and of the result type. If the method does not have the **abstract** or **native** modifier, it also has a body. In the bytecode format the body is represented by an array of bytecode instructions stored as a special attribute.

Each instruction has a so-called *opcode*—one byte that encodes the kind of instruction. For each opcode there is also a symbolic name, or *mnemonic*, which is used throughout this thesis to denote instructions. Following the opcode, an instruction can have a number of immediate values as parameters. An example for an immediate value is the constant pool index of a method reference for an invocation.

The Java Virtual Machine (JVM) instruction set is stack-based. That means, instructions which produce values, e.g., constants or the result value of a method call, push them onto the so-called *operand stack*—also called just stack where this is unambiguous. Required values likewise are read from the stack. For example, in order to call a method, the argument values have to be the top values on the stack when the method call instruction is executed. Thereby these values are popped from the stack and after the called method has completed its execution the result value is pushed onto the stack. Listing 2.4 shows an example of bytecode instructions that execute the statement `Clazz.method(1 + 2)`.

```
1 sipush       1
2 sipush       2
3 iadd
4 invokestatic #1 // void Clazz.method(int)
```

Listing 2.4: Bytecode instructions calling a method.

The **sipush** instructions each push a constant value on the stack, i.e., after the second instruction, the operand stack has the value 2 on its top and the value 1 is below. The **iadd** instruction pops two values from the stack, adds them and pushes the result back on the top. Thus, if the stack was empty before executing the instruction from the example, the stack contains only the value 3 afterwards. Finally, the instruction **invokestatic** calls a **static** method. The target method is specified in terms of a constant pool index—in the example #1—the method's symbolic name is written as a comment. Execution of this instruction pops the value (3) from the stack and passes it to Clazz.method.

Besides the operand stack, bytecode instructions can also use an arbitrary number of so-called *locals*. These are storage locations which can be randomly accessed by means of an index. Besides temporary variables, the arguments passed to a method are also stored as locals as shown in Listing 2.5. The listing shows the bytecode for the method Clazz.method which just passes its single argument (line 2) to the method Console.print.

```
1 void Clazz.method(int):
2   iload       0
3   invokestatic #2 //void Console.print(int)
```

Listing 2.5: Bytecode instructions accessing the method's arguments.

The instructions of a method are usually executed sequentially, but there are also instructions to alter the control flow. These instructions have as immediate value the offset of the target instruction relative to the jumping instruction's offset. The jump can either be unconditional as for the **goto** instruction or conditional. Conditionally jumping instructions are also called *branch* instructions. They pop at least one value from the stack and jump to the target instruction only if this value satisfies some condition. Otherwise, execution is continued with the instruction following the branch instruction; this is called the *fall-through case*. There are several different branch instructions, one example is **ifne** which pops one value from the stack and jumps if this is unequal to 0. Since the Java Virtual Machine represents the boolean values **true** and **false** as the numbers 1 and 0, the **ifne** instruction is also used to jump if a boolean value is **true**.

In addition to the control flow instructions, other parts of Java bytecode refer to instructions by their offset. Most importantly, this is the exception handler map. Similar to the **try-catch** structure in the source code, this map specifies the range of bytecode instructions—in terms of a start and an end offset—which may cause an exception and the offset of the instruction where to continue execution in the case of an exception—i.e., the **catch** block.

2.2 Binding of Join Points

Although pointcut-and-advice bind functionality to dynamic points in time the predominant implementation approach for PA languages binds join points to advice before the execution, usually at compile-time or class loading-time. This builds upon the idea of partial evaluation and inserting code to perform the residual dispatch [MKD03]. Partial evaluation of pointcuts leads to a set of instructions, the so-called *join point shadows* [HH04, DKM02], whose execution will always be selected by some of the pointcut sub-expressions. The sub-expressions that cannot be statically evaluated comprise the residual dispatch logic.

In the example from Listing 2.2, the pointcut sub-expression **call**(ResultSet Connection.execQuery(String)) can be statically evaluated to be **true** or **false** for any possible instruction: the execution of each instruction that calls a method named Connection.execQuery which takes a String as an argument and returns a ResultSet is a selected join point (lines 6 and 11, Listing 2.1).

As can be seen in Figure 2.1, not all executions of the joint point shadows (index a and b) are actually join points. Only those executions which happen in the control flow of Servlet.doPost (index a) are selected by the full pointcut.

Listing 2.6 shows a simplified version of the code that is generated by the AspectJ compiler for the example in Listings 2.2 and 2.1. Only as much details are shown as needed for illustrating the code generation.

```
1  class QueryInjectionChecker {
2    static void before_0(String sql) {
3      if(isAttack(sql))
4        throw new SQLInjectionException();
5    }
6  }
7
8  class Services {
9    Connection connection;
10
11   void service1(String sql) {
```

```
12      //...
13      if(<cflow-test>)
14        QueryInjectionChecker.before_0(sql);
15      connection.execQuery(sql);
16    }
17
18    void service2(String sql) {
19      //...
20      if(<cflow-test>)
21        QueryInjectionChecker.before_0(sql);
22      connection.execQuery(sql);
23    }
24  }
```

Listing 2.6: Compilation and weaving result of AspectJ.

All AO languages investigated in this thesis (presented in detail in Section 3.1) allow to write advice in terms of statements from the base language. These advice are compiled into methods—henceforth called *advice methods*—in the bytecode. For each aspect, the compiler generates a class—called *aspect class* from now on—that holds all its advice methods. For instance, the aspect in Listing 2.2, is compiled to the class in lines 1–6 in Listing 2.6; the advice in lines 6–9, Listing 2.2, is compiled to the method before_0 in lines 2–5 in Listing 2.6.

Examples of residual dispatch are shown in lines 13–14 and 20–21 in Listing 2.6. For advice execution, the weaver generates a call to the advice method in the residual dispatch; if context values have to be reified for use by the advice, these values are passed to the advice method as arguments (lines 14 and 21).

In [KM05] pointcuts are presented as the natural successors of method calls and declarations which are well-known mechanisms for binding meaning to points in the execution of a program. In fact, they bear a conceptual resemblance to virtual methods in object-oriented languages in the sense that both are late-bound to meaning.

Whenever a virtual method is called, some predicate is evaluated. The expressiveness ranges from a predicate on the dynamic type of the receiver in "traditional" method dispatch [ES90] to a predicate on the types of receiver and arguments in multi-dispatch [CLCM00], or, more radically, to an arbitrary predicate on dynamic values in predicate dispatch [Mil04]. Similarly, when a join point occurs, pointcuts are, at least conceptually, evaluated to determine which meaning to bind to the join point: the advice of matching pointcuts are executed; if no pointcut matches, the join point is executed un-

advised. This conceptual view is reflected in formalisms for object-oriented and aspect-oriented languages [CLW03, KHH+01, MKD03] and often is called the "direct semantics" of AOP in informal discussions.

The dispatch is preserved as first-class concept in the intermediate code of object-oriented languages such as Java [GJSB00], Smalltalk [Sma], and Self [US87]. In these languages, the compiler does not statically undertake any dispatching steps for method calls, it merely determines the dispatch strategy. In Java, e.g., this may be a dispatch using a virtual method dispatch table [ES90] (represented by the **invokevirtual** bytecode instruction) or, for interface methods, searching the method in the complete super type hierarchy [ACF+01] (represented by **invokeinterface**). However, all method calls remain virtual after compilation, i.e., their binding is per default determined by evaluating run-time predicate functions. It is the virtual machine that dynamically applies optimizations and decides to treat some of the method calls as non-virtual ones.

In contrast, despite the conceptual similarity to virtual methods, the "direct semantics" of AOP is not reflected in most current AO language implementations which share the conceptual work flow of first performing partial evaluation and second weaving residual dispatch [HH04, DKM02, MKD02]. During the weaving phase, aspect-oriented concepts, which are expressed by special language constructs in the source code, drive code generation and transformations in the compiler. An implication of this approach is that code originally modularized in aspects is merged with modules of the base program. Furthermore, there is no explicit representation of residual dispatch. There are several problems with this approach.

- It may be desirable to add an aspect to an application which is currently running. Consider the example of a client-server-application from the previous section. If the vulnerability to SQL command injections is discovered when the application is already in productive use, it is advantageous if the aspect can be added to the running system without the need to stop the application, re-compile it and start the new version. In an approach that binds join points early do advice, dynamic aspect deployment is expensive since the manipulation of application bytecode and its subsequent re-installation into the execution environment are time-consuming. Approaches exhibiting fast dynamic deployment behavior by preparing join point shadows with hooks and wrappers usually suffer from the footprint of the required infrastructure [Hau06] which degrades performance of other operations.

- The continuity property of incremental compilation is weakened: modifying *one* aspect potentially requires re-weaving in *multiple* other modules [RDHN06].

- The fact that aspect-oriented concepts become implicit instead of staying first-class in the generated bytecode hinders optimizations by the virtual machine.

- Dynamic class loading is poorly supported. It is, for example, possible to statically optimize a **cflow** pointcut to the best possible degree using sophisticated analyses [ACH+05b]. However, when classes are dynamically loaded which have not been available during static analysis the optimizations fail and the aspect may not be executed correctly anymore.

- The weaving approach followed by common AO language implementations is rather complex and slow: when weaving residual dispatch at a join point shadow, care must be taken of the control flow of the method containing the join point shadow, i.e., the weaver must update the method's instructions. Furthermore, join point shadows for pointcuts that match field accesses or method calls are generally spread all over the program and multiple join point shadows exist for the same accessed member. Hence, the same weaving action must be executed repeatedly at each of these shadows. As reported in [HH04] and shown in the evaluation in Section 2.5.2, even when no aspect is present in the code, the time spent for compiling an application with the AspectJ compiler increases considerably, compared to the time spent by a conventional compiler. For a simple aspect that logs the application's method calls the compile-time increases to more than 400% [HH04]. Poor compile time performance impacts the efficiency of the development process since it slows-down the "compile-test-debug" cycle [SDR08].

2.3 Virtual Join Points

To address the problems listed in the previous section, in this thesis, the approach is pursued to more cleanly support the "direct semantics" of AOP. The model of virtual join points is presented here as a counterpart to virtual method calls.

When executing a join point shadow, potentially a combination of advice has to be executed. The concrete execution of the join point shadow at runtime with all advice whose residual dispatch succeeded is called join point

implementation in analogy to "virtual method implementation". Different combinations of advice form a different join point implementation. In this thesis, the combination of all residual dispatches at a join point shadow is conceived as the join point shadow's dispatch function and, thus, the different advice combinations are the targets of this dispatch. This makes the join point shadow the correspondent to the virtual method call site. These correspondences hint that techniques to optimize away dispatching logic in object-oriented programming languages are applicable to aspect-oriented programming languages as well. This assumption is confirmed by the implementations presented in Chapter 4 and their evaluation in Chapter 5.

Similar to virtual method call sites, the dispatch at join point shadows has to be made explicit. For virtual methods the dispatch is explicitly specified by the call's signature and the method implementations in the type hierarchy. Unlike the dispatch functions for virtual methods that only take the type of the receiver into account, the dispatch function of join point shadows can take multiple context values into account. In addition to the receiver's type, the types of the arguments and the active object can be considered. Also other contexts such as the current control flow or the active thread can be considered [AGMO06, HMB+05, VSV+05, SMU+04, PAG03, RS03]: they are dimensions along which dispatch is performed.

The dispatch dimensions mentioned in the previous paragraph are supported by current AOP languages. Other dimensions are perceivable, though, that hint at the capabilities of upcoming and future AOP languages. If, for example, a pointcut language takes into account the history of execution [VSV+05, AAC+05] or the interconnections of objects on the heap [OMB05], dispatch dimensions come into scope that have to be manually implemented as part of the application code with main-stream AO languages. The generalized concept of virtual join point dispatch provides a basis on which such languages can be implemented.

```
1 before() :
2     call(ResultSet Connection.query(String sql))
3     && cflow(void Servlet.doPost(Request, Response)) {
4   // validate the SQL string
5   // if it is invalid, throw an exception
6 }
```

Listing 2.7: Pointcut-and-advice referring to dynamic properties of a join point's context.

To illustrate the concept of virtual join points, Figure 2.3 shows the virtual join point dispatch as it occurs at the join point shadow for the call of

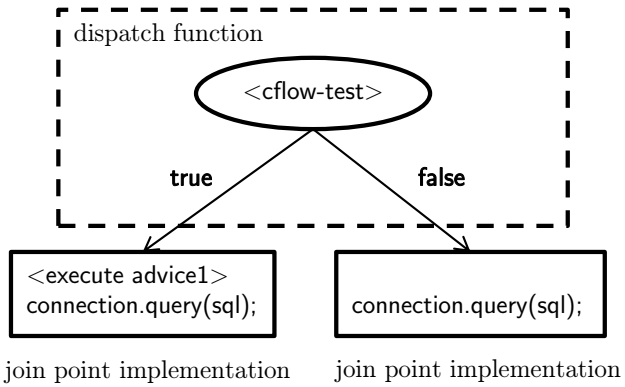


Figure 2.3: Virtual join point dispatch.

method Connection.execQuery whereby the pointcut-and-advice in Listing 2.7 is deployed. In the virtual join points approach, the dispatch function is explicit, symbolized by the decision diagram in the dashed box labeled "dispatch function". The evaluation of this function can have two results in the example—i.e., either the advice is applicable or not—and the result determines which join point implementation is executed at runtime. If the <cflow−test> dynamic property—on which the dispatch solely depends in the example—evaluates to **true**, it is dispatched to the join point implementation that executes the advice and the original join point shadow instruction; otherwise, a join point that only executes the original join point shadow instruction is the dispatch target.

In this thesis, the concept of *envelopes* has been developed which separates join point shadows from the application code. The separation makes the join point shadows including the dispatch to join point implementations explicit and reduces the complexity of manipulating their dispatch function. Envelopes are a level of indirection introduced in the code. Each method call, field read acces and field write access is replaced with a call to a *proxy envelope*, *getter envelope*, respectively *setter envelope* (the latter two are also uniformly called *accessor envelope*). Envelope methods are generated for any method and field declared in a class and call the corresponding method, respectively perform the access to the corresponding field. While envelopes are realized by means of standard Java bytecode, they form a first step towards making join point shadows explicit entities of the application's code.

Listing 2.8 shows the class Services from Listing 2.1 after envelopes have been introduced. Proxy envelopes are inserted for the methods service1 and service2 in lines 12–14 (Listing 2.8) respectively 21–23, and a getter and setter

is inserted for the field connection in lines 4–6 respectively 8–10. The field access sites are re-written to call the accessors in lines (lines 18 and 27).

Proxy envelopes get the name that the enveloped method originally had and the enveloped methods are renamed by adding a prime (') to their name in the example as shown in lines 16 and 25. Because proxy envelopes take over the original method names, method call sites do not have to be re-written but automatically call the proxy. The example also shows the class Connection with the inserted envelope for method execQuery (lines 33–37) into which the residual dispatch is woven (lines 34–35). Actually, envelopes are also generated for constructors including the implicit default constructor, but this is not shown because it would just add unnecessary complexity to the example.

The effect of introducing envelopes is as follows:

- An envelope method can be seen as the manifestation of a join point shadow on its own. Every manipulation of the envelope actually manipulates the join point shadow. Since envelopes are methods on their own and isolated from the base code, join point shadows can be manipulated without modifying the control flow in base modules.

- The weaving process itself becomes extremely simple. Envelope methods have a very primitive sequential structure, free of jump instructions, exception handlers, or debugging information. This means that all precautions with regard to maintaining the control structure of methods affected by weaving can be dropped.

- Inserting envelopes simplifies the search for weaving locations. Method call and field access join point shadows can only occur within envelopes, i.e., envelopes are the only weaving locations and the search for weaving locations can be directly targeted to them.

- The number of envelopes per member is limited: there is only one envelope per field and kind of access; the number of envelopes per method is equal to the number of overridden versions of the method.

```
1 class Services {
2   Connection connection;
3
4   final Connection get_Services_connection() {
5     return connection;
6   }
7
```

```
8     final void set_Services_connection(Connection connection){
9       this.connection = connection;
10    }
11
12    void service1(String sql) {
13      this.service‘(sql);
14    }
15
16    void service1‘(String sql) {
17      //...
18      get_Service_connection().execQuery(sql);
19    }
20
21    void service2(String sql) {
22      this.service2‘(sql);
23    }
24
25    void service2‘(String sql) {
26      //...
27      get_Service_connection().execQuery(sql);
28    }
29  }
30
31  class Connection {
32
33    void execQuery(String sql) {
34      if(/*test for control flow of Servlet.doPost*/)
35        QueryInjectionChecker.before_0(sql);
36      this.execQuery‘(sql);
37    }
38
39    private void execQuery‘(String sql ){
40      //perform the database query
41    }
42
43  }
```

Listing 2.8: Compilation and weaving result in the envelopes approach.

2.4 Prototype Implementation of Envelopes

The indirection of envelope methods can be introduced into application code by transforming each class separately. They are generated in the same class in which the enveloped member is defined and have the same visibility as their enveloped members. Furthermore, it is taken care that at each method call and field access site the envelope is invoked instead of the original call or access.

2.4.1 Generating Envelopes

To introduce proxy envelopes, any method to be enveloped is made private and it is renamed. The generated envelope gets the name and the signature of the original method. The body of an envelope method performs three tasks that are explained in the following by the example of Listing 2.9. The listing shows the bytecode of a plain proxy envelope—i.e., without any residual dispatch woven into it—of the execQuery method from the running example.

1. The arguments for the actual method call are pushed onto the operand stack (lines 2–3).

2. The original method implementation is called (line 4). The arguments to be passed to the original method are those passed to the envelope method.

3. It is returned to the caller (line 5); if the original method has a result value, this value is returned to the caller.

```
1 void execQuery(String sql);
2   aload          0
3   aload          1
4   invokespecial #1 // Services.execQuery‘(String)
5   return
```

Listing 2.9: Bytecode of a plain envelope.

The body of a field envelope method has a similar form as a proxy envelope's body. First the context for the field access is established. This is composed of the object whose field is to be accessed (**this** in envelope) and, in the case of a field-write access, the field's new value. Second, an instruction for direct field access and, third, an appropriate returning instruction follow.

Besides generating the envelope methods, code is re-written in a way that envelope methods are called instead of performing the direct access to the member. Re-writing code to insert calls to envelope methods does not require complex modifications of the original application code: since proxy envelopes have the same name as their enveloped method originally had, method calls do not have to be changed.

To replace direct field accesses with calls to respective envelopes, the code of all methods that access fields is modified. All instructions for direct field access are replaced by instructions calling the corresponding generated envelope methods. The name of the envelope method to be called is derived from the field reference and the kind of access. Thus, it is possible to generate the correct envelope method call even if the class declaring the accessed field is nor yet loaded. The instructions for direct field access can simply be overwritten with the instructions that call the envelope methods because they occupy the same number of bytes.

Envelope methods are generated in a way that facilitates to optimize away the introduced indirection. A JVM using a JIT compiler may decide to inline a method instead of calling it via a dispatch table when the call at hand is non-polymorphic [FGH02]. Small methods are more likely to be inlined because the ratio of additional time for the JIT compilation and the time saved by cheaper dispatch is better than for larger methods. The envelope generation makes use of this knowledge: the generated code embodies hints for the JIT enabling the latter to optimize away the introduced indirections. The call from the proxy envelope to the enveloped method is non-polymorphic; and so is the call to an accessor envelope. This is guaranteed by the way envelopes are generated. By making the enveloped method **private** and the accessor methods **final** a hint is given to the JIT compiler that envelopes can be inlined at call sites. Inlining envelopes basically optimizes them away when no advice are woven in them. Other optimizations that are possible only within the virtual machine are presented in Section 4.3.

2.4.2 Weaving in Envelopes

Due to the simplicity of their code, weaving in envelope methods is straightforward. For every residual dispatch that is to be inserted into an envelope, a separate code block can be generated. It is not necessary to update any other instructions—even if one or more of these blocks contain a branch instruction that skips the advice execution if some dynamic property is not satisfied.

For illustration, consider Listing 2.10 which shows an envelope that contains residual dispatch with a dynamic condition. The pseudo instruction in line 2 represents the test for the control flow. Again the exact implementa-

tion is left undetermined, but it is assumed that the test pushes the value **true** or **false** onto the operand stack. The following instruction (line 3) tests this value and jumps to the instruction in line 6 if the test has failed, which is the first instruction of the plain envelope code.

```
1 void execQuery(String sql);
2   <cflow-check>
3   ifne          +3
4   aload         1
5   invokestatic  #2 // QueryInjectionChecker.before_0(String)
6   aload         0
7   aload         1
8   invokespecial #1 // Services.execQuery'(String)
9   return
```

Listing 2.10: Bytecode of an envelope with residual dispatch.

If another code block for an additional residual advice dispatch is to be inserted between the first advice and the plain envelope code, none of the existing instructions have to be updated. Listing 2.11 shows the result of adding a second residual dispatch code block to the envelope. The branch instruction still skips the execution of the first block (lines 2–5) when the **cflow**-check fails: execution continues with the second residual dispatch block (line 6) before finally reaching the plain envelope block (lines 7–10); this is the intended semantics of inserting a residual dispatch block between two already existing blocks.

```
1 void execQuery(String sql);
2   <cflow-check>
3   ifne          +3
4   aload         1
5   invokestatic  #2 // QueryInjectionChecker.before_0(String)
6   invokestatic  #3 // QueryInjectionChecker.before_1()
7   aload         0
8   aload         1
9   invokespecial #1 // Services.execQuery'(String)
10  return
```

Listing 2.11: Bytecode of an envelope with residual dispatch for two pointcut-and-advice.

2.4.3 Limitations of the Prototype

The prototype implementation described above performs post-compile time weaving allowing to compare the approach with conventional aspect weavers. It has a few limitations with respect to the richness of aspect-oriented concepts that are supported and to the supported kinds of join point shadows. However, representatives of AO concepts that have to be handled differently in envelope-based weaving are realized by this prototype. Thus, it is sufficient to evaluate the performance and power of the envelope-based weaving approach.

In the following, the prototype's implementation is discussed in detail and its limitations are pointed out. Solutions are either outlined here, or will be presented along with the advanced prototypes presented in this thesis.

Possible join point shadows. This first prototype cannot generate envelopes for all kinds of members. Envelope methods are inserted into the class containing the enveloped member. Interfaces can define constants which are represented as (**static final**) fields, but it is not possible to insert envelope methods (which are by nature not **abstract**) into interfaces. Thus, no envelopes are generated for constants defined in interfaces. In order to envelope methods the original ones are renamed. The implementation of **native** methods is resolved by means of a naming scheme [Lia99] and re-naming the methods hinders the resolution.

Without the generation of envelopes for constants from interfaces and for **native** methods, it is not possible to advice accesses to them. In Java 6, it is possible to provide a plug-in for the virtual machine that can influence the look-up of **native** methods. This makes it possible to generate envelopes for them as discussed in this section. Such a plug-in has been developed and is part of the more advanced prototype presented in Section 3.5.1, but it will not be considered in the evaluation of the first prototype which has been developed for Java 5.

call versus execution pointcut designators. In the AspectJ language there are differences between the **call** and **execution** pointcut designators [BFTY04]. The main difference is that AspectJ allows to access the caller object only at join points selected by the **call** designator. Furthermore, both designators resolve to different join points in some cases. **call** selects join points before the target method is resolved—i.e., the static target method is considered—, while **execution** selects join points after method resolution.

As an example of the difference between **call** and **execution** pointcut designators in AspectJ, consider Listing 2.12. The statement in line 18 causes

the first advice (lines 25–27) to be executed, because the static target is the method m in class A. However, this method is not executed at runtime, but the overwritten version from class B, because of polymorphism. Thus, the second advice (lines 29–31) is not executed. Envelopes are generated in a way that the prototype supports the semantics of the **execution** pointcut designator. But the feature of accessing the caller object is still supported.

```
1  class A {
2    void m() {
3      //...
4    }
5  }
6
7  class B extends A{
8    void m(){
9      //..
10   }
11 }
12
13 class Client {
14
15   void main(){
16     ...
17     A a = new B();
18     a.m();
19     ...
20   }
21 }
22
23 aspect SampleAspect {
24
25   before() : call(void A.m()) {
26     // advice$_1$
27   }
28
29   before() : execution(void A.m()) {
30     // advice$_2$
31   }
32 }
```

Listing 2.12: Difference between **call** and **execution** pointcut designators in AspectJ.

Caller context value. In the conventional aspect weaving style, the caller object is a local value available in the method containing the join point shadow selected by **call**. Thus, the weaver can generate code that accesses the local to pass it to the advice. In the envelope-based weaving style, this is not possible, because the weaving is performed inside the envelope method.

To offer access to the caller object the JVM Tools Interface (JVMTI) [JVM] is used, a standard interface also used by debuggers to access the local values from any call frame. The caller object is the first local in the call frame just below the envelope's one.

Special methods. Constructors and static initializers are, at bytecode level, normal methods but with special names. The Java virtual machine specification stipulates that each constructor either calls a constructor from the super class or from the same class as its first statement. This constraint is enforced by the verifier [LY99] of the virtual machine which exploits the naming scheme to identify constructors and static initializers. Thus, renaming the constructor and static initializer methods would cause the verifier to fail. To avoid this, the identity of enveloped methods is changed in another way than re-naming it. In Java bytecode, the identity of a method is not only determined by its name, but by its signature—comprising the name and all parameter types. Thus, adding a parameter also changes a method's identity. In this prototype, this approach is followed in order to preserve the original method names which are required by the verifier.

Reflection. Methods and fields can also be called respectively accessed using the Java Reflection API [Java]. Reflective method invocations automatically call the proxy methods because they have the same name as the method originally had. This is not true for reflective field accesses. They bypass advice for a field access. Also, the transformation adds new methods to a class—the envelope methods—, which are presented to the application when the Java Reflection is used to get a list of methods declared in a class.

2.4.4 Dynamic Weaving in Envelopes

Since the overall goal of this thesis is to support *dynamic* AO languages, it is also to be evaluated how well the envelopes approach performs in a setting of dynamic aspect deployment. For this purpose, the prototype discussed in this section has been integrated as a Java 5 Agent using the bytecode instrumentation package [Ins] to, first, transform the application into the envelope-style and, second, weave advice into envelopes at runtime. Transformation of Java

class files is performed just before the class is defined by the JVM. A representation of the transformed class file is kept in memory which allows for an easy access to envelope methods. When an aspect is deployed, the affected envelope methods are searched for and weaving is performed as described in Section 2.4.2. Classes that contain envelopes which have been modified in the weaving step are redefined using the so-called Class Redefinition [Dmi01] feature of standard JVMs.

2.5 Evaluation of Weaving Approaches

In Section 2.3, the concept of virtual join points is introduced as a means to address the problems discussed in Section 2.2. The prototype of envelope-based weaving is a first step towards execution environments implementing the virtual join points approach. In this section, the prototype is evaluated with respect to its effectiveness in addressing the weaving performance which is one of the problems identified in Section 2.2. The effectiveness of virtual join points in addressing the other problems are addressed in the following chapters.

2.5.1 Evaluated Approaches

To rate the envelopes approach used in a post-compile weaver as well as a runtime weaver, the performance is measured and compared to other weaving approaches. A detailed discussion of the technology of the related approaches can be found in [Hau06].

There are two compilers publicly available for AspectJ-like AOP languages which follow the post-compile weaving strategy: the `ajc` compiler included in the AspectJ distribution [Aspa], and the *AspectBench Compiler* [ACH+05a] (`abc` for short). The `ajc` compiler employs some optimizations targeted at reducing the compilation time [HH04]. In order to cope with the complexity of the bytecode instrumentation, `ajc` uses the BCEL bytecode manipulation toolkit [BCE] to abstract from details of the Java bytecode.

The focus of `abc` is on extensibility. The customization power is derived from Polyglot [NCM03], a compiler framework, and Soot [Soo], a framework for static analyses of bytecode, on which `abc` is built. Often, `abc` generates code that exhibits better run-time performance as compared to `ajc` [ACH+05b]. However, the improved code quality comes at the cost of a compilation time compared to `ajc` [ACH+04]. For this reason, `abc` is not considered in the evaluation whose focus is to compare the weaving performance of different approaches.

Several execution environments with support for dynamic AOP are available. Of these, tow typical ones are considered here.

AspectWerkz [Aspd] transforms possible join point shadows at compile- or load-time. Advice invocations are either woven in during the transformation step or the shadows are merely *prepared* for later attachment of advice. Several other AOP implementations with runtime weaving follow basically the same approach, like JAC [PSDF01] and JBoss AOP [JBo].

AspectWerkz's weaving bears some similarity to the envelopes approach. In both cases generated methods are invoked at join point shadows. However, unlike envelope-based weaving, AspectWerkz does not aim at reducing the number of weaving locations. The purpose of generated methods rather is to facilitate late introduction of advice. Wrappers are generated on the caller site: one wrapper per affected member and per method in which the access occurs. Also, values from the original join point shadow context are passed to the invoked generated method, which requires costly modifications of the method containing the join point shadow.

PROSE [PGA02, PAG03], the second related AO language implementation taken into account, exists in two versions. The one considered here (called PROSE 1) relies on a standard JVM's debugger interface and is unique in that it does not instrument an application's code at all. Upon weaving an aspect into a running application, PROSE 1 registers breakpoints at the join point shadows of the aspect. Once a registered breakpoint is reached, execution branches to the PROSE infrastructure, which looks up the appropriate advice functionality and invokes it. In PROSE 1, no methods ever have to be recompiled, but context switches at debugger breakpoints are very expensive.

The second version of PROSE basically pursues the same strategy for deploying aspects. However, it does not rely on the JVM's debugger interface but provides similar features by means of an extension of the Jikes RVM. This approach is implemented in a way that most virtual machine optimizations are disabled; as a consequence, it cannot be used for a comparison.

2.5.2 Benchmarks

Depending on the setting in which a weaving approach is used, different performance characteristics have to be considered. E.g., for a post-compile time weaver, only the overall time is important; i.e., in the case of envelope-based weaving the generation of envelopes along with the weaving of advice. For a runtime weaver, these are two different concerns. The envelope-generation, or any preparation required by other weaving approaches, slows-down the start-up of the application, when classes are loaded and possibly transformed. The weaving in turn imposes a delay on the executing application. Furthermore,

the infrastructure required to perform the weaving, e.g., auxiliary data structures holding a suitable representation of classes or the facility to redefine classes, affects the performance of the executed application and also increases the memory requirement. This is of no concern at all for a post-compile weaver.

Because transforming an application to use envelopes and weaving advice does not require to re-write potentially complex application code, not much infrastructure is required. To show that this pays of in terms of a low overhead the prototype for envelope-based weaving is compared with AspectJ 1.2.1 [KHH+01], AspectWerkz 2.0 [Aspd], PROSE 1.2.1 [PRO]. Three different measurements are performed:

1. The performance in the context of both *static weaving* and *dynamic weaving* is measured.

2. The *memory consumption* is measured.

3. The *runtime impact* of aspect-oriented execution environments in terms of the overall performance of applications is measured.

All measurements have been performed on a Dual Xeon workstation (3 GHz per CPU) running Linux 2.4.27 with 2 GB memory. AspectJ and AspectWerkz have been run on the Sun HotSpot JVM, version 1.5.0_01, and PROSE has been run on version 1.4.2_08 of that VM (newer VMs do not support PROSE). The HotSpot JVM has been executed in client mode as well as in server mode, which performs immediate JIT compilation and more aggressive inlining.

Static weaving. Static weaving performance is measured by compiling the Xalan-J XSLT parser [xal] which consists of nearly 1,200 classes. Different compilation scenarios are used: without any aspect (*none*), with an aspect advising calls of one specific method (*call-one*), and with an aspect advising all calls to public methods declared in classes in the Xalan packages (*call-all*). The advice simply increases a counter. In AspectJ, advice can be woven at method call sites or execution sites. In the latter case the number of weaving actions is reduced, as it is in the envelope-based weaving approach. For this reason, `ajc`'s overhead is also measured when one specific execution site (*exec-one*), and execution sites of all public methods in Xalan packages (*exec-all*) are advised.

For performance reference values, Xalan was compiled by the Java compiler from the Eclipse JDT, version 0.452_R30x, which takes 6.4 seconds. For the different compilation scenarios, Table 2.1 summarizes the overheads

	Envelope-Based Weaving	`ajc`
no aspect	34.5%	63.9%
exec-one	–	111.5%
call-one	33.7%	187.6%
exec-all	–	219.4%
call-all	35.9%	289.0%
memory overhead	67 MB	121 MB

Table 2.1: Results of the measurements of static weaving.

in compilation time compared to the reference value incurred by the `ajc` compiler and the envelope-based weaving prototype. Even in the absence of aspects, the compilation with `ajc` is about 63.9% slower than compilation with the Eclipse Java compiler. This is because the weaver scans all binaries produced by the source code compiler for weaving specifications. This is similar for the envelope-based weaving prototype; although the prototype is faster in scanning the binaries.

Like with `ajc`, the overhead of the envelope-based prototype generally increases with the number of affected weaving locations. However, its increase rate is considerably smaller. Furthermore, the maximum overhead imposed by the prototype is 35.9% which is just about an eighth of the maximum overhead imposed by the `ajc` compiler.

Memory consumption. When comparing memory consumption, the prototype based on envelopes prototype benefits from the avoidance of memory-intensive data structures as needed by `ajc` during weaving. Table 2.1 also shows the memory required by the envelope-based weaving prototype and the `ajc` compiler in the compilation scenario *call-all*. With 121 MB of memory, the `ajc` compiler requires nearly twice the memory of the prototype.

Dynamic weaving. Dynamic weaving performance has been measured for AspectWerkz, PROSE and the prototype using envelope-based weaving. The `RayTracer` benchmark from the JavaGrande benchmark collection [Javb] has been used for this measurement. The application, consisting of 11 classes, has been extended with an aspect that advises all *calls* to public methods declared in the raytracer package with a simple counter-incrementing advice. For these measurements, the time needed to *prepare* the application classes is distinguished from the time for actually *deploying* an aspect at runtime; the results are shown in Table 2.2. PROSE does not conduct preparation.

	Envelope-Based Weaving	AspectWerkz	PROSE
preparation	66 ms	376 ms	N/A
deployment	101 ms	424 ms	677 ms

Table 2.2: Results of the measurements of dynamic weaving.

Preparation takes 376 ms in AspectWerkz and only 66 ms in the envelopes prototype. Deployment takes an additional 424 ms in AspectWerkz, 677 ms in PROSE and 101 ms in the envelopes prototype.

Runtime overhead. The runtime overhead inflicted by the different approaches is evaluated in another experiment. Envelopes add an indirection, affecting the runtime behavior of the compiled program. However, due to modern just-in-time (JIT) compiling virtual machines [PVC01, ACL+99], the introduced indirection only has small impact on the runtime performance, as the evaluation confirms.

Furthermore, the JVMTI facility required for enabling access to the caller object may cause some runtime overhead. To measure the impact of the above on the runtime performance of applications that do not deploy aspects and to compare the envelopes prototype with other AOP environments, the SPECjvm98 benchmark suite [SPE] is used. The benchmark applications have been run both the client and server versions of HotSpot 1.5.0. with and without inserted envelopes.

The results indicate that envelopes and the JVMTI facility impose an overhead of 7.7 % on a running application. The overhead is however reduced to 5.2 % when an application with envelopes is run on the HotSpot server VM.

The overhead imposed simply by enabling the JVMTI facility is also measured by executing the SPEC JVM98 benchmarks with the facility turned on and off on the HotSpot 1.5.0 JVM. In average the execution was slowed-down by 1.6% when run in server mode and not at all when run in client mode.

Chapter 3

The Aspect Language Implementation Architecture

In order to support virtual join points, the dispatch of join point shadows has to be specified declaratively. Since aspect-oriented programming languages are expressive and evolving, a powerful and extensible model is required for this specification. Therefore, a fine grained meta-model for pointcut-and-advice is presented in this chapter.

Furthermore, a framework is presented that abstractly implements generic control flows of execution environments for dynamic aspect-oriented programs defined in terms of the meta-model. To evaluate the meta-model and the framework with respect to their expressiveness and extensibility, a default instantiation of the framework is developed and representative modern AOP languages are mapped to the meta-model. It is also shown by means of a case study that the framework is suitable to separate the design and optimizing implementation of new AO language features.

Parts of this chapter have been published in the following papers.

1. *Christoph Bockisch and Mira Mezini. A Flexible Architecture For Pointcut-Advice Language Implementations. In* Proceedings of the Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms. *ACM Press, 2007*

2. *Christoph Bockisch, Mira Mezini, Kris Gybels, and Johan Fabry. Initial Definition of the Aspect Language Reference Model and Prototype Implementation Adhering to the Language Implementation Toolkit Architecture. Technical Report AOSD-Europe-TUD-7, Technische Universität Darmstadt, 2007*

3. *Christoph Bockisch, Mira Mezini, Wilke Havinga, Lodewijk Bergmans, and Kris Gybels. Reference Model Implementation. Technical Report AOSD-Europe-TUD-8, Technische Universität Darmstadt, 2007*

4. *Christoph Bockisch, Andreas Sewe, Mira Mezini, Arjan de Roo, Wilke Havinga, Lodewijk Bergmans, and Kris de Schutter. Modeling of Representative AO Languages on Top of the Reference Model. Technical Report AOSD-Europe-TUD-9, Technische Universität Darmstadt, 2008*

5. *Andrew Jackson, Siobhán Clarke, Matt Chapman, and Christoph Bockisch. Deliver Preliminary Support for Next-Priority Use Cases. Technical Report AOSD-Europe-IBM-80, IBM UK, 2007*

6. *Andrew Jackson, Siobhán Clarke, Matt Chapman, Andy Dean, and Christoph Bockisch. Deliver Preliminary Support For Top Priority Use Cases. Technical Report AOSD-Europe-IBM-64, IBM UK, 2006*

3.1 Common Features of Aspect-Oriented Languages

As motivated in the previous chapter, a declarative specification of pointcut-and-advice in the intermediate representation of a program is desirable for flexible and efficient support of dynamic AO concepts. Therefore, in this chapter the *Aspect Language Implementation Architecture* (ALIA) is proposed where the aspect-oriented concepts stay first-class until execution. Centric to this architecture is a language independent meta-model of core aspect-oriented concepts which acts as an interface between front-ends and execution environments, i.e., as an intermediate representation for aspect-oriented programming languages.

Figure 3.1 shows an overview of the ALIA approach. When a program gets compiled, the compiler generates an intermediate representation of the program consisting of two parts: the intermediate code of the object-oriented parts of the program and a model of the aspect-oriented parts. Besides the compiler and the execution environment, the language implementation comprises a meta-model that prescribes the structure of the aspects' model. The execution environment knows how to execute models provided in terms of the meta-model.

The usage of a meta-model facilitates extensibility. While the meta-entities cover the abstract core concepts common to most AO languages, they can be specialized to cover the advanced features of concrete language constructs. By using the specialized entities, a model of the aspect-oriented constructs of a program can be built and provided to an execution environment as first-class objects.

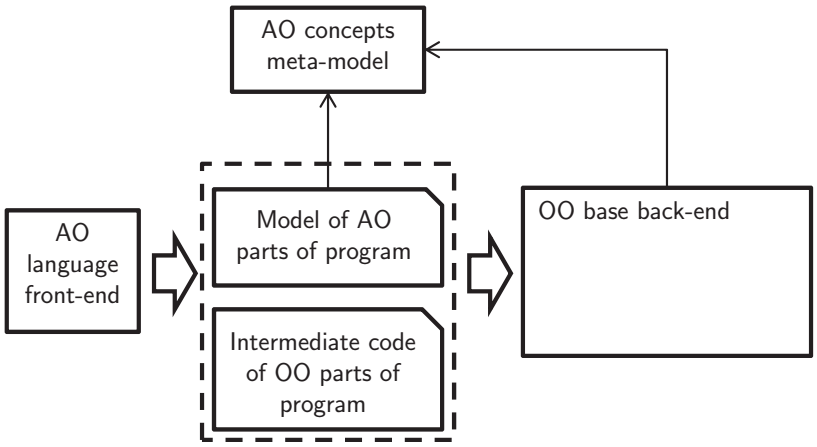


Figure 3.1: Compilation and execution of AO programs in ALIA.

A framework for language implementations that adhere to the ALIA approach including the meta-model has been developed in this thesis and is presented in Sections 3.2 and 3.3.

In order to identify the core concepts, state-of-the-art aspect-oriented programming languages are investigated here. The languages considered are *AspectJ* [KHH+01], *CaesarJ* [AGMO06], *Compose** [BA01] and *JAsCo* [SVJ03]. AspectJ is the most widely used AO language [Ker]; the other three languages provide more advanced concepts and are also considerably widely used [BJC08]. The meta-model discussed in Section 3.2 has been developed such that the concepts identified in this section can be expressed in the meta-model.

3.1.1 AspectJ

An aspect in AspectJ is a type that can contain pointcut-and-advice and so-called *static crosscutting* declarations [KHH+01] in addition to the standard class members; static crosscutting refers to the definition of methods and fields in an aspect, which are added to other types. The static crosscutting features of AspectJ are not relevant in the context of this thesis as only support for pointcut-and-advice is investigated. Based on [KHH+01] and the current AspectJ Programming Guide [Asp08], the elements of the AspectJ language that determine pointcut-and-advice are presented in the following.

Join Point Model. The join point model determines which kinds of points in the program execution can be selected as join points. In AspectJ, these are the *call* or *execution* of *methods* and *constructors*, the *execution of static*

initializers, the *execution of exception handlers*, and *read* and *write* access to *fields*. Listing 3.1 shows examples of possible join point shadows in AspectJ. The lines 3–4 form a method execution join point shadow, while line 18 is a join point shadow of a call to the same method. Lines 12–13 form the join point shadow of an exception handler. The example shows that in AspectJ join point shadows don't have to be single instructions but can also be ranges of instructions.

```
1  class C {
2    void method() {
3      someStatement;
4      return;
5    }
6
7    void method2() {
8      try {
9        someStatement;
10     }
11     catch(Exception exception) {
12       someStatement;
13       someStatement;
14     }
15   }
16
17   void main() {
18     method();
19   }
20 }
```

Listing 3.1: Code containing different kinds of join point shadows.

Pointcuts. Pointcuts are expressions that select a set of join points, i.e., executions of join point shadows with certain properties; the operands of these expressions are the *primitive pointcut designators* which can be combined by the boolean operators *and* (&&), *or* (||), and *not* (!).

There are two kinds of primitive pointcut designators. The first one selects join points only in terms of their join point shadows. Their name determines which kind of join point shadow is selected, e.g., calling a method or reading a field. The designators are parameterized with a pattern which is matched against the signature of, e.g., the called method at a call join point shadow. The pattern language is a subset of regular expressions plus means to express

sub-type relationships: the type pattern SomeType+ matches all types whose name is SomeType or which have a super-type of that name.

The other kind of pointcut designators selects join points based on runtime properties of the program at a join point. The designator's name specifies which runtime value is considered; that can be the **this** object, the receiver object or argument values, or the current control flow (in terms of methods currently executing on the call stack). The designators are parameterized with a constraint on the runtime value like the required dynamic type.

Advice. Advice are pieces of code which are associated with pointcuts. At every join point selected by a pointcut, the associated advice is executed. Furthermore, it is possible to specify whether the advice is to be executed **before**, **after** or **around** the actual join point. **around** means that the actual join point is replaced with the advice and the advice can control when the join point is to be executed—if at all. The special form **proceed** is used in the advice body where the actual join point is to be executed. Furthermore, it is possible to specify that an **after** advice is only to be executed when the join point terminated normally or abnormally, or in both cases. An **around** advice using **proceed** can change the values bound from the join point's context and thereby can execute the original join point in a modified context.

Pointcut Parameters. Values from the context in which a selected join point is executed—e.g., the **this** or receiver object—can be made available to an advice in terms of pointcut parameters. Listing 3.2 shows an example of a pointcut-and-advice that uses pointcut parameters. The keyword **before** specifies that the advice is to be executed before join points selected by the pointcut following the colon. The pointcut parameters are declared in the parentheses behind this keyword (line 1) similar to method parameters.

```
1 before(String sql):
2   call(ResultSet Connection.execQuery(CharSequence))
3     && args(sql)
4 {
5   if (isAttack(sql))
6     throw new SQLInjectionException();
7 }
```

Listing 3.2: Pointcut-and-advice using pointcut parameters.

The **args** primitive pointcut designator in line 3 uses the parameter name sql declared in the pointcut parameters in line 1 and binds the single argument from the join point's context to it. As a consequence, this value can

be used by the advice as in line 5. Furthermore, the **args** designator imposes a condition on the pointcut: it only matches if the value to be bound has the type specified in the pointcut parameter, String in the example. For **after** advice, it is also possible to use the join point's result value as pointcut parameter or—if the join point terminated abnormally—the exception thrown during the join point's execution.

Reflection. Besides using pointcut parameters, values from the join point context can also be accessed from within an advice reflectively in terms of the **thisJoinPoint** special form. It allows to access all values that can also be bound by primitive pointcut designators as well as meta-information about the current join point like the source location of its join point shadow.

Advice Precedence. The order of advice, applicable at the same join point, can be specified in terms of precedence at the granularity of aspects in AspectJ. When aspect A precedes another aspect B all pointcut-and-advice defined in A precede the pointcut-and-advice in B.

For **before** and **around** advice, the preceding advice is executed before the preceded one. Furthermore, should an **around** advice precede other advice, those are only executed when the **around** advice performs **proceed**. For **after** advice, the preceded advice are executed first. The precedence of pointcut-and-advice defined in the same aspect is implicitly determined by the order of their appearance in the source code. As an example, consider the example aspects in Listing 3.3, line 1–28, which all declare pointcut-and-advice for the same join point. The **declare precedence** statement in line 5 determines the advice's precedence. The output of this example is shown in Listing 3.4.

```
1  aspect Before1 {
2    before() : execution(* Base.m()) {
3      System.out.println("before 1");
4    }
5    declare precedence: Before1, Before2, Around, After1, After2;
6  }
7  aspect Before2 {
8    before() : execution(* Base.m()) {
9      System.out.println("before 2");
10   }
11 }
12 aspect Around {
13   void around() : execution(* Base.m()) {
14     System.out.println("around - 1. part");
15     proceed();
```

```
16      System.out.println("around − 2. part");
17    }
18  }
19  aspect After1 {
20    after() : execution(∗ Base.m()) {
21      System.out.println("after 1");
22    }
23  }
24  aspect After2 {
25    after() : execution(∗ Base.m()) {
26      System.out.println("after 2");
27    }
28  }
29
30  class Base {
31    public void m() {
32      System.out.println("method");
33    }
34  }
```

Listing 3.3: AspectJ code using a precedence declaration.

```
1  before 1
2  before 2
3  around − 1. part
4  method
5  after 2
6  after 1
7  around − 2. part
```

Listing 3.4: Output of executing the sample program in Listing 3.3.

Aspect Instantiation. As already mentioned, aspects are types in AspectJ and pointcut-and-advice are their members. This is an extension of the concepts of *classes* respectively *virtual methods*. The latter always execute in the context of an instance of their declaring class. Similarly an advice is always executed in the context of an instance of its declaring aspect. However, since advice are implicitly invoked when their associated pointcut matches, the receiver object in whose context the advice is executed also must be determined implicitly. Consequently, aspects are implicitly instantiated on demand; the strategy to retrieve these instances is called *aspect instantiation* in AspectJ.

The strategy is defined as a modifier of the aspect declaration, e.g., by the **issingleton** keyword in line 1, Listing 3.5. **issingleton** means that only one instance of the aspect is created and advice are always executed in the context of this instance. Thus, in the example in Listing 3.5 the advice always refers to the same counter in line 4. Other strategies are to select the aspect instance based on the **this** or receiver object, or the control flow at the join point.

```
1 aspect Before1 issingleton() {
2   private int counter;
3   before() : execution(* Base.m()) {
4     counter++;
5   }
6 }
```

Listing 3.5: AspectJ code declaring strategy for aspect instantiation.

3.1.2 CaesarJ

CaesarJ [AGMO06] also offers pointcut-and-advice as well as static crosscutting features. The static crosscutting of CaesarJ is much more sophisticated than AspectJ's, but this is out of the scope of this thesis. The pointcut-and-advice features of CaesarJ are basically the same as those of AspectJ with the difference that it is possible to *control the instantiation* of aspects and that they can be *deployed and undeployed at runtime* rather than being enabled during the complete program execution like in AspectJ. Beyond enabling and disabling aspects dynamically, CaesarJ also allows to deploy aspects only within a certain *scope* like the current thread. Listing 3.6 shows an example aspect in the CaesarJ language and its thread-local deployment.

```
1 public cclass CompanyLogger {
2   before() : execution(* company.*.*(..)) || execution(company.*.new(..)) {
3     System.out.println(thisJoinPoint.toString());
4   }
5 }
6 public class Main {
7   public static void main(String args[]) {
8     doSomething();
9     deploy(new CompanyLogger()) {
10      doSomething();
11    }
12    doSomething();
```

```
13    }
14  }
```

Listing 3.6: CaesarJ code performing thread-local aspect deployment.

The aspect type in CaesarJ is called **cclass** as shown in line 1, Listing 3.6; it can contain pointcut-and-advice in the syntax of AspectJ (lines 2–4). In line 9, first, an instance of the aspect is created and, second it is deployed thread-locally for the execution of the block in lines 9–11. It is also possible to write aspects with the same semantics as in AspectJ if the keyword **deployed** is placed in front of the **cclass** definition. Then the aspect is implicitly instantiated and active during the complete program execution.

3.1.3 JAsCo

Similar to CasearJ, the AO language JAsCo [SVJ03] provides aspect composition features more advanced than AspectJ. It adds so-called *aspect beans* to the Java language; these contain **hook***s* which are comparable to advice in AspectJ. However, **hook**s are dynamically associated with pointcuts by **connector***s*.

As an example, consider Listing 3.7 which shows a basic logger that outputs a message for each called method written in JAsCo syntax. The class Logger is an aspect bean and contains the **hook** defined in lines 2–9 which is the definition of the aspect bean's crosscutting behavior. Line 4 in the **hook**'s constructor (lines 3–5) declares an *abstract pointcut*, i.e., a pointcut that is parameterized with a pattern passed as an argument to the **hook**'s constructor at runtime. By calling a **hook**'s constructor, the abstract pointcut is concretized and associated with the advice (lines 6–8) defined in the **hook**.

The **hook** is instantiated twice (lines 13 and 14) with different patterns in a **connector** module, lines 12–18 in Listing 3.7. The first instance captures all executions of methods whose name starts with set while the second one captures all executions of methods with the exact name setX. The order of advice execution at shared join points, is specified in terms of an explicit itemization as shown in lines 16–17.

```
1 public class Logger {
2   hook Logging {
3     Logging (method(..args)) {
4       execution(method);
5     }
6     before() {
```

```
7         System.out.println("Executing " + thisJoinPoint.getSignature());
8       }
9     }
10  }
11
12  static connector LoggingDeployer {
13    Logger.Logging aspect1 = new Logger.Logging(* *.set*(*));
14    Logger.Logging aspect2 = new Logger.Logging(* *.setX(*));
15
16    aspect1.before();
17    aspect2.before();
18  }
```

Listing 3.7: JAsCo code declaring precedence of pointcut-and-advice.

3.1.4 Compose*

Compose* [BA01, dRHH+08] is a language for composition filters. Composition filters specify aspects by defining so-called *filter modules*. *Filters* defined in such modules intercept method calls (or conceptually: messages) and redefine the action to be performed. There are two different kinds of filters: *input* filters intercept method calls received by an object and *output* filters select method calls sent from an object. The former ones are comparable to **execution** join point shadows in AspectJ, the latter to **call** shadows. Figure 3.2 shows a schema of how filters are applied in Compose*. The definition of when filters are applied is relative to a specified object, called **inner**. The input filters are those applied on calls to or returns from a method invoked on **inner**. Output filters are applied on invocations that are performed by the **inner** object. The term *calling flow* refers to the control flow from the caller to the method invoked on **inner**, respectively, the *returning flow* is the flow after the method has finished execution and the control is returned to the caller.

For every filter the type is specified, as well as a condition and two actions; the first action is executed when the condition is satisfied (i.e., the filter accepts), the second if the condition is not satisfied (i.e., the filter rejects). The most commonly used filter type is *Dispatch*, declaring a redirection of the original method call whereby the actions specify the target of the redirection.

The specialty of Compose* is that the actions performed by filters are defined in a declarative way. Besides the *Dispatch* filter type, there are, e.g., the *Error*, *Wait*, *Before* and *After* filter types. *Error* filters abort the current join point including all subsequent filters, *Wait* filters queue the join

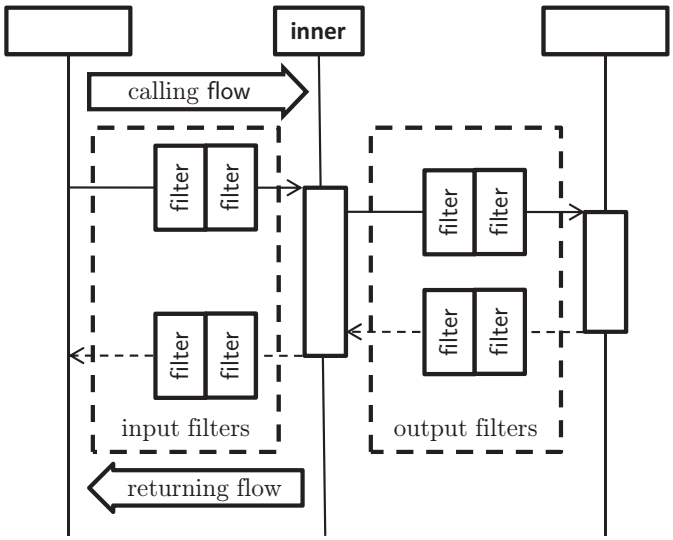


Figure 3.2: Schema of how filters are applied in Compose*.

point's execution until the condition is accepted, and *Before* and *After* filters call an additional method before or after the intercepted method call. This declarative specification enables a reasoning about the program's semantics with the aspects applied.

The high-level filters of Compose* can *influence each other*. They can, e.g., be mutually exclusive, and may also exclude the actual join point itself. For example, the *Error* filter throws an exception and terminates further execution of filters at the join point. Another example for filter functionality that can be declaratively specified is the *Meta* filter, which represents a specialized advice having reflective access to the join point.

The dynamic part of filter conditions is an expression of *basic conditions* which are predicates on the program state. Basic conditions are implemented as virtual methods in Compose*; it is possible to declare on which receiver object the method is to be invoked. Similarly the receiver object can be declared for advice or dispatch actions, which are also implemented as methods. The receiver may either be **inner**, **self**, an **external** or an **internal** object. **inner** and **self** refer to the object that is the receiver of the call to which the filters are applied; the difference is that a dispatch to **self** causes filters to be applied anew, while a dispatch to **inner** invokes the methods without applying filters. **external** and **internal** objects are specified in the **concern** module. For an **external** object, the descriptor of a static method is provided that has to be called to retrieve the appropriate receiver object. For **internal** objects, only the fully qualified class name is provided and the receiver object is implicitly created.

3.1.5 Summary of language features

The core concepts identified in this thesis by observing the above AO languages are: advice actions, join point shadows, static and dynamic properties of matching join points, context reification at join points, strategies for (implicitly) instantiating aspects and receiving the instances, advice ordering at shared join points, and dynamic and scoped aspect deployment. These concepts are covered by the meta-model presented in the following section.

3.2 The Language Independent Aspect Meta-Model

The *Language Independent Aspect Meta-Model* (LIAM) is developed in this thesis and captures the aspect-oriented core concepts identified in the previous section. The concrete AO concepts of the above described languages are mapped to LIAM and for some concepts the mapping is presented here to illustrate the meta-model. Especially, throughout this section the AspectJ language constructs and their mapping to LIAM are used to exemplify such a mapping. In Section 3.4, mappings of the more advanced languages CaesarJ, JAsCo and Compose* are presented to emphasize that the meta-model is appropriate to express state-of-the-art aspect-oriented languages.

Terms like pointcut-and-advice are usually used with the intuition of pointcuts and advice in the AspectJ language. However, the meta-model developed here has a slightly different granularity. For example, in AspectJ the order of different advice applicable at the same join point can only be determined per aspect whereas in LIAM ordering is a property of "pointcut-and-advice". Thus, a more concise terminology is used here, which is different from the AspectJ related one, in order to avoid confusion of terms. The meta-model is implemented in Java whereby most of the meta-entities are abstract classes.

Figure 3.3 shows the meta-entities defined in LIAM as well as their relationships. To exemplify the usage of the entities Figure 3.4 shows an object model that represents the sample aspect in Listing 3.8 copied from Section 2.1.1. Assume the aspect is compiled to the class QueryInjectionChecker that contains the **static** method before_0(String sql) which is compiled from the advice's body as in Listing 2.6. The object diagram in Figure 3.4 is explained along with the following discussion of the meta-entities.

```
1 aspect QueryInjectionChecker issingleton() {
2   before(String sql):
```

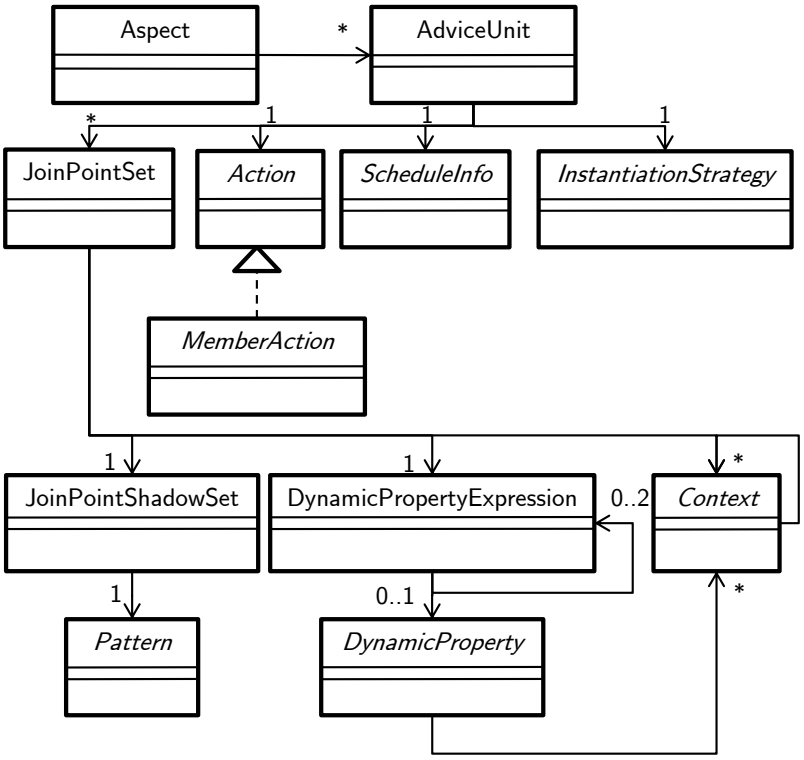


Figure 3.3: Entities of the Language Independent Meta-Model.

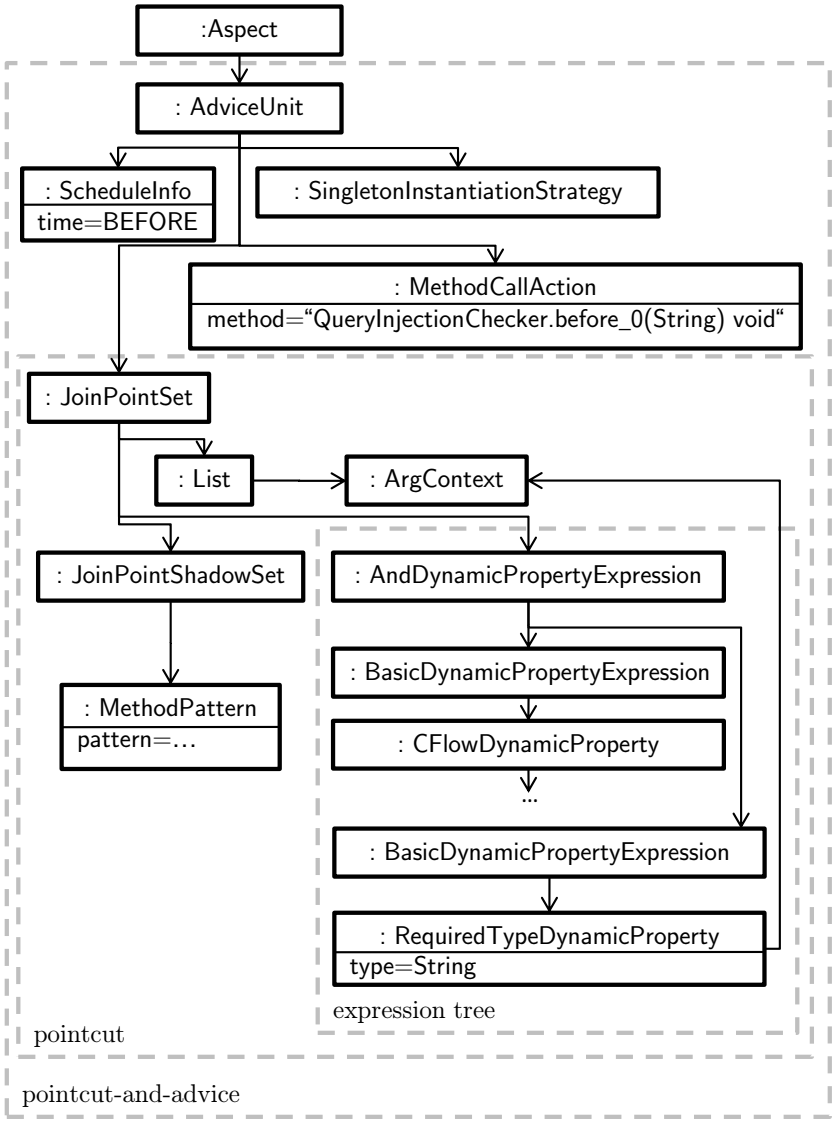


Figure 3.4: Example of a model in LIAM.

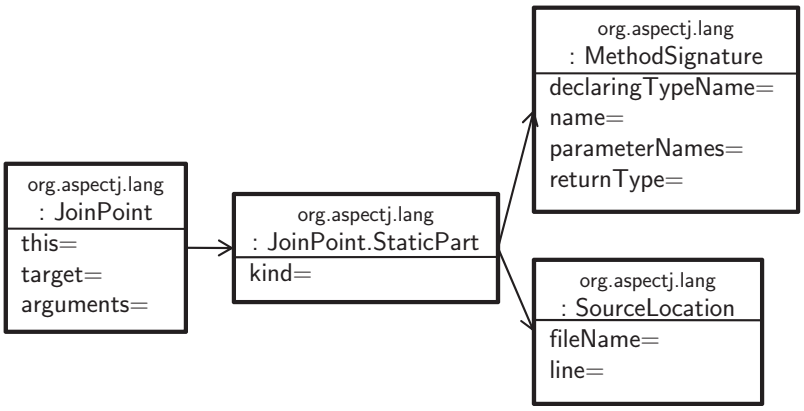


Figure 3.5: Runtime object model of JoinPoint for the ThisJoinPointContext.

```
3       call(ResultSet Connection.execQuery(CharSequence))
4         && args(sql)
5         && cflow(void FacadeServlet.doPost(Request, Response))
6     {
7       if (isAttack(sql))
8         throw new SQLInjectionException();
9     }
10  }
```

Listing 3.8: Aspect in AspectJ syntax.

Context. As seen in the previous section, many features of AO languages depend on runtime values from the context of the current join point. The meta-entity Context models this. Specializations of Context model concrete values like the argument values passed to a called method. Context is used by several other meta-entities discussed below, including Context itself. Composite Contexts model composite values like the JoinPoint object that is accessible in the AspectJ language by means of the **thisJoinPoint** special form. The JoinPoint object is a composition of all values available at a join point; Figure 3.5 shows an object diagram of such a value as it can be used by an advice. Figure 3.6 shows a model of the composite ThisJoinPointContext entity which can be used in a LIAM model to specify that a JoinPoint object has to be exposed at a join point.

Dynamic Property. The DynamicProperty meta-entity models a condition that can be satisfied by a property of the context in which a join point is executed at runtime. In Listing 3.8, line 4, the **args** pointcut designator plays

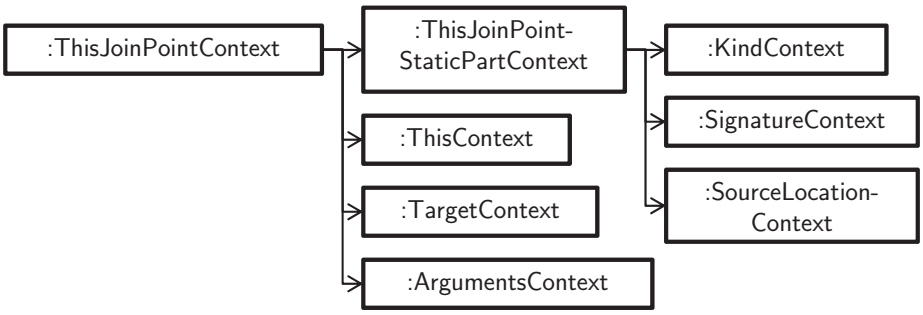


Figure 3.6: Object model of ThisJoinPointContext context in LIAM.

two roles, restricting the runtime type of the argument passed to the execQuery method and binding the value for use by the advice. In LIAM this is split up into several entities. As illustrated by the example LIAM model in Figure 3.4, the value is modeled as a Context entity, ArgContext in the example, used in two different places: the JoinPointSet and the RequiredTypeDynamicProperty both refer to the ArgContext. The former relationship means that the value is exposed to advice; the RequiredTypeDynamicProperty with the associated ArgContext means that at runtime the type of the argument is tested.

Dynamic Property Expression. Constraints for join points based on runtime values are represented by DynamicPropertyExpression forming an expression tree with DynamicPropertys as its leafs. The inner nodes are either an *and* or an *or* operator; negations cannot be inner nodes, but a leaf can also be the negation of a DynamicProperty. In Figure 3.4, the data structure marked by the box *expression tree* is an example of the sub-pointcut in lines 4–5, Listing 3.8. Specializations of the DynamicProperty meta-entity can have associated Contexts and represent concrete constraints for join points. In the example, the constraint of a type requirement for the argument value is expressed. The CFlowDynamicProperty is only partly shown in Figure 3.4 for brevity.

Join Point Shadow Set and Pattern. The JoinPointShadowSet meta-entity is associated with a Pattern object specifying lexical properties of matching join points. It embodies the quantification over join point shadows, i.e., instructions in the program. There is a hierarchy of Patterns implemented in LIAM which reflects the different possible join point shadows—like method calls or field reads—and their lexical properties—like their declaring class or their name. In the example, Figure 3.4, a MethodPattern is used to match method calls.

Join Point Set. A JoinPointSet associates a JoinPointShadowSet with a list of Contexts and a DynamicPropertyExpression. The intuition behind this is that the first two entities select a set of join points matching a lexical pattern and satisfying a dynamic constraint; at these join points the specified runtime values are exposed. A JoinPointSet corresponds to a pointcut sub-expression in AspectJ that consists of a single statically evaluable pointcut designator—e.g., a **call** or **get** designator—and an expression of pointcut designators that cannot be statically evaluated.

A full pointcut is represented by a collection of JoinPointSet objects in the meta-model. Every such object represents a single or-clause. Thus, this collection is the union set of all join points selected by any of its JoinPointSet objects. In AO language implementations based on weaving, join points are determined as a combination of a join point shadow and residual dispatch. In LIAM the residual dispatch is modeled as a DynamicPropertyExpression. At matching join points, the values modeled by the list of Contexts are exposed to the advice. In Figure 3.4 the collection of JoinPointSet and its associated meta-model entities are marked by the *pointcut* box. In Section 3.4 an automatic translator from AspectJ code to LIAM entities is presented which shows that all AspectJ pointcuts are representable in the meta-model.

Action and Instantiation Strategy. Sub-classes of Action arbitrarily determine what actually should happen as an advice. Also part of LIAM are concrete actions, namely a class hierarchy of MemberActions. A member action can be reading or writing a field, or calling a method. The accessed members can either be instance members, i.e., a virtual method or an instance field, or static members. In the case of instance members, an InstantiationStrategy determines how to retrieve the object whose member is to be accessed. In the example LIAM model in Figure 3.4 the SingletonInstantiationStrategy object reflects the **issingleton** keyword in line 1, Listing 3.8; this means that the advice is always executed in the context of the same aspect instance.

Advice Unit and Schedule Information. A collection of JoinPointSets is associated with an advice Action by an AdviceUnit entity, which also refers to a ScheduleInfo meta-entity. AdviceUnit captures the pointcut-and-advice concept in AspectJ while the ScheduleInfo defines relationships between Actions applicable at the same join point shadow, e.g., the order or mutual exclusion of Actions. The framework for executing LIAM models, discussed in the next section, also handles the actual join point event as an action at a join point shadow just like the ones added to a shadow because of a deployed AdviceUnit. Consequently, ordering actions includes arranging them relative to the join

point's action as is determined by the keywords **before**, **after** and **around** in AspectJ.

Aspect. An Aspect is a logical group of AdviceUnits to be deployed together. JoinPointSet, JoinPointShadowSet, AdviceUnit and Aspect provide logical groupings of entities of the meta-model and cannot be sub-classed.

3.3 The Framework for Implementing Aspect Languages

The meta-model LIAM presented in the previous section enhances the intermediate representation of programs and preserves the high-level aspect-oriented constructs from the programs' source code. Intermediate representations serve to decouple compilers and execution environments, i.e., the same intermediate code can be executed by multiple different execution environments with different characteristics. They may support different platforms or may exhibit an especially high performance for certain kinds of applications, for example. Thus, it is also desirable that multiple execution environments support the execution of applications specified in terms of LIAM.

To support this, a framework is developed providing a generic implementation of common functionality required to execute programs with a LIAM-based intermediate representation of aspects. The framework, called *Framework for Implementing Aspect Languages* (FIAL) is implemented in Java and is supposed to be used in execution environments with support for dynamic aspects, i.e., the pointcut-and-advice flavor of aspect-oriented languages with dynamic deployment of aspects. Figure 3.7 shows how the responsibilities are distributed between the FIAL framework and its instantiations.

FIAL implements several *components* required to execute dynamic aspect-oriented programs as well as *common generic work flows*. An execution environment has an internal meta-representation of the program it executes. In Java there are meta-entities for classes, methods and fields. In FIAL-based execution environments, LIAM-models act as a meta-representation for the aspect-oriented parts of the executed program. In addition to the presented entities, such an execution environment also has to manage the join point shadows in the executed program in order to allow for the aspects to be applied there. Common work flows are implemented in the framework in terms of template methods and refined in the concrete execution environment by implementing the provided call-backs.

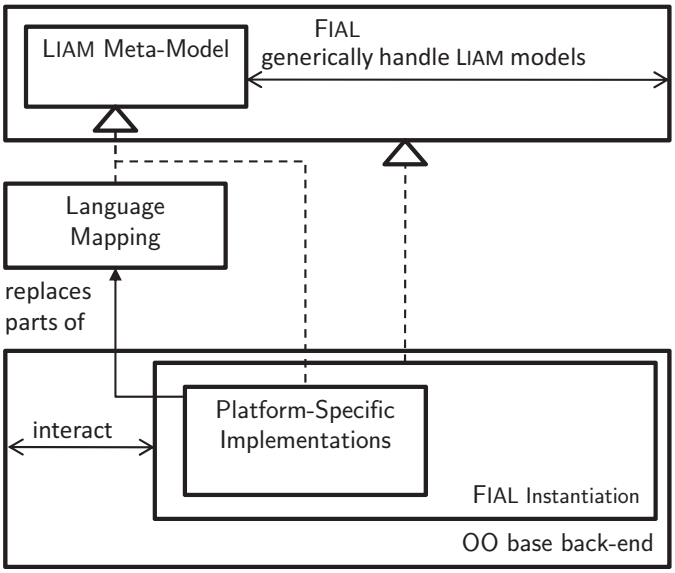


Figure 3.7: Interaction between FIAL and its instantiation.

3.3.1 FIAL's Execution Model

FIAL has two main components that act as interface to its instantiation and to the executed application. The *Factory component* generates LIAM models; since it is implemented as an abstract factory [GHJV95] it can be overwritten by instantiations of the framework in order to ensure that the LIAM entities are implemented in a way appropriate for the concrete execution environment.

The *System component* manages the join point shadows that are present in the executed program in terms of JoinPointShadow objects. Furthermore, it evaluates JoinPointShadowSet objects of LIAM models, and manages deployment and undeployment of aspects. A singleton instance of System is created and used throughout FIAL respectively its instantiations.

Each JoinPointShadow objects has a Signature object encoding which kind of action is performed (i.e., method call, field read access or field write access) and the signature of the called method, respectively accessed field. To evaluate a JoinPointShadowSet FIAL matches the Signatures of all JoinPointShadows against the JoinPointShadowSet's Pattern. Although, in general, FIAL is not restricted to any specific join point model, implementations of Signature and Pattern are provided by default that join point model of *method calls*, *constructor calls*, *static initializer executions*, *field reads* and *field writes*.

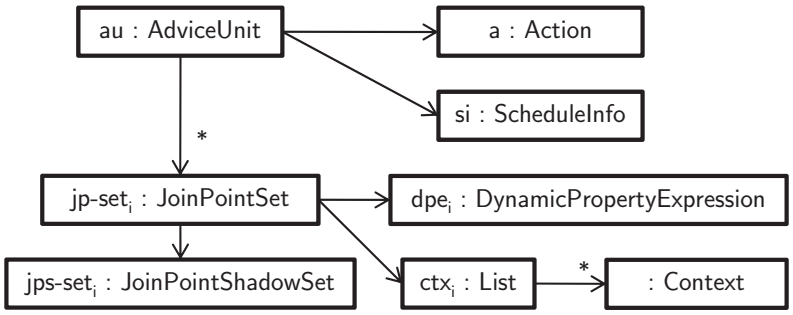


Figure 3.8: Object diagram of a general advice unit.

3.3.2 Work Flows

Dynamic Deployment. System allows clients to dynamically deploy and undeploy aspects provided as LIAM models. Deploying an aspect is divided into a series of deployment operations for advice units. Figure 3.8 shows a general data structure of an AdviceUnit *au*. It refers to an Action *a*, a ScheduleInfo *si* and several JoinPointSets $jp\text{-}set_i$ which in turn consist of one JoinPointShadowSet $jps\text{-}set_i$, a DynamicPropertyExpression dpe_i and a list of Contexts ctx_i. During deployment, any $jp\text{-}set_i$ is processed: from $jps\text{-}set_i$ a collection $jpshadows_i$ of JoinPointShadow objects is retrieved. A so-called BoundAction object is created to store directives about how to execute the advice action should it be applicable. A BoundAction consists of the Action *a*, the ScheduleInfo *si*, and the list of Contexts ctx_i. Together with the DynamicPropertyExpression dp_i, the BoundAction is attached to all JoinPointShadows in *jpshadows*. The collection of all pairs of DynamicPropertyExpressions and BoundActions that are attached to a JoinPointShadow constitute the shadow's residual dispatch logic. When multiple BoundActions are attached to a join point shadow, FIAL determines their order of execution using their associated ScheduleInfo objects. This is the responsibility of the Scheduler component.

All JoinPointShadow objects to which such a BoundAction object is attached during deployment are marked for code re-generation. For this purpose, System invokes an abstract method that has to be implemented by a FIAL instantiation and is responsible for re-generating code of the join point shadows. The instantiation must implement this method such that the next time the join point shadow is executed the new version of its code is executed. When an aspect is undeployed, FIAL determines the affected JoinPointShadows, removes the BoundActions originating from the aspect, and calls the instantiation to re-generate code for the join point shadows.

Dynamic Class Loading. The concrete execution environment has to notify System when new join point shadows become available by passing it JoinPointShadow objects that represent the newly available shadows. The System re-evaluates all existing JoinPointShadowSets to determine if they select any of the new join point shadows. Join point shadows that are added to a join point shadow set during class loading are further processed by FIAL. If the affected join point shadow set is referred to by a currently deployed AdviceUnit, a BoundAction is generated and added to the new join point shadow. A list of all loaded join point shadows is maintained in order to initialize new JoinPointShadowSets upon their creation.

Importing aspect definitions. So-called *importer* components can be provided to FIAL which are invoked just before an application starts to be executed. An importer component can generate LIAM models of aspects that are part of the executed program, e.g., by reading aspect definitions from a file. It is the responsibility of a FIAL instantiation to notify the framework when the application is about to start.

3.3.3 Weaving Directives

Join Point Shadows. FIAL makes all relevant information for generating the code of a join point shadow available through the interface of the JoinPointShadow class. While this class is abstract in the framework, the management of residual dispatch is already implemented. A framework instantiation is supposed to add instantiation specific data and functionality to the JoinPointShadow type.

Dispatch Functions. The framework manages two kinds of information about join point shadows. First, it provides a DispatchFunction that has to be evaluated at runtime in order to determine which actions to execute at a shadow and in which order. The dispatch function combines all DynamicPropertyExpressions that are attached to the join point shadow together with a BoundAction. The result of the dispatch function determines the BoundActions to be executed. This reflects the rationale of *virtual join points* as discussed in Section 2.3 where one dispatch function determines the appropriate implementation of the join point and the combination of advice actions reflects this implementation.

The FIAL instantiation receives an evaluation strategy for the dispatch function from FIAL. This strategy is provided as a binary decision diagram [Bry86] represented as a directed acyclic graph, as illustrated in Figure 3.9.

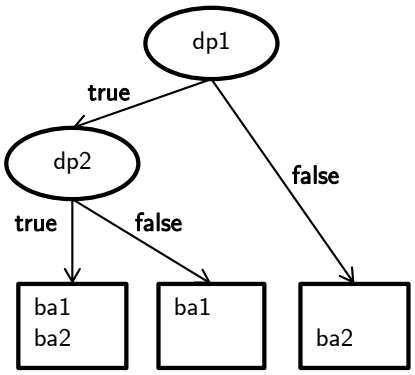


Figure 3.9: Evaluation strategy of a dispatch function as a BDD.

The inner vertices of the graph are DynamicProperty objects (dp1 and dp2 in Figure 3.9) which stem from the combined DynamicPropertyExpression objects of a join point shadow. Each vertex has two out-bound edges that specify which node to evaluate next if the DynamicProperty is satisfied, or not. Finally, the leafs (also called sinks) are annotated with a combination of BoundActions (ba1 and ba2 in Figure 3.9) to execute when the sink is reached during the evaluation of the dispatch function. In the example, dp1 is evaluated first; if it fails, only the bound action ba2 is executed. If dp1 succeeds, dp2 also has to be evaluated; if its evaluation succeeds, i.e., dp1 and dp2 are **true**, the bound actions ba1 and ba2 have to be executed. If dp2 fails, i.e., dp1 is **true** and dp2 is **false**, only ba1 gets executed.

Evaluating all DynamicProperty objects before evaluating any BoundAction is only possible under the assumptions that the evaluation of DynamicPropertys has no side effects and their result is not influenced by the execution of BoundActions. These assumptions are generally true in the AO languages investigated in this thesis [SBM08].

The approach to join point dispatch bears the opportunity of optimizations: DynamicPropertys need to be evaluated only once and the evaluation strategy can be optimized with respect to the average runtime cost of evaluation. A possible optimization is to evaluate those DynamicPropertys first that are most likely to fail. More sophisticated optimizations of the dispatch function are subject to future work and a preliminary discussion is given in Section 7.2.1.

Bound Actions. Directions for executing actions are provided in terms of BoundAction objects describing how to execute the actions. Figure 3.10 shows an example object diagram of a BoundAction (ba). The action to perform is to call a virtual method, hence, an aspect instance has to be retrieved to serve

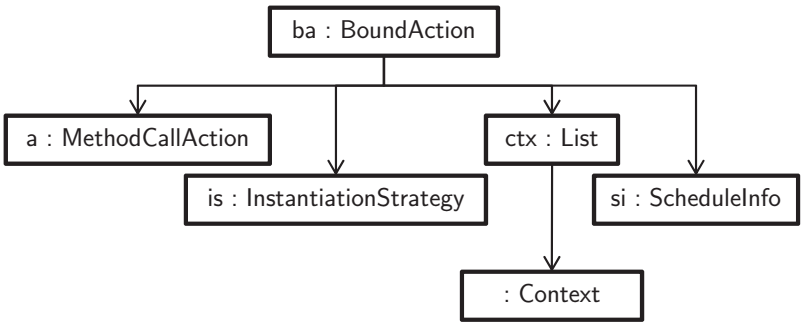


Figure 3.10: Object diagram of a BoundAction.

as the receiver of this call. The associated InstantiationStrategy (is) describes how to accomplish aspect instance retrieval. The arguments to be passed to the called method are represented by the list (ctx) of Contexts. Finally, the MethodCallAction declares which method is to be executed.

Action Order Elements. FIAL provides a facility to determine the order in which to generate code for the bound actions associated with a join point shadow. For this purpose, it passes the ScheduleInfo objects of the BoundActions of the join point shadow to the Scheduler component of FIAL, which returns the appropriate order encoded as a chain of ActionOrderElement objects. The structure of such a chain is defined by the class diagram in Figure 3.11.

Each ActionOrderElement object stores a list of *before* actions, followed by an *around* action, which, in turn, is followed by a list of *after* advice to be executed after the *around* advice has finished. The lists store the actions in the order of their execution. Any ActionOrderElement, aoe_i, may be linked to a following element, aoe_{i+1}, specifying the actions to execute, when the *around* advice of aoe_i **proceed**s. Once the execution of the last *after* advice of aoe_{i+1} is finished, the *around* advice of aoe_i continues after its **proceed**, followed by the *after* advice of aoe_i, if any.

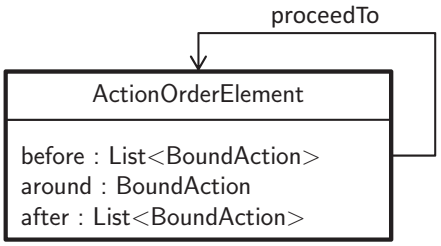


Figure 3.11: Data structure for representing the order of advice.

Figure 3.12 shows the order structure returned by a precedence-based scheduler for pointcut-and-advice in the aspects from Listing 3.3. In the figure, m refers to the original join point action; the other bound actions are labeled with the message printed by the corresponding advice.

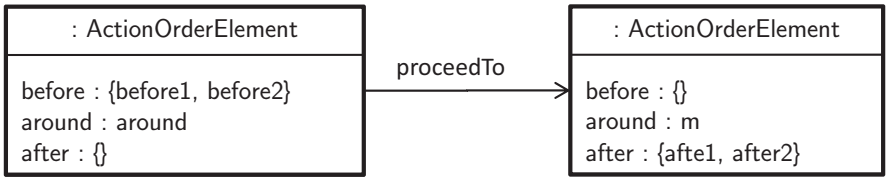


Figure 3.12: Object diagram of an ActionOrderElement data structure.

3.4 Language Mappings

It is claimed in this chapter that LIAM models are supposed to act as intermediate representations of aspects. Thus it would be required that a compiler generates and persists such models. This approach has been investigated by developing a compiler for the AspectJ language that generates LIAM models. It has been found that this approach requires re-implementing complex tasks of a compiler like type checking that are not related to handling aspect-oriented constructs have to be re-implemented. A different integration approach that allows to concentrate on pointcut-and-advice is presented in the following.

The observation has been made that many existing AO compilers—e.g., the `ajc`, CaesarJ and Compose* compilers—work in two phases. The first phase generates bytecode and the second phase performs the weaving. The first phase passes a full description of the aspects to the second one in order to specify how it should weave the aspects. Furthermore, it is possible to interrupt the compilation process after the first phase in the investigated compilers. The description is proprietary to each compiler, but it is possible to write specific translators for each such notation that generates the corresponding LIAM model.

To prove the appropriateness of LIAM to express aspects defined in up-to-date AO programming languages, mappings for four different widely-spread languages are discussed here; these are the languages AspectJ, Compose*, JAsCo and CaesarJ.

For the first two, the existing compiler could be re-used and stopped after the code generation. A component for the *import* call-back of FIAL has been

developed that translates the compiler specific description of aspects into LIAM models.

For JAsCo, a different approach has been chosen, i.e., a cross-compiler. The JAsCo compiler is not implemented in a way that it can easily be kept from weaving and a proprietary aspect definition cannot be retrieved. Thus, for an integration, the compiler has either to be modified or a new one has to be implemented. Regarding the pointcut-and-advice part of JAsCo, it is very similar to AspectJ. Thus, after having implemented the importer for AspectJ, the best choice was to implement a JAsCo-to-AspectJ cross-compiler. In contrast to a dedicated compiler, which maps the source language's high-level concepts directly to the lower-level constructs of LIAM, a cross-compiler merely maps these concepts to the corresponding concepts of the target language. And this target language is, when compared to LIAM, typically richer in high-level constructs, which makes mapping concepts much easier. Indeed, the mapping is particularly straight-forward if the source and target language have many high-level concepts in common, as is the case with the JAsCo and AspectJ languages.

For CaesarJ, so far no automatic translation has been implemented. However, it is reasonable to assume that it is possible. The difference of CaesarJ's pointcut-and-advice part and AspectJ is not significant. The only differing features are scoped and dynamic deployment of aspects. As is discussed in Section 3.3, dynamic deployment is supported by FIAL; deploying an aspect in a scope can be realized as follows: before the aspect is deployed, the LIAM model is copied and for all advice units in the copy, a dynamic property that checks if the current join point is in the specified scope is *and*ed to the dynamic property expression of all its join point sets. The remainder of the pointcut-and-advice features offered by CaesarJ is equivalent to those of AspectJ and is thus also covered in the following subsection. In fact, the CaesarJ compiler internally uses the AspectJ weaver. But an older version of the proprietary aspect description is used than that understood by the AspectJ importer for LIAM.

The concepts of the languages discussed in Section 3.1 are all supported by the concrete LIAM entities developed in this thesis. These entities often are more general and finer grained than, e.g., the keywords of the discussed languages, and thus more re-usable. As a consequence, language constructs often do not map to a single entity, but to compound entities.

In this thesis the RequiredTypeDynamicProperty has been implemented which requires a Context. This may be parameterized with an ArgContext object to model the constraint posed by **args**. Additionally, to expose an argument value to the advice, an ArgContext object must be attached to the respective JoinPointSet.

3.4.1 AspectJ Mapping

AspectJ offers a "load-time-weaving" mode where the compiler generates bytecode for object oriented modules like classes and methods, including those that represent aspect types and advice bodies. To the latter ones, the compiler adds runtime visible annotations that more or less contain the source code of the aspect-oriented constructs like pointcuts. In this mode weaving is performed at class loading time, when the program is to be executed. This is exactly what is suggested here as an integration approach. Thus, only an importer that reads the proprietary aspect definition format produced by the AspectJ compiler has to be implemented.

As already shown in Section 2.2, the `ajc` generates a class for each aspect and a method in this class for each advice in the aspect. When compiling with Java 5 compatibility, runtime visible annotations are added to these classes and methods that define the aspect-oriented parts of the aspect. Listing 3.9 picks-up the example from Section 2.2 and shows the generated annotations which are explained below. The compiler also generates the file META−INF/aop−ajc.xml that lists all classes that have been generated for aspects, as well as some weaver options that have originally been passed to the compiler.

```
1 @Aspect
2 class QueryInjectionChecker {
3   @Before("call(ResultSet Connection.execQuery(String)) && args(sql) && cflow
        (void FacadeServlet.doPost(Request, Response))")
4   static void before_0(String sql) {
5     if(isAttack(sql))
6       throw new SQLInjectionException();
7   }
8 }
```

Listing 3.9: Compilation and weaving result for class Services and aspect QueryInjectionChecker.

The importer for AspectJ, first searches the class path for such aop−ajc.xml files and loads every one it finds. The format of the file as well as the semantics of combining several descriptor files is specified in the AspectJ Development Environment Guide [Aspb]. The importer processes the files according to this semantics and determines the aspects, which have to be active during the execution. Afterward, the importer loads the classes, that are generated from the aspects, and processes their annotations.

Most interesting are the annotations of the advice methods, @Before, @Around, and @After, which determine when the method is to be called. These

annotations contain pointcuts that follow—with few exceptions—the syntax of pointcuts in the AspectJ source language [Asp08].

A parser has been implemented for the AspectJ pointcuts with the parser generator JJTree [JJT]. This generates parsers that build an abstract syntax tree (AST) of the parsed input. In order to actually create the LIAM entities, a visitor for the generated AST of the pointcuts has been implemented. While visiting the AST of a pointcut, it generates the LIAM entities corresponding to the pointcut. With the visitor, an algorithm is provided for transforming arbitrary pointcuts into LIAM entities. This algorithm shows that all pointcuts, which select a non-empty set of join points, can be expressed in LIAM.

In detail, the different parts of an aspect are translated as follows. In AspectJ, the instantiation strategy is defined aspect-wide. It is specified by the @Aspect annotation of the aspect class. The importer creates the InstantiationStrategy once per imported class and uses it for all AdviceUnits derived from the aspect class.

Aspect precedence is not handled by the AspectJ importer, but a possible implementation is outlined here. An aspect class may contain one or more methods with an annotation of the type @DeclarePrecedence that specifies the precedence order of aspects. Since a precedence-based Scheduler implementation is available for FIAL, this specification can be translated into appropriate ScheduleMethodData objects.

For each processed advice method, the importer creates an AdviceUnit. The InstantiationStrategy created at the beginning of the aspect class' processing is associated with the AdviceUnit as well as a MethodCallAction describing a call to the advice method and a collection of JoinPointSets derived from its annotated pointcut.

The visitor maps all pointcut designators which are statically evaluable to JoinPointShadowSets and the remaining designators to DynamicPropertys combined in a DynamicPropertyExpression. For those that bind context values, additionally, a Context object is created.

FIAL does not differentiate between the **call** and **execution** pointcut designator of AspectJ, this trade-off has already been discussed in Section 2.4. It is left to the concrete execution environment where code is woven, i.e., at the call site or in the called method. Thus, **within** and **withincode** pointcut designators are mapped to DynamicPropertys that select join points based on the topmost frame in the call stack. This is in contrast to the `ajc` and `abc` implementations which realize **within** and **withincode** as lexical properties of join point shadows [HH04, ACH+05b].

Since in AspectJ all aspects are active throughout the whole execution, the importer deploys all aspects that it has generated from the aspect descriptions. Thus, they are active from the start of executing the application.

3.4.2 JAsCo Mapping

The original JAsCo compiler is not implemented in a way that it can easily be interrupted after bytecode generation and before weaving, and it also has no easily accessible proprietary description of the pointcut-and-advice parts of aspects. To conduct a case study of mapping JAsCo to LIAM, a cross-compiler from JAsCo to AspectJ has been developed for reasons of simplicity instead of inventing a new format for describing JAsCo aspects.

Advanced concepts of JAsCo that go beyond those of AspectJ, and how they could be mapped to LIAM is discussed at the end of this sub-section. The cross-compiler has been developed by *Vrije Universiteit Brussel* and further components required for the integration with FIAL have been developed cooperatively [BSM+08].

The JAsCo compiler compiles **hook**s and **connector**s to Java classes. Additionally, *deployer code* is generated by the cross-compiler in the form of an AspectJ aspect. The aspect intercepts the join points selected by the **connector**'s pointcuts, sets-up the required reification of those join points in a form expected by JAsCo's **hook**, and finally triggers the **hook**'s advice method. Additionally some runtime library code has been developed to translate from the AspectJ reflective object **thisJoinPoint** to those of the JAsCo runtime. Finally, an aop−ajc.xml file which declares the additional deployer as an AspectJ aspect is generated; this enables the use of the AspectJ importer discussed in the previous sub-section.

As an example of how the cross-compiler works, consider Listing 3.10, a shorter version of the example from Section 3.1.3. Cross-compiling the Logger class generates two files, one for the Logger class and one representing the **hook** Logging as an inner class.

```
1  public class Logger {
2    hook Logging {
3      Logging (method(..args)) {
4        execution(method);
5      }
6      before() {
7        System.out.println("Executing " + thisJoinPoint.getSignature());
8      }
9    }
10 }
```

```
11
12 static connector LoggingDeployer {
13   Logger.Logging aspect1 = new Logger.Logging(* *.set*(*));
14 }
```

Listing 3.10: Example of logging implemented in the JAsCo language.

For the connector, the cross-compiler generates a *plain Java class* (in the example LoggingDeployer) and a *class with AspectJ 5 annotations* that has the same name but is placed into a distinct package, i.e., jasco.ajconversion. While this aspect is created directly in the form of Java bytecode, it is, for the sake of explanation, shown as Java source code in Listing 3.11.

The pointcut language of JAsCo is nearly identical to the one of AspectJ and, thus, pointcuts can be used in the AspectJ output with nearly no changes. Only the pattern language differs in two ways. Although the wildcard named "*" exists in both pattern languages, it has different meanings. In JAsCo "*" stands for any character also across sub-package names; in AspectJ the segments of package names are matched separately and the wildcard ".." stands for any package segments. E.g., the pattern com*Example would match the class name com.something.Example in JAsCo, while it would not in AspectJ. In the AspectJ pattern language only, e.g., comSomeExample would be matched. However, this can be handled by the cross-compiler: for a pattern of the form p1*p2 the AspectJ pattern p1*p2 || p1*..*p2 is created. The second difference is the "?" wildcard in JAsCo which matches any single character and has no AspectJ counterpart. Thus, the cross-compiler cannot handle pointcuts using this wildcard.

```
1  @Aspect
2  class jasco.ajconversion.LoggingDeployer {
3    static Log hook; // Gets initialized statically.
4
5    @Before(value="(args(..)) && (execution(* *.set*(..)))", argNames="
         thisJoinPoint")
6    public void hook_before(JoinPoint thisJoinPoint) {
7      // JAsCo-ify the join point.
8      MethodJoinpoint m = JPBuilder.buildFrom(thisJoinPoint);
9      // Call the actual advice code.
10     hook.before(m, thisJoinPoint.getTarget(), thisJoinPoint.getArgs());
11   }
12 }
```

Listing 3.11: Java class with AspectJ 5 annotations representing the instantiated **hook** from Listing 3.10.

This cross-compiled deployer has a field referring to an instance of the inner class corresponding to the JAsCo **hook** (line 3) and a method (lines 6–11) annotated with an AspectJ 5 pointcut annotation (line 5). The poincut is composed of the abstract pointcut defined in the **hook** and the pattern that is passed to the **hook**'s constructor; at intercepted join points, context required by the **hook**'s advice method is set-up and the advice method is called.

Two features of JAsCo are not supported by AspectJ but are supported by FIAL. Due to the cross-compiler approach, they are, however, emulated in the generated AspectJ code. First, the semantics of dynamic deployment can be mimicked by making the cross-compiled deployer test at run-time whether the hook it refers to is currently active or not. But by doing so, the deployment and undeployment of aspects is not explicit to FIAL and thus potential optimizations as discussed in Section 4.3 cannot be effective.

Second, JAsCo allows for a more fine-grained control of the order in which multiple **hook**s are executed: if a **connector** deploys multiple **hook**s, it is possible to define the precise order in which the **hook**s' advice are executed. This is not possible in AspectJ, which allows precedence declarations only at the level of aspects, but not at the level of advice. While in some cases it may be possible to split a single JAsCo **hook** into multiple AspectJ aspects, thereby mimicking JAsCo's fine-grained precedence strategies, this complicates cross-compilation a lot: a hook's private fields have to be shared by all aspects which were generated from the hook in question. Again, FIAL would be able to support JAsCo's fine-grained precedence model, and the limitation is due to the detour via AspectJ aspects.

Patterns in FIAL are not limited to AspectJ patterns. Although there is a default implementation for AspectJ-like patterns, other implementations can be provided. Thus, also JAsCo's extended pattern language can be mapped to FIAL.

3.4.3 Compose* Mapping

Compiler. The Compose* compiler consists of a pipeline of phases which gradually perform transformations of the source code eventually leading to weaving aspects into the base code. It exists for three different platforms: .NET, Java and C. To support reuse, the Compose* compiler is divided into a platform-independent part and three platform-dependent parts, one for each supported platform. The platform independent part contains most of the functionality of the compiler, e.g., compiling the composition filter specification, resolving references, and reasoning about the compositions' correctness. The platform-dependent parts implement the platform specific be-

havior, which includes extracting information about the base program, e.g., as type information, and performing weaving. Just executing the platform-independent part of the Compose* compiler has the effect that no weaving is performed.

However, the intermediate representation used to exchange weaving directives between the platform-independent and the platform-dependent part is not easily readable because it is already too low-level. An earlier compiler phase, however, generates an intermediate representation at a suitable level. Thus, a new compiler phase has been implemented that writes out this intermediate representation. This phase simply serializes the intermediate representation. The file is later read by the importer which de-serializes intermediate representation.

The heart of the Compose* compiler is the message flow analysis for reasoning about how a given message behaves in the presence of the specified composition filters, e.g., as shown in Figure 3.2. The output of this analysis, the so-called *execution model*, gives precise information about how a given message behaves in the filter set. Therefore, it can be used to translate the filter set to base code for a specific method. This translation is done by the compiler's *Inliner* module, which generates a so-called *abstract instruction model* for a given filter set and message. The abstract instruction model specifies in a platform-independent way the base code for a given filter set. This code represents combined conditions of several filters and also the order is determined in this step. After this step of the compiler pipeline, the high-level constructs of Compose* are broken down to smaller parts corresponding to the granularity of LIAM. Thus, the new compiler phase is inserted directly following this one. In the following, results of the *Inliner* module are presented which are the basis of the importer discussed thereafter.

Importer. The instructions for abstract input filters are provided by the compiler as pairs of MethodInfo and FilterCode objects. The MethodInfo object is a fully qualified descriptor of a method. When this method is called the filter composition is to be applied. For each such pair, the importer uses the MethodInfo to generate a JoinPointShadowSet—whose pattern exactly matches this method—which in turn is associated with all actions it derives from the FilterCode.

The FilterCode object represents the abstract instructions to be executed whenever a message passes through a set of filters. Abstract instructions are Blocks, Branches, Jumps, and FilterActions. Branches have ConditionExpressions attached, which determine under which conditions the **true** respectively **false** branch is taken during execution. When the abstract instruction model is to

be imported and transformed into AdviceUnits, however, the conditions under which FilterActions are executed need to be determined. This is done by propagating the ConditionExpressions the Branches are annotated with [dR07]. This propagation is made feasible by the fact that the control flow graph corresponding to any such abstract instruction model is a directed acyclic graph; thus, its instructions can be processed in topological order. After each Instruction has been annotated with the condition under which it gets executed, the ConditionExpressions used by the Compose* compiler are converted to DynamicPropertyExpressions as expected by FIAL.

This conversion is straight-forward but for one issue: the dynamic property expressions are expected to be in negation normal form whereas the former are not. Thus, negations must be pushed downward to the level of DynamicPropertys, i.e., the leafs in the expression tree. This is done by applying *De Morgan's* laws during the transformation.

Also, DynamicPropertyExpressions cannot explicitly represent the Boolean constants **true** and **false**, which is, however, possible in Compose*'s abstract instruction model. While a **null** DynamicPropertyExpression is implicitly understood as true, something similar is impossible for false—and unnecessary: if applying constant folding to a ConditionExpression yields the constant false, no AdviceUnit is generated at all.

The DynamicPropertys themselves have to be generated from the atomic Conditions used in the ConditionExpressions of Compose* during the transformation to DynamicPropertyExpressions. These are always provided in terms of a method to be called. Thus, a DynamicProperty has been implemented for the Compose* integration that calls the condition method and returns its result value.

For each FilterAction a DynamicPropertyExpression is derived by the importer. Context binding is only possible in terms of the *Meta* filter in Compose* which is currently not supported by the importer. Thus, together with the previously generated JoinPointShadowSet the DynamicPropertyExpression forms the JoinPointSet.

Next, the importer generates the appropriate Action entity and associate it with the JoinPointSet into an AdviceUnit. The Compose* compiler already reflects mutual exclusion of filter actions in the abstract instruction model as generated by the Compose* compiler. Thus, two actions of Compose*'s abstract instruction model, namely the *SkipAction* and the *ContinueAction*, do not occur in this model as they only control the exclusion of other actions; only those actions that really provide some functionality actually occur in the model the importer works with. The filter actions that can occur are *DispatchAction*, *AdviceAction*, and *ErrorAction*. A *MetaAction* could also occur at that level of the model, but it is not supported by the importer

because it may affect the filter composition reflectively at runtime whereas FIAL requires a composition fix at weaving time.

An error filter action is translated into an ErrorAction of LIAM. Dispatch and advice filter actions work very similar: both lead to a method that is being called and thus are realized as MethodCallAction in the LIAM model. The difference between both is that potentially multiple advice actions can be performed at a join point, but only one dispatch action. Usually, the dispatch action is the join point action, i.e., that action that has lead to the join point in the first place. However, in Compose* it is also possible that the original join point action is replaced by a different action, e.g., calling the same method on another object or calling a completely different method. This is why the join point action is explicitly contained in the abstract instruction model.

In FIAL, the join point action is also an explicit Action at a JoinPointShadow. For an advice unit derived from a dispatch action the Compose* importer generates a ScheduleInfo declaring that the join point action is to be skipped if the action is executed.

The abstract instruction model contains all actions to be performed at the method call, including the join point action, thus, also an advice unit with this action is generated by the importer. However, this action cannot be a normal method call, because this would produce a new join point with advice being applied. Thus, the special ReplaceWithJoinPointActionAction is used for filter actions that dispatch to the originally intercepted method, which is a wildcard and will be replaced with an action that actually performs the original join point action.

Finally, the importer ensures that the actions are performed in the correct order. The order is determined by the FilterCode data structure by the order of their appearance in the control flow graph. Since the importer independently generates an AdviceUnit for each action, this order gets lost. Thus, the order must be re-established by attaching the correct ScheduleInfo to the Actions. A priority-based scheduling strategy is chosen. As the FilterCode specifies all actions that share the same join point shadows, priorities can be provided separately when processing a single filter action. The FilterActions are brought into a topological order with respect to the abstract instruction model's control flow graph. The index within this order is then used as the derived Action's priority. Hereby, it must be noted that actions in the so-called calling flow should be executed first if they appear first in the topological order, while those actions in the returning flow that appear first should be executed last.

Discussion. The Compose* importer can handle the abstract instruction model provided by the standard Compose* compiler to a great extend. But there are also some limitations, which, however, are well localized. The current version of the importer can only handle **external** objects and the **inner** object as the target of a call. In Compose*, **internal** objects are shared among all filters in a filter module. Since the filters may intercept different messages, they have different join point shadows whereby each join point shadow is handled separately by the importer. The abstract instruction model currently provided by the additional Compose* compiler phase is not sufficient to identify which filters originate from the same filter module. Thus, it is not possible to ensure that filters from the same filter module use the same **internal** objects.

Condition methods in Compose* can either require no arguments or one argument that captures the join point's context, similar to **thisJoinPoint** in AspectJ. This feature has not yet been implemented, but can be realized by an appropriate Context entity used by the respective DynamicProperty entities. Furthermore, the importer only handles input filters; output filters are currently ignored, but could be realized similar to the above.

Finally, the *Meta* filter is not supported because it can change the filter composition reflectively, i.e., at runtime, but FIAL requires an ordering that is fixed at weaving time. It is conceivable, however not verified, that the dynamic composition capabilities of the Meta action can be mapped onto Actions and dedicated DynamicPropertys.

3.4.4 Discussion

The outlined importer-based integration approach for AO languages, often facilitates to re-use the original tooling. In the case of AspectJ, the original compiler can be used with just an additional compiler option. This enables, e.g., using the AJDT [AJD] to develop AspectJ applications and to execute them on a FIAL-based execution environment. To do so, the "Non-standard compiler option" −XterminateAfterCompilation has to be specified in the "AspectJ Compiler" page of the AJDT project's properties. On the same properties page, the "Outxml" option must be enabled which causes the compiler to write the aop−ajc.xml file. Finally, the compiler compliance level must be set to 5.0 or higher in the "Java Compiler" properties—only then the compiler generates the runtime visible annotations required by the importer.

However, with specifying the −XterminateAfterCompilation option not only the compiler is barred from weaving the aspects; it also does not evaluate pointcuts anymore. Thus, the compiler also does not determine the crosscutting structure of the aspects which would normally be used by the IDE to

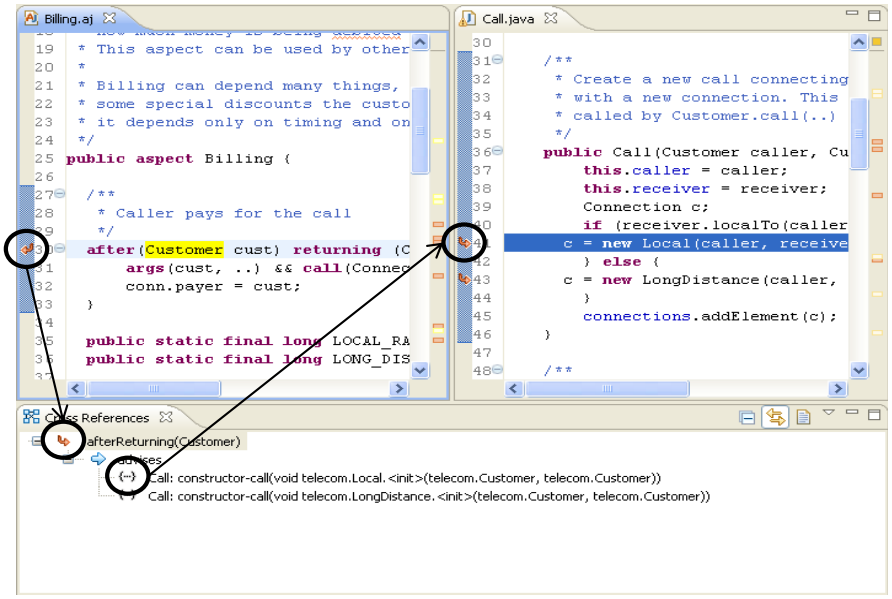


Figure 3.13: Linking of pointcut-and-advice and advised locations in AJDT.

show, e.g., in the "Cross References" view, or which would be used to facilitate navigation between advised join point shadows and their advice, as depicted in Figure 3.13. A solution to this limitation is discussed in Section 7.2.4

3.5 Execution Environments

To evaluate the appropriateness of FIAL to act as a framework for implementing execution environments for AO languages, different instantiations of FIAL applying different weaving techniques have been developed.

FIAL requires that each entity's semantics is implemented in a method of the entity's class, which facilitates a default code generation strategy for each kind of meta-entity from LIAM. This default code executes the entity's semantics by invoking the method. This way, the concept represented by an entity is implemented in isolation from other concepts.

Treating each concept separately, makes it easy to add new concepts and precisely define their semantics, namely by providing a programmatic specification that can be executed and tested. This strategy furthermore supports the generation of optimized code for those entities specifically supported by an execution environment as discussed further in Chapter 4.

The default implementation of the LIAM entities is based on implementing their meanings as pure Java methods. A Context entity, for example, has a

method which returns the corresponding context value and a DynamicProperty has a method that returns either **true** or **false**. Below, these methods are generally referred to as "perform", however, they are called differently for each kinds of entity, e.g., isSatisfied for DynamicPropertys or getValue for Contexts.

FIAL employs a factory for the creation of LIAM entities. Hence, the execution environments presented here can provide an appropriate implementation of the entities whereby the "perform" is implemented specially for those entities that have to interact with the execution environment.

For entities which require context values to perform their semantics, those values have to be passed as arguments to the "perform" method. Since the entities are objects and the above described methods are virtual, each LIAM entity has a numerical ID and FIAL provides a registry that facilitates to retrieve an entity by its ID. Thus, the code generated for executing an entity, is as follows:

1. Push the ID on the operand stack as a constant.
2. This constant is passed to FIAL's registry which returns the LIAM entity object.
3. For each Context required by this entity, generate code according to this enumeration.
4. Call the entity's "perform" method.

3.5.1 Envelope-Based Reference Implementation

The so-called *Envelope-based Reference ImplementatioN* (ERIN) has been implemented as an instantiation of FIAL. It supports all features of FIAL and performs, as the name suggests, envelope-based weaving. It is an advanced version of the prototype performing runtime weaving with envelopes, presented in Section 2.5.2, realized as a Java 6 agent.

A Java agent contains a class with a method named premain. This class is invoked by the virtual machine after it has finished its own initialization. When this method is invoked on the ERIN agent, it starts the initialization of FIAL and installs a class loading interceptor. The last action of the premain method is to *notify FIAL that the application is about to start.*

The agent uses the bytecode instrumentation package to intercept class loading: When a class is loaded by the virtual machine, the agent processes the class data, brings the bytecode in the envelope-style, generates JoinPointShadow objects and stores meta-information about the class to be

used for weaving. FIAL is *notified of these JoinPointShadows*. In ERIN the JoinPointShadow objects correspond to envelope methods.

When the *call-back for join point shadow code re-generation* is invoked by FIAL ERIN re-generates the code of the envelope methods related to the affected join point shadows. Subsequently, the JVM's Class Redefinition facility is used to replace the bytecode of those classes containing envelope methods that have been changed.

Most of the default LIAM entity implementations are independent from the concrete FIAL instance. However, the primitive context values like the receiver or the **this** object depend on the specific weaving strategy. As already discussed in Section 2.4, in envelope-based weaving the **this** object is not in the local context of the join point shadow. Thus, ERIN provides the Context implementations representing these primitive values which are implemented by means of the JVMTI as presented in Section 2.4.

Code is generated for envelopes according to the *weaving directives* provided by the corresponding JoinPointShadow object. ERIN, first, generates code that evaluates the dispatch function and stores the evaluation's result in a local variable. Second, code for each bound action is generated separately according to the default code generation strategy. In front of each code block for executing a bound action, a branch is placed which tests the result of the dispatch function and skips the bound action if it is not applicable. The dispatch function's result is encoded as a bit vector where the bit at index i determines whether the i^{th} bound action is to be executed (bit is 1) or not (bit is 0).

The applicable BoundActions and their execution order are provided for a JoinPointShadow in terms of a structure of ActionOrderElements. Figure 3.14 shows (a) an action order structure and (b) how it is mapped onto a sequence of bytecode instructions. The dashed arrows in sub figure (b) represent the control flow of the branch that test for the bound action's applicability in the dispatch function's result.

The solid arrows illustrate the control flow that has to happen when an *around* advice **proceed**s. In order to support that, the code for bound actions representing **around** actions is inlined into the envelope instead of generating code for invoking the advice method; i.e., the method's implementation is copied into the envelope. ERIN requires that the **proceed** special form is compiled into a method call which can be identified by naming conventions. Such instructions are replaced by a "jump to subroutine" instruction (**jsr**) that jumps to the first bound action from the next advice order element. When the last of its bound actions has been executed, the execution is continued just behind the **jsr** instruction representing **proceed**.

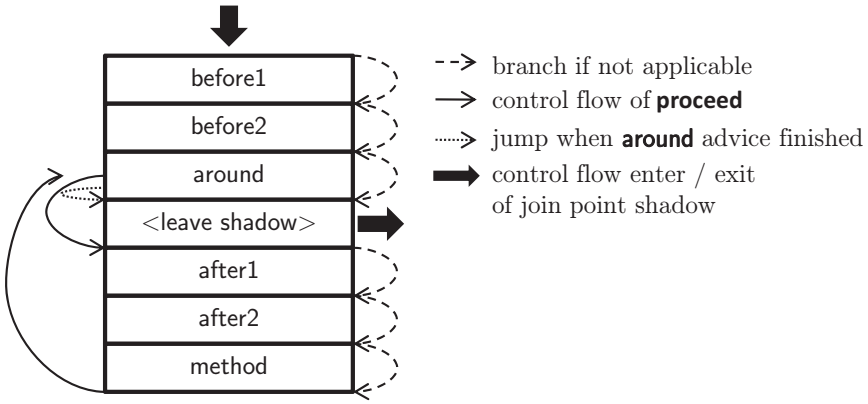


Figure 3.14: Instruction sequence generated for ordered BoundActions.

The inlined code is further modified in two ways. First, local variables are re-numbered to avoid interference with local variables of the envelope method or other inlined around advice. Second, **return** instructions are replaced by instructions that jump behind the inlined advice method's code to ensure that the execution continues with the code of the next bound action after the inlined advice method; this is shown as dotted arrows in Figure 3.14.

Inlining of **around** advice cannot be performed when the **proceed** occurs in an anonymous nested type in the **around** advice as observed in [ACH+05b, HH04]. They also describe an implementation based on closures that can be used in all situations. A similar solution is currently not implemented in ERIN.

3.5.2 Static Weaver

Another instantiation of FIAL has been developed in this thesis to show its appropriateness also in settings of static weaving. The weaving strategy pursued here is similar to that of the standard AspectJ compilers: join point shadows occur in the application code and are not relocated to envelopes. However, still the code generated for the different entities follows the default code generation strategy.

A main class beginning from which all referenced classes are determined recursively is passed to this static weaver. All determined classes are processed and *JoinPointShadow objects are generated which are passed to FIAL.* After all classes have been processed, *the importer call-back is triggered*, thus aspects may get defined and deployed. Thereby, FIAL attaches BoundActions to the JoinPointShadows. After the importer call-back has been executed, all affected join point shadows are modified. Then all classes that have been

loaded and processed in the first place are written out again. Those instructions that make-up join point shadows are not written unchanged, but code is generated for them according to their JoinPointShadow object, everything else is copied verbatim.

This execution environment does not support dynamic aspect deployment. Thus the *call-back for code re-generation* is to be implemented.

3.6 AO Language Design and Implementation with FIAL

The proposed architecture of language implementations and the developed framework, FIAL, that reifies this architecture facilitate a separation of the design and implementation of aspect-oriented programming languages. To demonstrate this, a case study of designing and implementing a simple example language is presented here.

Thereby, no dedicated syntax and keywords are designed nor is a compiler implemented. Following the importer-based integration approach of FIAL, Java annotations can be added to a program that can be translated to a LIAM model by an importer. Thus, *language design* encompasses defining the annotations, providing the importer, implementing additional concrete LIAM entities, and providing runtime support for the language. *Language implementation* integrates additional support for executing the new concrete LIAM entities, i.e., the constructs of the new language, into an execution environment.

The sample language is a domain specific aspect-oriented extension to Java for enforcing the decorator design pattern [GHJV95]. In this pattern, so-called "decorator" objects are associated to particular "decoratee" objects at runtime. Decorators and decoratees implement a shared component interface. All calls to operations on a decorator are forwarded to its referenced (decoratee) component, but the decorator has the possibility to execute additional behavior and to store additional data before or after the component's operation. A class diagram of the decorator pattern is shown in Figure 3.6. The structure is recursive in that the decoratee of a decorator can be either a basic component or another decorator.

A prominent example of the decorator pattern can be found in the java.io package. The code snippet in Listing 3.12 shows the usage of two readers, where the BufferedReader is a decorator for the StringReader. An issue with the decorator design pattern is that the decorated object remains an autonomous object on which operations can be directly invoked thus bypassing the decora-

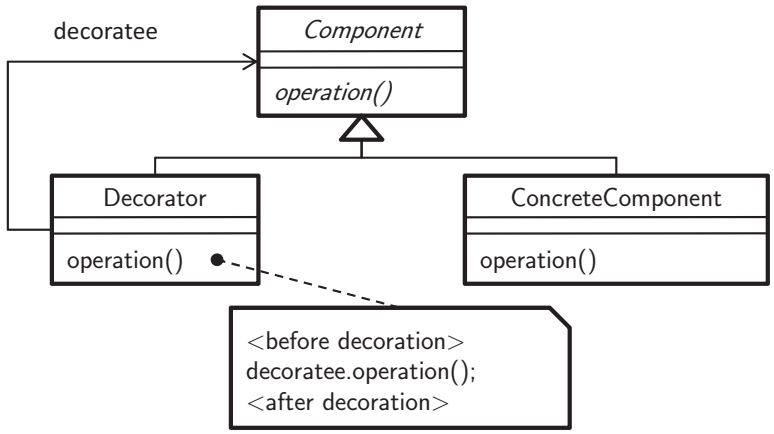


Figure 3.15: Class diagram of the decorator pattern's structure.

tors. For illustration consider line 4 in Listing 3.12 where the close operation is called on the decorated object directly. Thus, the decorator cannot perform clean-up itself, e.g., releasing the buffer. The correct behavior would be to call the close operation on the decorator object as shown in Listing 3.13, line 4.

```
1 Reader decoratee = new StringReader("something\nsomething else");
2 Reader decorator = new BufferedReader(decoratee);
3
4 decoratee.close(); // at this point, the "decorator" does not know
5                    // that the stream is actually closed. It, e.g.,
6                    // misses the opportunity to do some clean-up
```

Listing 3.12: Bypassing the decorator.

```
1 Reader decoratee = new StringReader("something\nsomething else");
2 Reader decorator = new BufferedReader(decoratee);
3
4 decorator.close(); // the "decorator" can perform clean-up and
5                    // will call the "decoratee"'s close method
```

Listing 3.13: Proper use of the decorator pattern.

The sample AO language discussed in this section enforces the correct use of decorators. The semantics of the language is based on the following assumptions.

- It is possible to specify a class whose objects may play the decorator role, called *decorator class*.
- It is possible to specify a class whose objects may play the decoratee role, called *decoratee class*.
- The decorator-decoratee-relationship is established when the constructor of the decorator object is called.
- The decorator-decoratee-relationship is between two objects where one plays the decorator role and one plays the decoratee role.
- There is a one-to-one decorator-decoratee relationship—otherwise enforcing to redirect operation calls on the decorator is not sensible (it would not be clear which decorator to call, or in what order).
- Only the decorator object is allowed to call operations directly on the decoratee object. Calls from other objects to the decoratee must be forwarded to the decorator.

In this section, it is discussed how to develop runtime support that realizes the language design according to the above assumptions. When a pair of a decorator class and a decoratee class is registered with the runtime support, an aspect is generated and deployed that enforce the behavior defined in the itemization above. Listing 3.14 shows an example usage of the runtime support. In line 1 the relationship between decorator class BufferedReader and the decoratee class StringReader is established. The runtime recognizes that the decorator-decoratee relationship is established in line 4 and automatically forwards the call of the close operation in line 6 from the decoratee object to the decorator object.

```
1 DecoratorRuntime.enforce(BufferedReader.class, StringReader.class);
2
3 Reader decoratee = new StringReader("something\nsomething else");
4 Reader decorator = new BufferedReader(decoratee);
5
6 decoratee.close(); // the language runtime support forwards this call
7                     // to the object decorator
```

Listing 3.14: Proper use of the decorator pattern in the sample language.

3.6.1 Realizing the Sample Language in FIAL

In this case study, the step of implementing an annotation as well as a FIAL importer is omitted. Instead, the application has to pass a pair of Class objects to the runtime support component developed which generates LIAM models and deploys the aspects. This component could as well be implemented as an importer reading annotations, but this case study is not about designing the syntax to be used in the source code. Instead, the goal is to investigate how easy it is to define and implement the semantics of a language construct.

To realize the language, the class DecoratorRuntime has been implemented to provide runtime support. It defines a static method enforce that requires the decorator class and the decoratee class as arguments. The runtime has three different responsibilities.

1. A facility is provided that allows to store and query the mapping of decorator and decoratee objects.

2. The establishment of a decorator-decoratee relationship for two objects is recognized and the objects are stored in the mapping.

3. Invocations of operations on an object for which a decorator object is stored in the mapping are redirected to its decorator object.

Decorator-Decoratee Relationship. The class DecoratorEnforcingPolicy is defined to store the decorator-decoratee relationship in a map. As shown in Figure 3.16 this class allows to add a decorator-decoratee dependency via the addAssociation method. With the methods hasDecorator and getDecorator it can be tested for a specified object if a decorator is registered respectively the decorator object can be retrieved. For each pair of classes one instance of the DecoratorEnforcingPolicy class is created by the runtime support.

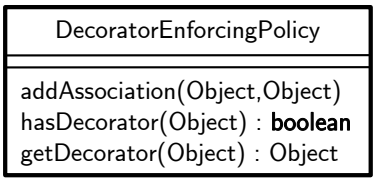


Figure 3.16: Class managing dependencies of decoratees and decorators.

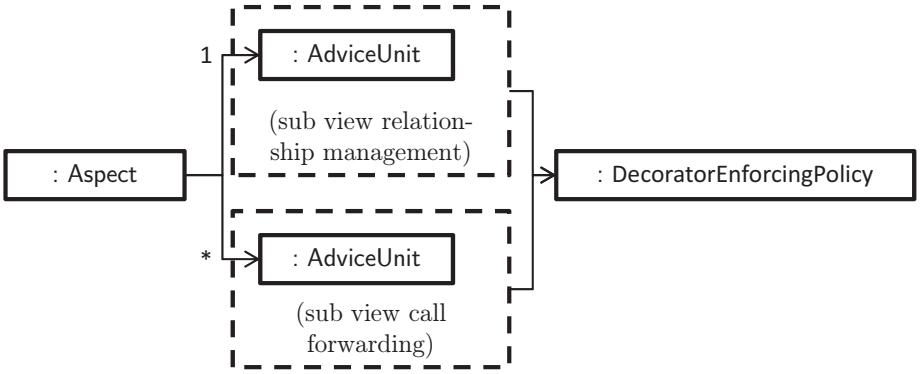


Figure 3.17: Structure of the aspect for enforcing the decorator pattern.

A LIAM aspect is constructed with advice units that perform the other two responsibilities as shown in Figure 3.17; one of them tracks the dependencies and the others forward calls from decorated objects to their decorators. Each aspect has an associated instance of DecoratorEnforcingPolicy to maintain and query the decorator-decoratee mapping.

Maintaining the Relationship. The advice unit for maintaining the mapping, shown in Figure 3.18, intercepts each call to a constructor of the decorator class that takes an instance of the decoratee class as an argument. Instances of the classes JoinPointShadowSet and a ConstructorPattern are created to select these constructor calls. In the figure, those parts of the aspect that are parameterized with the concrete classes, for which the decorator semantics is to be enforced are highlighted by a grey box. Thus, Decorator is a placeholder for the concrete class of the decorator; similarly, Decoratee is a placeholder for the decoratee class.

The JoinPointSet binds the target and the argument of the constructor call, which correspond to a decorator and decoratee object that are about to be associated. The action associated with the selected join points is a TrackDecoratorAction. This is implemented as a sub-class of MethodCallAction and specifies that the method addAssociation is called. This method takes both objects bound by the JoinPointSet as arguments and passes them to

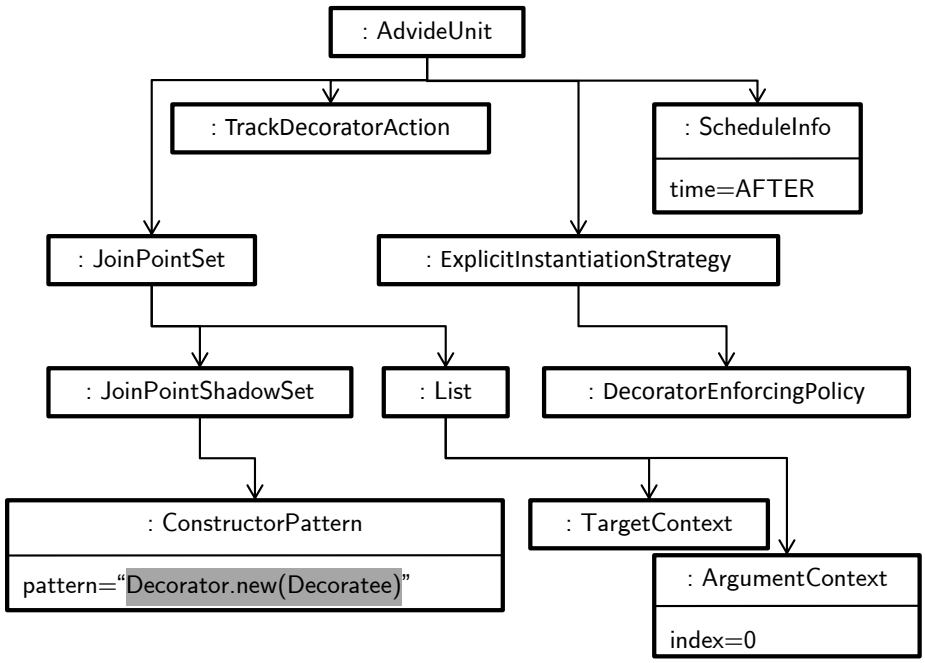


Figure 3.18: Advice unit for updating the decorator-decoratee mapping.

addAssociation which has to be called on the DecoratorEnforcingPolicy object assigned to the aspect. Therefore an instance of ExplicitInstantiationStrategy is used as instantiation strategy of this advice unit which always returns the DecoratorEnforcingPolicy object that holds the decorator-decoratee mapping for the currently processed pair of decorator class and decoratee class.

Forwarding Calls. To realize the forwarding of method calls on decorated objects, one advice unit is set-up for each method declared in the decoratee class. Figure 3.19 shows the common structure of these advice units. For each advice unit the JoinPointShadowSet selects all calls to the method, the concrete signature of which is inserted into the grey box in the MethodPattern. The action to be performed at selected join points is calling the method with the same name and signature on the associated decorator object, as shown by the MethodCallAction which also is parameterized with the concrete method.

The JoinPointSet binds all argument values to be passed to the advice action. That means, the arguments are not changed by forwarding the call. Only the receiver object is changed; the new receiver object is provided by the DecoratorInstantiationStrategy. This requires the receiver object on which the method is originally called, hence the use of the associated TargetContext. The code of the DecoratorInstantiationStrategy is shown in Listing 3.15; the

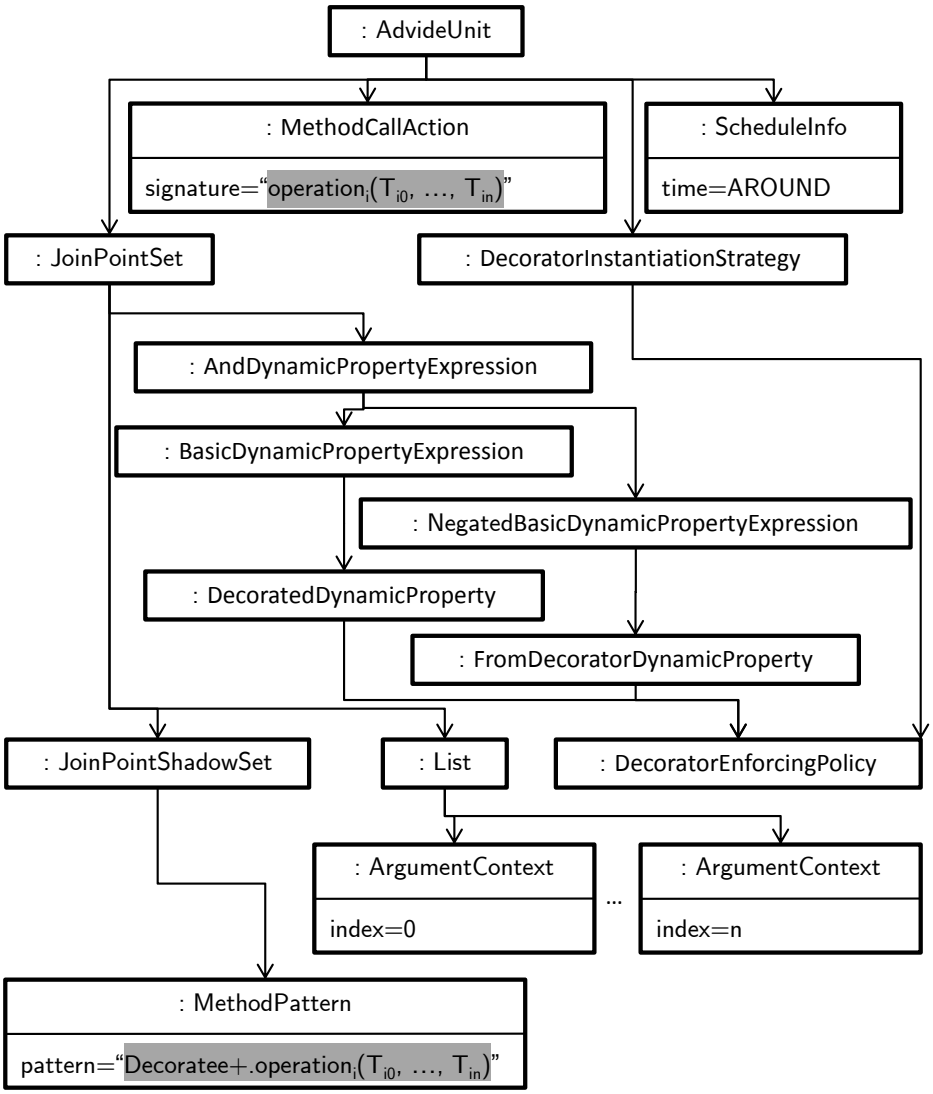


Figure 3.19: Advice unit for forwarding calls on decorated objects.

constructor calls the super constructor of InstantiationStrategy in line 6 passing a TargetContext object to signal the aforementioned dependency. The aspect's DecoratorEnforcingPolicy object is stored in a field of the instantiation strategy. To determine the advice instance at runtime, the method getAdviceInstance (line 10) is called with the receiver object as argument. The policy object is used to look-up the decorator of the receiver object in line 11.

```
1  public class DecoratorInstantiationStrategy extends InstantiationStrategy {
2
3    private DecoratorEnforcingPolicy policy;
4
5    public DecoratorInstantiationStrategy(DecoratorEnforcingPolicy policy) {
6      super(Collections.singletonList(Factory.createTargetContext()));
7      this.policy = policy;
8    }
9
10   public Object getAdviceInstance(Object decoratee) {
11     return policy.getDecorator(decoratee);
12   }
13 }
```

Listing 3.15: Implementation of DecoratorInstantiationStrategy.

A method call must be forwarded only under two conditions that are expressed in the DynamicPropertyExpression in Figure 3.19. The first condition, which is implemented by DecoratedDynamicProperty (Listing 3.15), states that the receiver object at the join point must be associated with a decorator object.

```
1  public class DecoratedDynamicProperty extends DynamicProperty {
2
3    private DecoratorEnforcingPolicy policy;
4
5    DecoratedDynamicProperty(DecoratorEnforcingPolicy policy) {
6      super(Collections.singletonList(Factory.createTargetContext()));
7      this.policy = policy;
8    }
9
10   public boolean isSatisfied(Object decoratee) {
11     return controller.hasDecorator(decoratee);
12   }
13 }
```

Listing 3.16: Implementation if DecoratedDynamicProperty.

The class FromDecoratorDynamicProperty in Listing 3.17 implements the other condition. The dynamic property tests whether the caller at the join point is the decorator associated with the receiver object. In difference to the implementation of DecoratorInstantiationStrategy and DecoratedDynamicProperty, this dynamic property test also binds the caller object (line 7). To determine if the dynamic property is satisfied, in line 13, the decorator for the receiver object is looked-up; if this decorator is identical to the caller object, the dynamic property is satisfied.

In the full dynamic property expression in Figure 3.19, the second dynamic property is negated and *and*ed with the first one; thus, the expression is satisfied, if a decorator is associated with the receiver and the caller is not the decorator.

```
1  public class FromDecoratorDynamicProperty extends DynamicProperty {
2
3    private DecoratorEnforcingPolicy policy;
4
5    FromDecoratorDynamicProperty(DecoratorEnforcingPolicy policy) {
6      super(Arrays.asList(
7          AspectModelFactory.createThisContext(),
8          AspectModelFactory.createTargetContext()));
9      this.policy = policy;
10   }
11
12   public boolean isSatisfied(Object caller, Object callee) {
13     return caller == policy.getDecorator(callee);
14   }
15 }
```

Listing 3.17: Implementation if FromDecoratorDynamicProperty.

The advice units in Figure 3.18 and Figure 3.19 are enclosed in one Aspect (see Figure 3.17) which is passed to the deploy method of FIAL. After being deployed, the aspect takes effect and the decorator pattern is enforced for the instances of the decorator class and decoratee class which has been provided to the runtime support for this sample language.

Optimizing Implementation. The code snippets presented in Listings 3.15–3.17 show that implementing the semantics of this language extension requires very little code and can be realized using high-level constructs like maps. Programs using this language extension can be executed on any execution environment that instantiates the FIAL framework. The code that

sets-up the aspect as discussed above is not shown here. But with approximately 150 lines of code it is also rather simple.

A language implementer can provide dedicated support for the new language concepts and can use the LIAM entity implementations provided by the designer as an *oracle*: the optimized implementation can be tested for correctness by executing a program that uses the new language abstractions on an execution environment without optimizations and on the execution environment with optimizations. The program must behave identically in both cases.

For the example language there are some optimization possibilities. The mapping maintained in DecoratorEnforcingPolicy tells the implementer that each object has at most one associated decorator object. Thus, a possible optimization is to enhance the execution environment's memory layout for objects instead of using a central map to maintain the decorator-decoratee relationship. One additional slot in each object is reserved to store the reference to the decorator object. Such an execution environment would handle TrackDecoratorAction, DecoratedDynamicProperty, FromDecoratorDynamicProperty and DecoratorInstantiationStrategy in a special way. Instead of compiling code that invokes the addAssociation method, for a TrackDecoratorAction the execution environment generates code that stores the decorator object in the decoratee object's new slot. Similarly, for the other entities code is generated by the execution environment that performs the look-up via the new slot.

A language designed and implemented in the proposed way also benefits from optimizations that are already applied in FIAL-based execution environments for common LIAM entities. Examples for such entities, used in the design of the sample language are ExplicitInstantiationStrategy, TargetContext or ArgumentContext.

This discussion shows that neither optimization issues have to be considered by the language designer nor does the language implementer have to care about how the new concepts are actually used in the designed language.

Chapter 4

Optimizing AO Concepts in the Virtual Machine

In this chapter, techniques are described that enable a virtual machine to perform so-called speculative optimizations of dynamic aspect-oriented constructs. This means that their execution is optimized whereby optimistic assumptions are made about the context in which they will be executed; measures are taken to undo the optimizations if the assumptions break at runtime. Speculative optimizations are an established technique, e.g., to improve the performance of virtual method calls.

Furthermore, it is shown how an explicit representation of AO concepts in the intermediate representation—and thus also in the VM's internal representation of the program—can be exploited by a virtual machine to perform advanced optimizations. This is shown by the example of an optimizing implementation of the **cflow** *dynamic property of join points.*

Parts of this chapter have been published in the following papers.

1. *Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. In* Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006*

2. *Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient Control Flow Quantification. In* Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006*

4.1 Motivation for Dynamic Optimizations of AO Concepts

Many modern programming languages provide virtual machines (VMs) that execute programs written in the respective languages and *manage* the execution. In order to execute a program, a virtual machine compiles the program's intermediate representation (IR) into native machine code (MC) instructions with a so-called *just-in-time* (JIT) compiler.

In simple terms, at JIT compilation each IR instruction is translated to a sequence of MC instructions. Then, optimization means to choose from different alternative compilation schemes for an IR instruction. Some alternatives are not correct in all situations, thus an optimizing JIT compiler analyzes the IR code to determine which ones are applicable. Additionally, collected profile data can be used during JIT compilation to guide optimizations, taking into account the actual characteristics of the executed application [AHR02].

A simple example of dynamic optimization is the compilation of IR instructions that can trigger class loading, e.g., the invocation of a static method. The semantics of Java stipulate that a class is loaded at the latest possible time, only just before it is actually used. Thus, the general compilation scheme is to generate code that queries the VM whether the class is already loaded; if it is not, the VM is requested to load the class. Afterward, the method itself is invoked. But since compilation happens at runtime in the virtual machine, it may be that the class is already loaded at JIT compilation time. In this case, the JIT compiler omits the machine code for potential class loading and only generates a call to the method.

The optimizations as well as the management provided by the VM are based on the fact that the intermediate representation is rather abstract and preserves most of the high-level concepts of the programming language; the IR instructions describe *what* should happen and not *how* it should happen leaving the decision about how to best realize the concept to the virtual machine.

In contrast to object-oriented concepts, aspect-oriented concepts are implicit in the intermediate representation used as input by the VM which hinders the VM to apply optimizations.

In this chapter two aspect-oriented concepts, namely dynamic aspect-deployment and the **cflow** dynamic property of join point, are investigated. They are made explicit to the virtual machine thereby facilitating optimizations. The prevalent implementation of these concepts is to generate IR instructions that cover the most general case or to emulate the concept by the means of standard VM services like the debugging interface.

One might argue that adding AOP-specific optimizations to a Java virtual machine (JVM) also has disadvantages, such as increasing the complexity of the JVM, and the dependencies between the JVM and AOP constructs. However, aspect-oriented programming is a programming paradigm, just like object-oriented programming. As such, it is natural to design and implement VMs dedicated to aspect-oriented languages just as dedicated VMs for object-oriented languages exist.

The performance of aspect-oriented programs is likely to be a key factor in whether AOP is adopted mainstream and treated as a language feature. Applying targeted optimizations in the virtual machine is an effective way to achieve good performance; one goal of this research is to advocate and evaluate this approach by investigating its performance potential. Furthermore, providing the optimizations as an instantiation of FIAL still offers the choice between a specific platform with optimized AOP support and less invasive but platform independent implementations like ERIN.

4.2 Dynamic Optimizations of OO Concepts

Since aspect-oriented languages are usually realized as extensions of existing languages, it is also natural to extend a virtual machine of the base language with support for aspect-oriented concepts. The most relevant aspect-oriented languages, including the ones discussed in the previous chapter are based on the Java language. Thus, a Java virtual machine has been chosen to be extended in this thesis. There are several open-source virtual machines for Java. A survey [MR07] that investigates four actively developed open-source Java virtual machines has shown that the Jikes Research Virtual Machine (RVM) [AAB+05, Jik, AAC+99] is the best choice when extensions to a production-strength virtual machine are to be developed. Among the surveyed virtual machines are HotSpot [PVC01], the reference implementation by Sun, and the default virtual machine of the Apache Harmony project (called DLRVM) [DRL08].

Jikes is itself implemented in Java [AAC+99] and it is designed to be extensible and to facilitate research [AAB+05]. It employs state-of-the-art virtual machine techniques [ACL+99, AFG+05] and thus offers a performance comparable to that of production virtual machines. Like other modern virtual machines [PVC01, SYK+01, ATBC+03, ITK+03] the Jikes RVM implements an advanced optimizing *just-in-time* (JIT) compiler, as well as an *adaptive optimization system*. For efficient start-up, methods are initially compiled with the quick so-called *baseline* compiler. As the program executes, the application is profiled to identify *hot methods*, i.e., those methods

that contribute the most to the program's execution time. These methods are re-compiled with an *optimizing* compiler. Optimizing compilation is only applied to hot methods because high optimization levels consume more compile time and usually generate larger machine code than non-optimizing compilation. If a method continues to be identified as hot, it may be re-compiled multiple times at higher optimization levels, until the highest level is reached.

Both, the baseline and the optimizing compiler, inject so-called *yield points* into the compiled code which are guaranteed to be executed regularly. To achieve this, yield points are placed inside each loop, at the beginning, and at the end of methods. When a yield point is executed, a call-back method of the Jikes RVM is called enabling it to perform thread switching. Furthermore, Jikes collects the profile data of the executing application at yield points [AHR02], by taking a sample of which methods are currently executing.

The baseline compiler performs no optimization whatsoever. It translates the bytecode of a method into machine code in one single pass; the generated machine code still uses an operand stack, like the Java bytecode, which it emulates in memory. Each bytecode instruction is handled separately, leading to poor machine code but quick compilation.

The optimizing compiler translates the stack-based Java bytecode into a register-based high level intermediate representation. The latter is transformed in several passes which each apply optimizations. Finally, the intermediate representation is transformed to machine code.

One of the most important optimizations is *method inlining*. The general compilation scheme for a method invocation is to look-up the correct implementation via a look-up table and set up a new frame in which the method is executed on the call stack. A frame is a region in memory that contains local values of the executing method. It is also used to pass argument values to the method respectively retrieve the result value from the called method.

If the JIT compiler can already resolve which method is going to be called it can decide to *inline* the target method. That means that the method invocation is replaced with the body of the target method. Inlining both avoids the look-up and the establishment of the stack frame; furthermore, it enables subsequent optimizations [Mel99].

For illustration, consider the class Scaling in Figure 4.1 and the code in Listing 4.1. Scaling defines the method scale which can scale an integer value to a different range, e.g., a measured value can be scaled down to draw it on the screen. In the example, however, the method just returns the passed argument, thus no scaling actually happens. Listing 4.1 shows a method drawGrid that draws a point on a canvas representing the maximum value that

can be achieved in a measurement. By the example of the call to Scaling.scale in line 7 different issues of method inlining are discussed in the following.

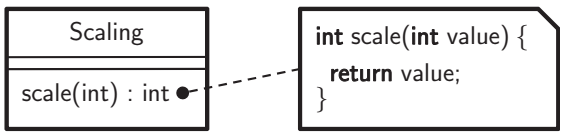


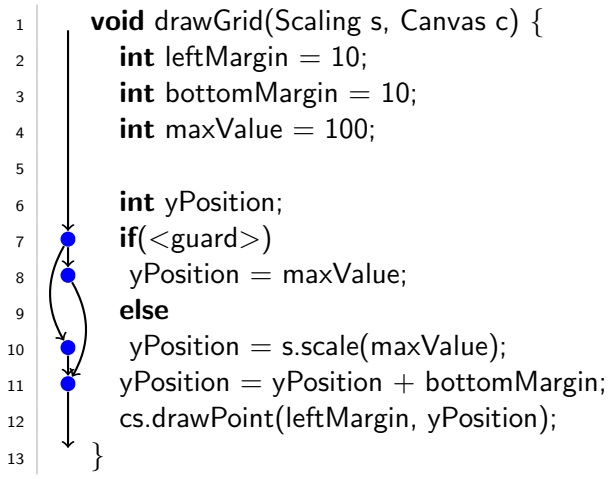
Figure 4.1: Class with a monomorphic method.

```
1  void drawGrid(Scaling scaling, Canvas canvas) {
2    int leftMargin = 10;
3    int bottomMargin = 10;
4    int maxValue = 100;
5
6    int yPosition;
7    yPosition = scaling.scale(maxValue);
8    yPosition = yPosition + bottomMargin;
9    canvas.drawPoint(leftMargin, yPosition);
10 }
```

Listing 4.1: Code calling a virtual method.

Above it is mentioned that inlining facilitates subsequent optimizations. Consider the case that the method scale in the example is **final** and, hence, the JIT compiler can resolve that the implementation in the class Scaling is always the target of this method call and it can be inlined. Because both methods, drawGrid and scale, share the same local context after inlining, the argument and result values do not have to be passed via the call stack frame. Even more important, the JIT compiler now can jointly optimize the methods drawGrid and scale. Since scale just returns its argument, the assignment in line 7 becomes yPosition = maxValue after inlining. Because maxValue and bottomMargin are constants in the example, the compiler can further optimize the lines 7–8; the value of yPosition can be determined by the compiler to be 110. Thus, also the arithmetic operation + does not have to be performed at runtime.

It is not always the case that a method is declared **final** even if no class is actually loaded at runtime overwriting the method. To benefit from the advantage of inlining, the JIT compilers of most JVMs perform *speculative method inlining* [DA99, SYN02, AHR02] if they detect at compile time that no class overwriting the method is currently loaded (the method is said to be *currently monomorphic*). An example of a monomorphic method is shown in



Listing 4.2: Guarded inlined method.

the class diagram in Figure 4.1. Speculative inlining means that the code of the target method replaces the invocation, but measures are taken to ensure that the inlined code is only executed when the method is still monomorphic at the time of execution.

The simplest form of speculative inlining is *guarded* inlining. Listing 4.2 shows pseudo code of the example with guarded inlining applied to the invocation of Scaling.scale. Line 8 represents the inlined code of the method which is prepended with a condition referred to as <guard> in line 7. This condition can be a *class test* that tests if the receiver object has the type Scaling. In this case it is correct to execute the inlined version of the method, thus, saving look-up and the establishing of the call frame. Otherwise, the inlined code is skipped and a full dispatch is executed as shown in line 10. For example, after the class Shrinking which extends Scaling is loaded as illustrated in Figure 4.2 the scaling argument passed to the drawGrid method can also be of type Shrinking.

Guarded inlining has two main drawbacks. First, the guard condition has to be evaluated at runtime whenever the method is invoked. Second, the presence of the full dispatch in the so-called *guarded region* (lines 7–10 in Listing 4.2) hinders subsequent optimizations based on data flow information derived from analyzing the inlined code. Simplifying the value of yPosition to the constant 110 as discussed above is the result of a data flow analysis.

To comprehend the second drawback, consider Listing 4.2: in line 11, the JIT compiler must assume that the value of yPosition is either set in line 7 or

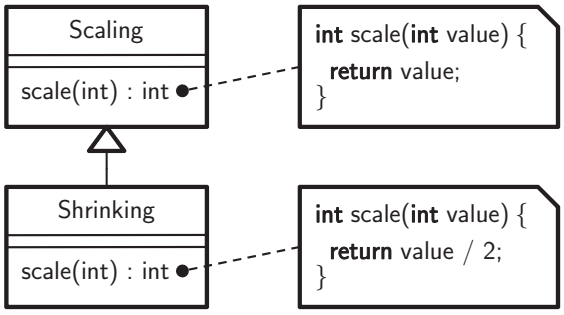


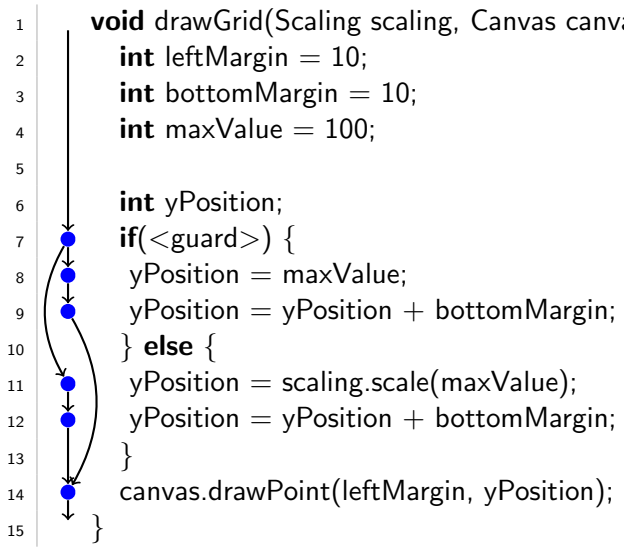
Figure 4.2: Class with a polymorphic method.

in line 10. In the latter case, its value is not known to the JIT compiler and it cannot simplify the expression in line 11. The control flow graph to the left of Listing 4.2 shows all possible paths through the method emphasizing that line 11 is reachable via two different paths.

To enable more effective optimizations in the presence of guarded code, techniques such as *code splitting* [SYN02, AR02] can be used to prevent the data flow information from the full dispatch path from merging back into the main control flow. While this approach increases the method size and, thus, compile time, it increases the quality of the compiled code. Listing 4.3 shows the result of code splitting. Code following the inlined method invocation is duplicated and attached to both the inlined method body (line 9) and the full dispatch (line 12). There are still two different possible paths through the method as shown by the control flow graph to the left in Listing 4.3, but lines 9 and 12 are only reachable on one path. Thus, the JIT compiler can further optimize the version following the inlined code and can evaluate yPosition to the constant 110 in line 9.

More advanced variants of speculative inlining are *code patching* and *on-stack-replacement*. In both approaches, the JIT compiler generates machine code only correct as long as the assumption that the called method is monomorphic is still true. When the assumption breaks, the compiled code which is optimized based on the assumption has to be invalidated; the assumption breaks when a class is loaded that overwrites a method which is inlined somewhere. There are three main steps when performing these speculative optimizations.

1. *Dependency tracking.* When inlining methods, the VM performs bookkeeping to identify the methods which need to be invalidated when classes are loaded. For example, if class loading causes calls to method m to become polymorphic, any compiled method into which the method m is inlined needs to be invalidated.



```
1   void drawGrid(Scaling scaling, Canvas canvas) {
2     int leftMargin = 10;
3     int bottomMargin = 10;
4     int maxValue = 100;
5
6     int yPosition;
7     if(<guard>) {
8      yPosition = maxValue;
9      yPosition = yPosition + bottomMargin;
10    } else {
11     yPosition = scaling.scale(maxValue);
12     yPosition = yPosition + bottomMargin;
13    }
14    canvas.drawPoint(leftMargin, yPosition);
15  }
```

Listing 4.3: Guarded inlined method with code splitting.

2. *Re-compilation of invalidated methods.* When class loading occurs, the VM invalidates all compiled methods identified by the dependency tracking from step 1 above. Next time an invalidated method is invoked, it is compiled anew whereby optimizations are applied based on the new class hierarchy. This ensures that future invocations of these methods will execute correctly.

3. *Invalidation of methods that are currently active on the stack.* Although re-compilation (from step 2 above) ensures that *future* invocations will execute correct code, the VM must also invalidate methods that are *currently* active on the stack. For example, consider a method main that is invoked only once and loops indefinitely. If inlining was performed in main, and that inlining needs to be invalidated, the optimized code for main needs to be updated *while its frame is on the stack*, to ensure that the remainder of execution performs a full dispatch.

Steps 1 and 2 are accomplished fairly easily in a VM. Dependency tracking to map inlined call sites to compiled methods is easily maintained using a lookup table. Re-compilation also is easy to perform because VMs already have advanced re-compilation support for their adaptive optimization infrastructure. The main challenge is step 3, invalidating methods that are active

on the stack, which is typically tackled by either code patching or on-stack-replacement (OSR).

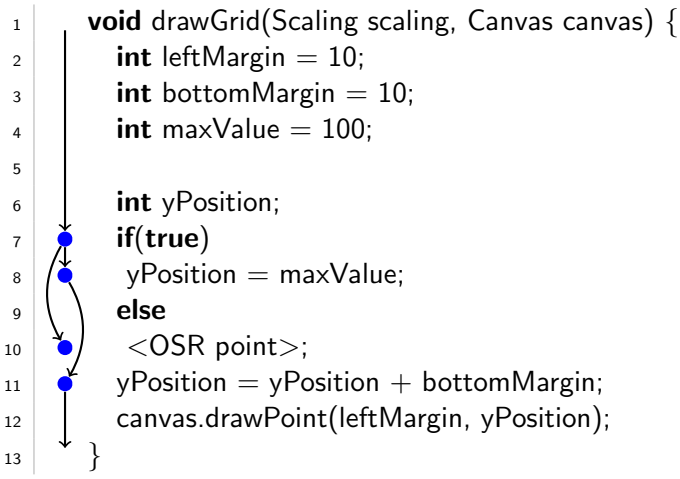
Code patching [SYN02] can be used to gain the functionality of guarded inlining, but without the runtime overhead of executing the conditional test. In a code patching approach, the guard is assumed to be true by default, and is compiled into a no-op instruction. For illustration, consider that <guard> in Listing 4.2 is replaced with **true**. The inlined code executes unconditionally and there is no overhead of executing a conditional test. However, if the assumptions change and the inlined method body is invalidated, the no-op instructions that guard the inlined code are dynamically replaced, i.e., *patched* with a jump to a full dispatch. The guard in the code patching approach is also called *patch point*.

Code patching is effective at removing the runtime overhead of the guard itself; however, the existence of a path with a full dispatch still interferes with optimizations that are enabled by a joint data flow analysis of the caller and the inlined callee methods. Even though the guard starts out as a no-op, it represents a potential "if-then-else" structure in the method's control flow (where the "else" clause is the full method dispatch). Instructions that follow the guarded code must be optimized assuming that either path may have been taken, thus data flow information from the inlined code cannot be propagated outside of the guarded region. Code splitting can be used to improve the code quality when code patching is used as a speculative inlining approach.

On-stack replacement (OSR) [HCU92, FQ03] allows exchanging the compiled version of a method while it is active on the call stack. This is in contrast to code patching which only modifies the compiled version of a method to invalidate the inlining of methods. OSR can be used for several purposes, including de-optimizing code for debugging [HCU92], undoing speculative optimizations in the presence of class loading [HCU92, FQ03], and promoting long running methods to higher levels of optimization [FQ03].

When using OSR as an invalidation mechanism for speculative inlining no code is generated for the full dispatch. A so-called on-stack-replacement point (OSR point) is placed in the "else" case of the guarded region as shown in line 10 in Listing 4.4. The guard is realized by using the code patching technique, thus no overhead is imposed by evaluating the guard's condition. When an OSR point is executed, the method containing the OSR point is recompiled and the new compiled version replaces the old one.

The key of OSR is that if execution jumps to an OSR point because the inlined code has been invalidated, it never merges back into the body of the compiled code. This is illustrated by the control flow graph to the left of Listing 4.4; note that there is no outgoing path from line 10. Therefore, the



```
1   void drawGrid(Scaling scaling, Canvas canvas) {
2     int leftMargin = 10;
3     int bottomMargin = 10;
4     int maxValue = 100;
5
6     int yPosition;
7     if(true)
8      yPosition = maxValue;
9     else
10     <OSR point>;
11    yPosition = yPosition + bottomMargin;
12    canvas.drawPoint(leftMargin, yPosition);
13  }
```

Listing 4.4: Speculative inlining with on-stack-replacement.

data flow information along this path does not merge back in and does not hinder optimizations based on forward data flow. No code splitting is required to achieve a high quality of generated code and compilation is simplified. The disadvantage is that the infrastructure required to facilitate on-stack-replacement is fairly complex, and not implemented by all virtual machines; the Jikes RVM, however, does implement OSR.

It is also possible to perform *polymorphic inlining*, i.e., inlining of methods which are known to be polymorphic. In this case, all possible target implementations are determined. The JIT compiler performs guarded inlining using a class test for those implementations for which it is expected to be beneficial. In contrast to monomorphic inlining the guarded region of polymorphic inlining resembles a "switch-case-default" structure where the "default" part also is the full dispatch.

4.3 Optimized Dynamic Aspect Deployment

Speaking in terms of virtual join points, deploying an aspect means that the dispatch function of join point shadows is modified. As discussed in Section 2.4.1, envelope methods are the manifestation of join point shadows. They contain code that represents the dispatch function and the potentially applicable actions. When aspects are deployed or undeployed, i.e., when the

dispatch function of a join point shadow is changed and actions are added to or removed from it, the envelope method's code has to be exchanged.

The challenges for an execution environment with support for dynamic aspect deployment are to *ensure that dynamic deployment is performed correctly and efficiently* while the *performance of other operations is not degraded*. Section 2.5.2 shows that envelope methods offer a conceptually clean abstraction of join point shadows and thus simplify dynamic weaving. As shown in that section, envelope-based weaving itself already addresses the requirement of fast aspect deployment. But it is also shown that the presence of envelopes in the application code reduces the application's performance.

In this section, techniques are presented that address the runtime performance challenge posed by envelopes and even further improve the efficiency of aspect deployment. The techniques are seamless extensions of established optimization techniques available in most modern VMs. They have been prototypically implemented in this thesis by making the Jikes RVM aware of envelope methods.

The presentation and evaluation of the optimization techniques are organized around two dimensions of their design space. The first dimension concerns the strategy for envelope call insertion and distinguishes between *eager* and *lazy* insertion. The second dimension concerns the speculative inlining strategies for envelopes.

Replacing every join point shadow instruction with an invocation of the appropriate envelope method eagerly is the same approach as is performed without virtual machine integration, described in Section 2.4.1. Without VM integration, the indirection imposed by envelopes already is frequently inlined. In the prototype of VM integrated envelopes, the optimizing compiler's inliner, however, is modified to ensure the indirection is always inlined. The way envelopes are generated, guarantees that the call from a proxy envelope to the enveloped method is always statically resolvable, the same holds for calls to field accessors (see Section 2.4.1). Hence, an enveloped method can always be inlined into its proxy and an accessor envelope can be inlined into its caller.

Another configuration of the developed prototype creates envelopes at class loading-time, but only inserts calls to an envelope method when advice is attached to it. That is before any aspect deployment is performed, the application's bytecode is not changed and performs method calls and field accesses as usual. However, when advice is woven into an envelope, all calls to the enveloped methods or accesses to the enveloped fields are replaced with calls to their envelope.

In both approaches dynamic weaving at a join point shadow requires to re-create the bytecode of an envelope method. Thus, after weaving, the en-

velope method must be re-compiled by the JIT compiler. Since it may have been inlined into other methods, care must be taken that every method executes the new version of the envelope method after deployment. Therefore, envelope methods must always be inlined by means of one of the speculative inlining techniques presented in Section 4.2.

While the result of both approaches is very similar, introducing envelopes eagerly or lazily has non-obvious effects on the performance. Those are discussed in Section 4.3.1 and evaluated in Section 5.2.2.

For the dimension of the speculative inlining technique for envelopes, it can be chosen from suite of techniques presented in Section 4.2. Section 4.3.2 discusses two approaches for invalidation to ensure correct dynamic weaving. The virtual machine integration of envelopes enables the extension and reuse of the VM's infrastructure for speculative optimizations. Additionally, it also allows the elimination of some of the limitations of envelope-based weaving discussed in Section 2.4. In Section 4.3.3 it is discussed how virtual machine integration enables a seamless support of special language features like advising reflective method and field accesses.

4.3.1 Eager versus Lazy Envelope Call Insertion

Eager Envelopes. When envelope calls are eagerly inserted into the application's code, an additional indirection is introduced into the application's bytecode. To ensure a high performance of the application, this indirection is optimized away for those envelopes which do not contain any advice.

The *baseline compiler* provided by the Jikes RVM does not offer the possibility to inline method calls. Consequently, the indirections are not removed from baseline compiled methods. But methods that are hot will be promoted to a higher level of optimization at which envelopes are optimized accordingly. With the exception of the envelope call, the baseline compiler generates the same machine code as it does in the absence of envelopes.

Jikes' *optimizing compiler* rewrites the intermediate code in multiple passes. In this prototype, the first pass is enhanced to specially handle envelopes such that the intermediate code is equivalent to what the unmodified compiler generates. Consequently, subsequent optimizations are carried out the same way in both cases.

The decision whether a method is inlined or not is made by the so-called *inliner* and depends on several conditions. The inliner uses heuristics—some of them are discussed in the following—to determine whether a method *callee* should be inlined into another method *caller*. By construction it is ensured that enveloped methods as well as accessor envelopes are always monomorphic, i.e., can always be inlined into their caller. Although the inliner can

also detect this, some of its heuristics prevent it from always making these decisions. Hence, it is modified to always decide to inline enveloped methods and accessor envelopes. When the inliner decides to inline an envelope, it is further modified to decide that a guard must be applied, to facilitate dynamic aspect deployment. For the implementation of this prototype, three inlining heuristics are of interest: *inline size*, *inline sequence length* and *field analysis*.

The inliner considers the size of the called method when deciding whether or not to inline it; the shorter the method the more beneficial is inlining. For envelopes the inliner would make its decision based on the proxy method's bytecode size. Proxies interfere with size estimates because the proxy itself is small, but it is guaranteed to have a (possibly large) enveloped method inlined into it. Thus, the inliner is modified so that it does not inline a proxy unless it would have inlined the corresponding enveloped method.

Figure 4.3 illustrates why the inline sequence heuristic needs to be modified in the presence of envelopes. It sketches the intermediate code generated by the optimizing JIT compiler when compiling a method m. On the left hand side, the generated code without envelopes is shown. On the right hand side, the code is shown as compiled in the presence of envelopes. The inline sequence begins with the method compiled in the first place, i.e., m in the example. When the compiler reaches the call to o (marked by an asterisk) without envelopes the inline sequence is (m, n) and has the length two. When the same code is compiled with envelopes, the inline sequence at the invocation of o is (m', m, n', n) where m' and n' are the (renamed) enveloped methods and m and n are their proxy envelopes. Thus, the inline sequence length is now four instead of two. If two is the maximum inlining depth, o would be inlined in the case without envelopes and not inlined with envelopes. In the prototype, proxy methods are optimized away when they do not contain advice. As a consequence, unadvised envelopes do not increase the size of the compiled code and, therefore, the inline sequence length heuristics is adjusted such that they are not counted.

Finally, for values resulting from a field read access the optimizing compiler applies special analyses on field types which, in turn, are used by the inliner. As field accesses are replaced by calls to accessor methods, the analysis is modified to be also applicable to accessor envelope calls and to lead to the same result as it would have had when applied to the access itself.

The *adaptive optimizing system* (AOS) also is modified in the prototype. Before optimization and inlining occur, envelopes execute as explicit method calls which are profiled by the AOS. The existence of a large number of envelope calls pollutes the profile data, causing compilation to occur at different times than usual, and delays optimization of some of the application meth-

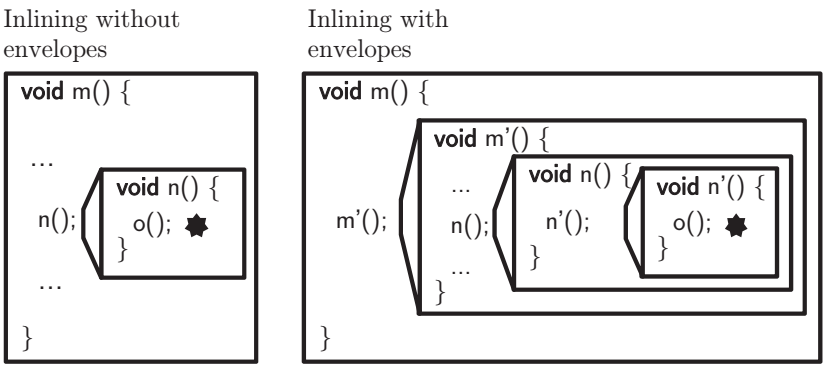


Figure 4.3: Inline sequence without and with envelopes.

ods. As a result, startup time is degraded and the application has to run longer before it reaches the fully optimized state.

To cope with this problem, Jikes RVM's profiling facility is modified so that if an enveloped method is sampled at runtime the sample is credited to the proxy envelope instead. As a result, the proxy is identified as a hot method, re-compiled with optimizations, and the inlining modifications ensure that the (hot) enveloped method is inlined into the proxy, thus compiled with the same optimizations.

Lazy Envelopes. The approach of eager envelopes requires little infrastructure provided by the virtual machine. It works with any kind of speculative inlining. However, only the final performance is comparable to that of executing an application without envelopes. Avoiding the slow start-up due to unoptimized envelopes in baseline compiled code, can be achieved by inserting the envelope indirection lazily: envelopes are created at class loading-time, but are not invoked if they do not contain advice. When the JIT compiler compiles a method call or a field access, it determines if the applicable envelope method contains advice. Only in this case, the JIT compiler generates a call to the envelope method. Otherwise, code for directly calling the enveloped method or accessing the field is generated. Thus, if no aspects have been deployed, the executing program is essentially identical to the original program without envelopes. However, to enable weaving in the envelopes and to let them take effect requires a sophisticated invalidation mechanism.

When advice is deployed at runtime, it is woven into the envelope method that corresponds to the relevant method or field access just like in the eager envelopes approach. But, since envelopes are no longer being called by default, all code performing either a call to the advised method or an access of

the advised field must be dynamically updated to invoke the newly advised envelope instead.

Because the envelope indirection is also missing in the baseline compiled code, both the baseline and optimizing compiler are modified to enable injection of the indirection into the compiled code. Therefore, both compilers have to record dependency information for all compiled calls and field accesses, to identify code to be updated when an advice is deployed.

Lazy envelopes ultimately achieve the same result as eager envelopes, but the lazy approach is operating at a lower level and is manually performing a number of tasks handled automatically in the eager approach. With eager envelopes, the inliner handles the inlining of envelopes, which automatically provides the necessary dependency tracking and invalidation mechanism. However, lazy envelopes are more effective at eliminating the overhead of envelopes, since envelopes are essentially eliminated from the system until needed; no modifications to the inlining heuristics and adaptive optimization system are needed. In addition, the code generated by the baseline compiler no longer contains unnecessary envelope calls, which minimizes overhead during program start-up.

4.3.2 Speculative Inlining Techniques for Envelopes

When weaving occurs, the affected envelopes must be recompiled and all compiled code containing inlined copies of these envelopes must be invalidated; for the discussion in this subsection, omitting the envelope call in the lazy envelopes approach is considered equivalent to inlining an envelope method in the eager approach.

To track envelope method inlining dependencies, Jikes RVM's existing method dependency lookup table is re-used. When an envelope is inlined into a method, a dependency is added to this table. During dynamic weaving the existing invalidation routine is called passing the envelopes to which advice is attached. This call triggers Jikes RVM's mechanism for invalidating all compiled method bodies that contain an inlined copy of the envelopes without the attached advice. At the next invocation of these methods they are re-compiled.

After dynamic weaving has occurred, several compiled methods may have been invalidated and need to be re-compiled. The adaptive optimization system of the RVM treats these methods in the same way as it does at program startup, i.e., initially compiling them at a low level of optimization and promoting them to higher levels over time. This approach smooths out the performance impact of dynamic weaving and avoids delays that may occur if all affected methods were immediately optimized aggressively.

To invalidate methods that are currently active on the stack, two common techniques as discussed in Section 4.2, can be used, namely guarded inlining with code patching and on-stack replacement.

Guarded Inlining with Code Patching. Inserting guards on all inlined envelopes is relatively easy using Jikes RVM's infrastructure for speculative inlining. But even though the guard itself has no runtime overhead, the presence of the full dispatch path interferes with a number of optimizations (see Section 4.2). Jikes RVM performs a simple local *code splitting* pass, to help exploit optimization opportunities created by guarded inlining. Using envelopes, however, creates a larger number of guarded inlines, thus stressing this splitting infrastructure. To address this problem, Jikes RVM's implementation of *code splitting* and *redundant guard elimination* is extended.

A new splitting algorithm has been developed in this thesis to perform slightly more aggressive splitting. It is similar to the algorithm for feedback-directed splitting described in [AHR02], but for simplicity does not use profile information. The algorithm maintains a work list of merge points. A merge point for the algorithm is defined as a basic block with at least one *infrequent* and one *non-infrequent* incoming edge. A basic block is a sequence of instructions that are always executed sequentially, i.e., branching instructions can only be the last instruction of a basic block. an edge is infrequent if it either leads to a basic block with a full dispatch or if the edge is reachable by a path containing an infrequent edge. It is desirable to eliminate merge points because these basic blocks cannot be optimized taking data flow analysis results of the inlined code into account.

The left hand side of Figure 4.4 illustrates this setting: The boxes 1–5 are basic blocks of a method connected by arrows that represent the possible control flows through the method. Basic block 1 ends with a branch to either block 2 or 3; 2 is the code of an inlined envelope and 3 is the full dispatch. After any of blocks 2 or 3 the basic block 4 is executed, and thereafter block 5. It is expected that the inlining decision of the envelope will not break, i.e., the basic block 2 is executed regularly, or *non-infrequently*, and 3 is executed *infrequently*. Therefore, basic block 4 is identified as a merge point and no optimizations can be applied to 4 that are based on data flow information gathered by analysis of basic block 2.

The work list is initialized with all the control flow merges created by guarded inlining. Afterwards, each basic block in the work list, representing a merge point, is duplicated. The infrequent path is directed to the duplicated

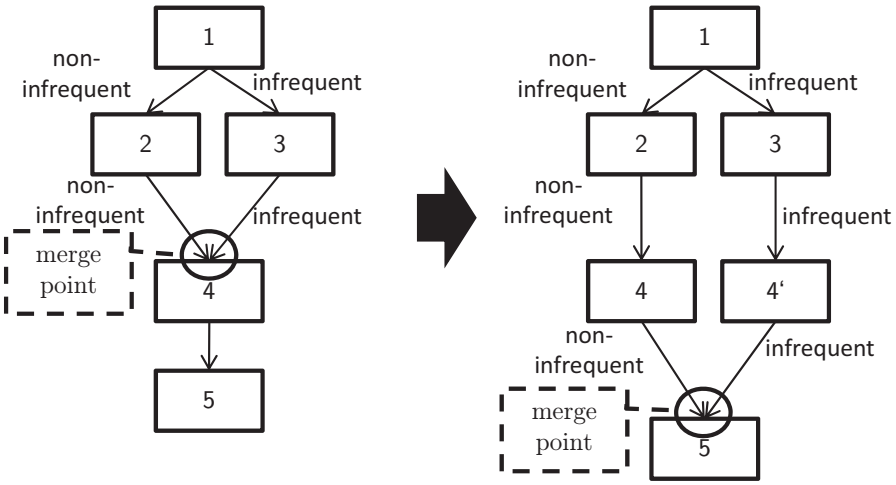


Figure 4.4: Diagram of the code splitting algorithm using merge points.

block; the non-infrequent path remains unchanged[1]. If duplicating the block creates a new control flow merge it is added to the work list. The algorithm continues until all merge points in the work list are processed, or a space bound is reached because of the code duplication.

The right hand side of Figure 4.4 shows the result of the first iteration of this algorithm. The work list is initialized with basic block `4` which is duplicated by the algorithm: one copy (basic block `4`) is only executed after basic block `2` and the other one (`4'`) is only executed after basic block `3`. After performing this split, basic block `5` becomes a merge point and the algorithm continues if the space bound is not already reached. When compiling `4`, optimizations can be applied derived from data flow information of `2`. Since `4'` is executed infrequently, it does not need to be optimized aggressively.

The size thresholds for code duplication can be varied to adjust the aggressiveness of the splitting. Similar to method inlining, more aggressive splitting has the potential to produce more efficient code, but will consume more compile time and compiled code space.

While this splitting algorithm has been designed to improve the performance of envelopes, it has also been discovered that it improves the performance of the base Jikes RVM (independent of using envelopes). The performance of the `mtrt` benchmark of the SPEC JVM98 benchmark suite [SPE] was improved by over 20%.

[1]The algorithm is also applicable in situations other than speculative inlining of envelopes where multiple infrequent or non-infrequent paths to a basic block may exist. In this case all infrequent paths are directed to one copy of the basic block and all non-infrequent paths are directed to the other one.

A desirable side effect of splitting is that many guards may become redundant and can be removed. Additional optimizations are enabled by Jikes' thread switching implementation based on yield points: if invalidation did not occur by the time the previous yield-point was executed, it is guaranteed not to occur until the next yield-point executes. To exploit optimization opportunities opened up by code splitting and the property of invalidation happening only at yield points, a redundant guard removal phase has been developed that focuses specifically on removing code patching inline guards. Guards are redundant if they are guaranteed never to fail.

This optimization is implemented in Jikes RVM as a linear pass by traversing the dominator tree, and propagating guard "liveness". In this case, liveness means that the code is safely protected by an existing guard, so no further guards are needed. The "true" branch of a guard creates liveness, and liveness is killed by yield-points, calls, or a control flow merge that contains a non-live incoming edge.

In addition to improving envelope performance, this optimization also has provided modest performance improvements for the base version of Jikes RVM (without envelopes), improving `mtrt` by about 3%.

With eager envelopes, envelope calls are not optimized in the baseline compiler, so invalidation is needed for the optimizing compiler only. In the lazy envelopes approach, invalidation is needed in baseline code as well, so patch points are placed in code compiled by both the baseline and optimizing compiler and the baseline compiler also contributes to the dependency lookup table. Two additional changes are necessary to Jikes RVM.

1. Jikes RVM does not normally use guards in combination with static methods because call sites of static methods cannot be affected by dynamic class loading. However, call sites of static methods can be affected by dynamic aspect deployment, because their envelope may be inlined. Hence, those parts of the JIT compiler have to be modified that assume that a guarded inline must have a receiver object.

2. For guarded inlining of polymorphic methods the JIT compiler of the Jikes RVM always uses a class test as the guard. Thus, when inlining polymorphic envelopes, the envelope aware Jikes RVM additionally inserts a patch point prior to the class test, to allow for invalidation of the inlining of the envelope when dynamic weaving occurs.

On-Stack Replacement. Jikes RVM contains an implementation of on-stack replacement (OSR), described by Fink and Qian [FQ03] and reviewed in Section 4.2. Using Jikes RVM's mechanism of OSR-based speculative inlining

works reasonably well for envelopes; however, given the expected frequency of envelope calls in the code one further step of optimization is applied. A patch point and OSR point are not placed at each inlined envelope call site, but only behind all yield-points and calls, similar to the redundant guard removal algorithm discussed above. When the method is invalidated, these patch points are all patched and the OSR point will be executed once the next yield-point is reached.

To ensure correctness in this model, when invalidation occurs, the VM must wait for all threads to progress to the next yield-point to ensure that OSR was triggered if necessary. Methods that are further up the call chain perform OSR when the stack frame is popped and the patch point after the call site is executed. A similar technique of waiting for threads to reach the next yield-point is used in [AR02] to allow removing additional redundant guards, thus creating larger regions of guard free code.

In addition to adding the facility of generating patch points, the baseline compiler is also extended to generate OSR points to facilitate lazy envelope call insertion.

4.3.3 Special Language Features

Envelope-based weaving affects *Reflection*, provided by means of an API in Java, [Java] in different ways. Java's Reflection allows for introspection of classes and their members, based on the names of classes, methods and fields. The reflective objects of methods and fields can be used to invoke the represented method, respectively access the represented field. In the envelope-based weaving approach, methods are added to classes which should not be accessible to the programmer who expects a class' structure to be as it is defined in the source code. To cope with this, Jikes' reflection mechanism is modified to omit accessors and enveloped methods, so that they remain hidden from the programmer introspecting the application.

When the programmer deploys an aspect which advises calls to certain methods or accesses to certain fields, he/she expects that also reflective calls or accesses are advised. This is an issue for every aspect weaving approach, not just for envelope-based weaving. A solution could be to instrument reflective operations with code to check at runtime whether there is an advice for the operation to be executed; in this case, the advice must also be called. However, as stated in [BVD05], an aspect-aware reflection mechanism provided at VM-level is more appropriate.

Since in the envelope-aware Jikes RVM proxies are generated with the name and the signature that their enveloped method originally had, reflective Method objects retrieved by a method name already encapsulates the ap-

propriate proxy envelope. Reflective invocation, hence, executed the proxy including its advice. In contrast, the Field object from the Reflection API provides methods for setting and getting the value of fields, bypassing envelopes. The implementation of this class is modified in order not to perform a memory access for writing or reading the field but to call the respective accessor envelope instead.

Native methods (marked with the **native** modifier) are methods whose implementation is not available as Java bytecode but as system dependent machine code provided by means of a dynamic library. To execute such a method, the virtual machine searches the loaded dynamic libraries for the function providing the appropriate implementation. The look-up makes use of a naming convention for the native function in the library [JNI]—the function's name is derived from the Java method's name and signature. Since methods are renamed at envelope creation, this look-up mechanism is adapted to still be able to find the native implementation (this is similar to the Java 6 agent outlined in Section 2.4).

4.4 Control Flow Quantification

The dispatch function at join point shadows generally has to evaluate dynamic properties specified in a pointcut. Such an evaluation always consumes execution time when the join point shadow is executed. Hence, it is desirable to speed-up the evaluation of dynamic properties. The situation is similar to virtual method dispatch; the property to be evaluated in this case is the dynamic type of the receiver object. An efficient data structure—a dispatch table—provides a general yet efficient approach to perform the dispatch. Furthermore, optimizations like speculative inlining, are capable of improving the performance of the dispatch even further in special cases.

The evaluation of dynamic properties based on the control flow that contribute to virtual join point dispatch is particularly time consuming because the control flow is not explicitly available in the execution environment. Instead, current AO language implementations track the current control flow in a way that it can be accessed by the code evaluating the dispatch function at a join point shadow. In this thesis, an optimized implementation of those dynamic properties has been developed that refer to the methods currently executing on the stack. More specifically, an optimization technique is developed for AspectJ's **cflow** pointcut designator.

As discussed in Section 3.2, legal AspectJ pointcuts can always be represented by combining one or more sub-pointcut expressions that have the form <staticProperties> && <dynamicProperties> with an *or* operator; when a

join point shadow selected that is by <staticProperties> is executed and at the same time <dynamicProperties> matches, the join point shadow's execution is selected as a join point by the sub-pointcut. Dynamic properties other than **cflow** are not of interest in this section; throughout the remainder of this section it is assumed that pointcuts containing a **cflow** designator are always of the form shown in Listing 4.5.

```
1 <staticProperties> && cflow(<pointcut>)
```

Listing 4.5: A pointcut *or* clause containing a **cflow** pointcut designator.

The sub-pointcut denoted by <staticProperties> describes the join point shadows whose dispatch function depends on the **cflow** dynamic property while **cflow**(<pointcut>) defines when the dynamic property is satisfied. The **cflow** designator is parameterized with a pointcut that matches join points whose control flow determines the execution scope of interest. In the following, these join points are called control flow *constituent* join points and mark *entries* and *exits* of control flows of interest. The join point shadows matched by <staticProperties> are called *dependent* shadows. In general, the pointcut used in the **cflow** designator can also refer to dynamic properties of join points constituting the **cflow**. Thus, the **cflow** is constituted at *join points*; in contrast, at dependent join point *shadows* the condition expressed by the **cflow** designator is tested as part of the virtual join point dispatch.

4.4.1 Current Implementation of Control Flow Quantification

There are different implementations of **cflow** in current AO execution environments, each addressing the following two issues:

1. At constituent join points, actions need to be taken to monitor the state of the control flow such that it becomes possible to determine at dependent shadows whether certain control flows are active.

2. At dependent shadows, the dispatch function depends on whether the control flow referred to by the **cflow** designator is currently active.

It is usually possible in AOP implementations to access context values of constituent join points, and to use such context information in advice attached to dependent join points. Therefore, in AspectWerkz, `abc` and `ajc` a stack is maintained which stores context values from the constituting join points. Control flow checks are implemented by testing whether the stack is

empty; in this case the control flow is not active. The context values can be accessed at a dependent shadow by reading the value from this stack.

When no access to context values from the constituent join point is required, the `abc` and `ajc` compilers utilize a more efficient infrastructure for **cflow**. Instead of growing and shrinking stacks that represent the control flows, *counters* are incremented and decremented. Each **cflow** is assigned a counter and code that updates the counter is inserted at its entries and exits. When a control flow matched by a **cflow** designator is entered, the corresponding counter is incremented; it is decremented when the control flow is left. This is necessary because a control flow can be entered recursively. Thus, a control flow matched by a **cflow** designator is activated when a constituting join point is entered the first time and it is deactivated when the join point is exited the last time. At dependent shadows, the residue checks whether the counter is greater than zero. If so, the control flow is active and the dynamic property is satisfied.

Counters must exist once per thread for this approach to work; otherwise, different threads entering and leaving the same control flow would corrupt the counter states. To realize thread-locality, the ThreadLocal class from the Java class library is used to manage the counters.

The `abc` [ACH+05b, ACH+05a, ACH+04] compiler performs intra- and interprocedural static analyses to improve the performance of code using **cflow** pointcuts [SdM03].

Of the *intra*procedural optimizations implemented in `abc`, only one is of further interest with regard to this section. Others deal with binding context values from constituent join points, which is out of the focus of this prototype. The interesting optimization is the one that ensures that counters are reused throughout a method [ACH+05b]. From the observation that retrieving a counter from thread-local storage can be expensive, the implementors of `abc` have derived the following optimization. Whenever a control flow counter is required several times in a method the counter is stored in a local variable and has to be retrieved from the thread-local storage only once. For example, at control flow constitution, the counter is updated via the local variable at entering and exiting the control flow; if multiple join point shadows appear in a method and depend on the same **cflow** dynamic property, or if a dependent shadow is executed in a loop, the counter only has to be retrieved from the local variable. Since local variables are implicitly thread-local, this optimization is obviously correct.

Furthermore, as a result of *inter*procedural analysis, `abc` can completely avoid weaving **cflow** infrastructure at some join point shadows. Due to its cost in terms of time and memory consumption, the interprocedural analysis is only activated at the highest optimization level which can be chosen by

the user when compiling AspectJ source code. The interprocedural analysis determines three sets of join point shadows for each pointcut expression containing a **cflow** designator. The computed sets are as follows for a pointcut of the form <staticProperties> && **cflow**(<pointcut>).

1. The first set contains those shadows with the specified <staticProperties> that *may* be executed in a control flow constituted by a join point that matches <pointcut>. At the shadows contained in this set, residual advice dispatch must be woven.

2. The second set contains shadows with the specified <staticProperties> that are guaranteed to occur *only* in a control flow constituted by join points matched by <pointcut>. At these shadows, advice invocation can be performed unconditionally.

3. In the third set, join points matching <pointcut> are contained that *may* influence the evaluation of residues at shadows in <staticProperties>. At the shadows of these join points, counter or stack maintenance must be woven.

Interprocedural analysis [ACH+05b] exploits a call graph of the entire application, which is why *all* classes reachable from the application's entry points must be known at compile-time. That is, abc performs a *whole-program analysis* that needs to know all possible entry points and the class files for all classes reachable from there. This puts Java applications under a closed-world assumption, which contradicts Java's dynamic class loading capabilities. If the virtual machine dynamically loads classes that are not known at compile-time, new execution paths may be possible due to late binding of method calls in Java. If this happens, the interprocedural analysis becomes unsound.

Another implementation approach for **cflow** is to *inspect the call stack* at dependent join point shadows. To determine whether the control flow is active, the stack frames are walked down one by one. For each frame, the method is matched against the constituent's pointcut. If the match succeeds the current join point is executed in the desired control flow.

In Java, the call stack can be accessed by creating an instance of Throwable, which can be queried for the stack frames via its getStackTrace method. JAsCo [SVJ03], JBoss AOP [JBo], and Prose [PGA01] follow this approach.

The stack inspection approach does not require any infrastructural code at control flow entries and exits. Thus, there is no cost imposed at these points. Also, no measures have to be taken for ensuring thread locality, because the call stack that a residue accesses is *always* the one of the currently executing

thread. However, the cost imposed by stack walking at dependent shadows is high and not constant; it directly depends on the depth of the call stack. In cases where the control flow is not active, the entire stack must be walked and inspected.

In the first version of Steamloom$_1$ [Hau06, HMB+05, BHMO04][2], Haupt et al. have implemented the approaches based on counters respectively stack walking at the virtual machine level. Compared to implementations restricted to Java bytecode generation and using the functionality of the standard class library, the VM integration led to an improved performance of updating and querying thread-local values and of accessing the call stack respectively. This implementation has shown that stack walking is still not feasible even with virtual machine support. The counters at virtual machine level outperform the previous implementations in the multi-threaded case. In a single-threaded application it performs slightly worse than `abc` because neither inter- nor intraprocedural analyses are applied.

4.4.2 Control Flow Guards

Similar to Steamloom$_1$'s strategy, the prototype presented in this section is also based on integrating **cflow** into the virtual machine. In addition to utilizing more efficient data structures as is already done by the above outlined optimizations in Steamloom$_1$, this prototype uses a completely novel approach for maintaining information about active control flows and exploits analysis results available in the JIT compiler. However, it is important to note that these analyses do *not put the application under a closed world assumption.*

The technique developed in this thesis is inspired by the concept of guarded inlining. Guards are lightweight tests of inlining assumptions. For an efficient implementation of **cflow**, in this thesis the approach of *thin guards* [AR02] is extended.

The primary application of thin guards is to reduce the performance penalty of dynamic class loading in Java. As discussed in Section 4.2, the JIT compiler speculatively inlines methods that are monomorphic but may become polymorphic when a class is loaded that overwrites the method. Such speculative optimizations can be guarded by a lightweight check to ensure that correct execution will occur if the assumptions change in the future. The specialty of thin guards is to make the test extremely cheap.

[2]The suffix "1" is used in the following as a distinction to the new version Steamloom$^{\text{ALIA}}$ discussed in Section 7.2.2.

Throughout the executed application many assumptions are made, each one relying on the fact that a specific method is not overwritten. Before executing the optimized code, it must be tested if the assumption is still valid and this test has to be as efficient as possible. As discussed in Section 4.1 modern virtual machines employ code patching or on-stack replacement instead of such a test, but thin guards have been invented as a cheaper alternative to class tests that requires less infrastructure than code patching and OSR.

All the assumptions that are made are mapped to a vector of condition bits, e.g., stored in a word that is kept in a register or at a constant memory location. To check an assumption this word is loaded and the corresponding bit is tested. When an assumption is invalidated, the bit this assumption is mapped to is updated to reflect that the assumption is invalid. Because multiple assumptions are mapped to the same bit, some assumptions which are actually still valid are invalidated as a side effect. But that is for the sake of an easy and efficient invalidation mechanism.

From the concept of thin guards, the idea has been borrowed to store conditions in the bits of a quickly accessible word and to let the bits reflect whether the condition is currently satisfied or not. In the optimization for **cflow** presented here, the VM maintains a guard bit for every relevant control flow, and this bit is updated on entry/exit to that control flow. These bits can be stored as a bit vector in the thread for efficient access.

In contrast to the use of thin guards for speculative inlining, each **cflow** is mapped to a distinct bit. Sharing bits between inlining assumptions is semantically correct because performing the full dispatch is always correct. In contrast, **cflow** test must be successful if and only if the control flow denoted by the **cflow** designator is currently active.

To exemplify the approach, consider the example in Listing 4.6 which shows a recursive implementation of a function to compute the n^{th} Fibonacci number. The sub-pointcut in line 12 determines the depending join point shadows; these are the calls to fib in lines 3, 7 and 8. These shadows only become join points when they are executed in the control flow of fib which is specified by the sub-pointcut in line 13, i.e., the first call to fib is not selected, but all subsequent recursive ones are.

The control flow is entered at the beginning of method fib in line 5 and left after this method in line 9. Assuming that the **cflow** declared in line 13 is mapped to index 0, Listing 4.7 conceptually shows how the two methods test and fib are compiled when employing thin guards. As before in this thesis, the example is shown in terms of Java code for simplicity. Actually, the **cflow** guard bits have no representation at source code or bytecode level.

The execution of the fib method constitutes the control flow in question, hence, at the beginning of this method in line 10 the **cflow** bit is set to 1,

```
1  class Fib {
2    public int test() {
3      return fib(5);
4    }
5    public int fib(int n) {
6      if (n <= 1) return 1;
7      return fib(n−1) +
8        fib(n−2);
9    }
10 }
11 aspect Aspect {
12   before() : call(int Fib.fib(int)) &&
13     cflow(execution(int Fib.fib(int)))
14   {
15     // ...
16   }
17 }
```

Listing 4.6: Recursive implementation of Fibonacci numbers with an aspect intercepting recursive calls.

indicating that the control flow is active. The pseudo-object thread represents the current thread, while cflowState represents the bit vector used to track the active control flows of each thread. In line 21, the exit of the control flow is reached. However, the **cflow** bit cannot simply be set to 0 because the corresponding control flow may already have been active when the method fib has been entered. To cope with recursive control flows, the **cflow** word is stored at the beginning of the method in line 9 before it is updated and restored at the end of the method in line 21.

At dependent shadows in both test and fib (lines 3, 16 and 18), the same bit is tested to determine whether the **cflow** dynamic property is satisfied.

As soon as a pointcut-and-advice with a **cflow** dependent pointcut is deployed, an index within the **cflow** bit vector is assigned to the control flow referred to by **cflow**. This mapping from **cflow** pointcut designator to its index in the **cflow** bit vector can be accessed by the JIT compiler. Thus, when it compiles the code for updating or querying a control flow specified by a **cflow** designator, the index of the bit to be updated, respectively tested, can be considered a constant.

```
1  int test() {
2    int result;
3    if((thread.cflowState & 1) != 0) advice();
4    result = fib(5);
5    return result;
6  }
7
8  int fib(int n) {
9    int oldState = thread.cflowState;
10   thread.cflowState |= 1;
11   int result;
12   if (n <= 1) {
13     result = 1;
14   } else {
15     result = 0;
16     if ((thread.cflowState & 1) != 0) advice();
17     result += fib(n-1);
18     if ((thread.cflowState & 1) != 0) advice();
19     result += fib(n-2);
20   }
21   thread.cflowState = oldState;
22   return result;
23 }
```

Listing 4.7: Woven pseudo code using **cflow** word.

4.4.3 Implementation of Control Flow Guards

To implement control flow guards, several of Jikes RVM's components have been extended. To store **cflow** state information thread-locally, the virtual machine's multi-threading design is exploited. One word is added to every thread object, and this word is used to store the thread's **cflow** state. Jikes RVM uses a mixture of preemptive and cooperative *multi-threading*. A small number of operating system level threads, which are called *processors* in the Jikes RVM, can execute a large number of Java threads. A processor is represented as an instance of the VM_Processor class and a special hardware register—called the *processor register*—always holds a reference to the current processor. To allow for a more efficient access to the guard bits, they are copied into the call stack frame of methods that either constitute or depend on control flows. This requires an extension to Jikes thread-switching logic and the layout of call stack frames. Both the baseline and the optimizing compiler, have to be extended to generate the code reflecting the layout.

Each processor object has a list of thread objects that are alternately executed by the processor. An object representing the currently active thread is referenced by a field of the processor object.

One word of storage is allocated per thread to accommodate **cflow** state bits, called the **cflow** *word* in the following. This allows for monitoring 32 different control flow pointcuts, which should be enough for most applications. The system can fall back to a more conventional counter-based strategy if more than 32 different control flow pointcuts are used. The guard-based implementation can, then, be used for the 32 most frequently used control flows, and the fallback strategy for additional pointcuts. Furthermore, the number of bits used as control flow guards can be increased by using two or more words if applications using a large number of control flow pointcuts become common. As limitation the number must be fixed and cannot grow at runtime since the space for holding the guard bits must be allocated in all threads.

A copy of the active thread's **cflow** word is held in a field of the processor object. Upon every thread switch the value in the processor object is synchronized with the value in the thread object. Since the address of the processor object is always held in a register and the position of the **cflow** word in the processor object is a constant offset known at compile-time, the bit vector can be accessed with as little overhead as a single memory load or write operation.

Baseline Compiler. For the baseline compiler, the operations of **cflow** constitution and check are implemented in a straightforward way. For accessing

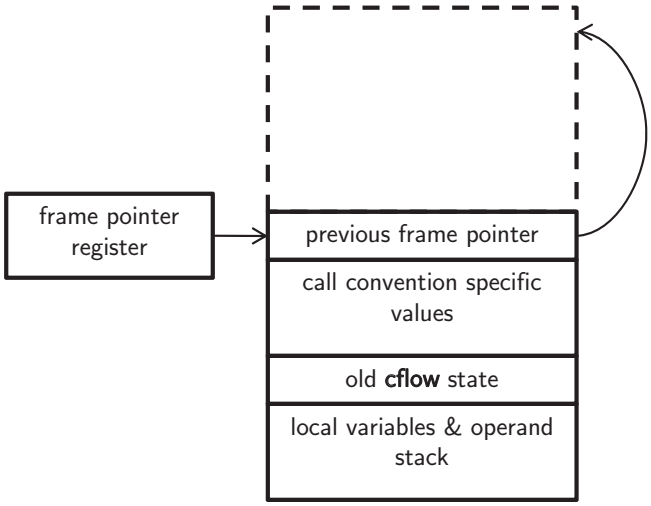


Figure 4.5: Stack frame layout for baseline-compiled methods.

the **cflow** word, a memory load or write operation is used. To test or modify a single bit, standard bit operations (like bitwise *and* or *or*) are used.

At constituent shadows, an additional word is allocated in the method's stack frame. Before setting the **cflow** bit, the old value is copied into this word. When the method is left, it is copied back into the processor object's field to restore the **cflow** state.

Jikes' stack frame layout for baseline-compiled methods is shown in Figure 4.5. The old control flow state is stored in a special slot of the stack frame, which is located at offset stack_offset.

Optimizing Compiler. Support for the **cflow** word in the optimizing compiler is implemented using the high-level intermediate representation generated during the first phase of the optimizing compiler. At this stage, an unlimited number of virtual registers is available. In later phases, these registers are mapped to the (limited) set of physical registers. If the number of physical registers is not sufficient to hold all virtual registers used in the previous phase, they will automatically be stored in and loaded from memory.

The optimizing compiler is extended to load the control flow state information into a virtual register, called the **cflow** *register*, at the beginning of a method. When a control flow is entered at a constituent shadow, three steps are performed.

1. The current value of the **cflow** word is stored in a separate virtual register called *backup register*.

2. The control flow word in the **cflow** register is modified.

3. The processor object field is updated with the new value of the **cflow** register.

Upon leaving the constituent join point, the virtual register and the processor object field are restored from the backup register. When control flow state has to be tested at dependent shadows, it can be accessed directly from the virtual register it has been stored into at method entry.

If the compiler decides to inline a method into another method, the inlined method's high-level intermediate representation is generated independently and then inserted into the outer method. This implies that it uses separate **cflow** and backup registers for storing control flow state. This is indeed necessary for correct behavior, e.g., if the inlined method constitutes the control flow. In this case, the **cflow** and backup registers of the outer and inlined methods hold different values.

At first sight, this compilation strategy does not seem to differ significantly from the one described for the baseline compiler. It might even look less efficient, because control flow state is read at the beginning of every method, although it is probably required only in a small fraction of the methods executed. However, since the approach operates on the high-level intermediate representation, the optimizing compiler applies all its standard optimization techniques in later phases. Those relevant for this prototype are presented below.

- If the virtual register holding control flow state is never read, the compiler detects this and eliminates the memory load operation that initializes the register. Thus, methods that do not access control flow state do not exhibit any overhead.

- If control flow state information is frequently required (e.g., when a dependent shadow appears in a tight loop), it is loaded only once from main memory and kept in a physical register. Basically, by using a virtual register for storing control flow state, the decision on whether to keep the value in a physical register or in memory is left to the optimizing compiler's advanced algorithms.

- The same applies to the old control flow state value, which is saved at constituent shadows. Again, the compiler can decide whether to keep it in a physical register or to store it in memory, based on how many registers are needed by the method.

At dependent shadows, the control flow state is accessed directly from the **cflow** register of the caller. When the control flow is left, the processor object is updated from the value stored in the backup register. In addition to the standard optimizations performed by the compiler, the prototype of the **cflow** optimization employs two custom optimizations in the optimizing compiler:

1. If a method is inlined, the backup register is not initialized by loading it from the processor object. Instead, the value is copied from the outer method's **cflow** register. In addition to saving a memory load operation, this optimization allows the compiler to eliminate some virtual registers if one of the methods does not modify control flow state. In this case, subsequent phases of the compiler can infer that both virtual registers hold the same value and thus map them to the same physical register.

2. When the JIT compiler encounters a **cflow** check during the compilation of a method that constitutes the control flow referred to by the same **cflow**, the check is omitted because it always succeeds. If the dependent shadow is in an inlined method this optimization also checks if the relevant control flow is constituted by the caller method. Since Jikes performs nested method inlining, the optimization even considers several nested calls. For the Fibonacci example from Listing 4.7, this optimization removes the guards for advice executions inside the fib method.

The second of the above optimizations can be seen as a somewhat weaker form of `abc`'s interprocedural analysis. It does not perform a whole-program analysis, which is not feasible in a virtual machine due to time and memory constraints. Instead, it is restricted to the set of methods inlined into the method currently being compiled. This set will always be reasonably small, as the compiler avoids creating large method bodies. The advantage of this approach is that eliminating the test does not put the application under a closed-world assumption because the test is only eliminated in a copy of the method that is guaranteed to be always executed in the desired control flow.

Chapter 5

Evaluating Dynamic Optimizations of AO Concepts

In Chapter 4, optimization techniques for two aspect-oriented concepts, namely dynamic aspect deployment and **cflow***-based join point constraints, are presented. To evaluate their effectiveness, benchmarks have been executed on the developed prototypes as well as on existing systems with comparable support. Since no elaborate benchmark suites exist that measure such dynamic features of aspect-oriented execution environments, appropriate benchmark approaches have been developed in this thesis. The benchmark approaches and the results of running them are presented in this chapter. The results show the superiority of the optimizations developed in this thesis as compared to other implementations of the same concepts.*

Parts of this chapter have been published in the following papers.

1. *Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. In* Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006*

2. *Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient Control Flow Quantification. In* Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006*

3. *Christoph Bockisch, Mira Mezini, Wilke Havinga, Lodewijk Bergmans, and Kris Gybels. Reference Model Implementation. Technical Report AOSD-Europe-TUD-8, Technische Universität Darmstadt, 2007*

5.1 Benchmarks for Dynamic Features of AO Languages

To evaluate the effectiveness of the optimizations presented in Chapter 4 the performance of the developed prototypes is compared to that of related approaches. To date there is no comprehensive benchmark suite to compare dynamic AO execution environments with respect to the performance they exhibit when executing realistic applications. Furthermore, because currently no applications are available that excessively use dynamic aspect-oriented features, it is not possible to use existing applications as benchmarks; and without real-world samples, it is not possible to know how exactly the dynamic AO features will be used.

For these reasons, in this thesis established object-oriented benchmarks have been extended by deploying synthetic aspects to them. Synthetic means that the aspects do not perform anything functional beyond utilizing the aspect-oriented feature whose impact on performance is to be measured. To compensate the lack of knowledge about common use scenarios of the dynamic AO features, the aspects used in the benchmarks in this section provide scenarios with a mix of different characteristics.

The evaluations in this section are all based on the SPEC JVM98 [SPE] benchmark suite, version 1.04. This benchmark suite is intended to measure the performance of Java virtual machines and consists of seven applications[1] that make use of Java language features, such as object creation, floating point operations, or multi-threading in different ways. Five of the benchmark applications have been derived from real-world applications. All benchmark runs are performed with the maximal input size as recommended by the run rules. Each benchmark application is executed in its own virtual machine instance in order to expose each benchmark to the same start-up behavior of the virtual machine.

The SPEC JVM98 benchmark harness executes each application repeatedly, whereby the number of iterations can be configured. In the evaluations reported in this section, the number of iterations has been chosen to be large enough for the applications to reach steady-state. The steady-state is reached when all initializations of the virtual machine that potentially execute in parallel with the application are finished and the adaptive optimizing system has identified all hot methods. Hence, no methods are re-compiled in the steady-state.

[1] There is an eighth application in the benchmark suite which however just tests the correctness of the virtual machine. Thus, according to the official run rules, this application (`check`) is not included in the evaluations in this section.

To measure the **cflow** optimization, in addition to the SPEC JVM98 benchmarks, so-called micro-benchmarks [HM04a, PAG03] are used to measure the performance of single operations like constituting or querying a control flow. The application used to perform the micro-benchmark of an operation is written in a way that the execution environment cannot apply optimizations to the operation. Thus, micro-benchmarks determine the worst-case performance imposed by the operation.

The approaches compared in this section are realized as extensions of different Java virtual machines or generate code targeted toward specific virtual machines. Thus, it is not possible to directly relate the execution times of benchmark application runs. Instead, the ratio of the execution time on the extended JVM to that of the un-extended underlying JVM, called the *reference JVM*, i.e., the imposed runtime overhead, is calculated and compared. That is, an overhead of 1.0 means that a benchmark application is executed 1.0% slower by the evaluated approach than by the reference JVM; in other words, the execution time is 1.01 times as long. All measurements have been made on a Dual Xeon workstation (2x3 GHz) with 2 GB RAM running Linux 2.4.23. The overhead is presented throughout this chapter in percent. Tables always show the different benchmarked approaches as columns and the executed benchmark applications as rows.

5.2 Evaluating Optimized Dynamic Deployment

Different alternatives of the envelope-aware Jikes RVM implemented in this thesis, i.e., lazy versus eager envelopes and on-stack-replacement versus code patching, are first compared with regard to their start-up and steady-state performance. Thereby, only the impact of the facility for dynamic deployment is measured. This evaluation shows that lazy envelopes with on-stack-replacement offer the best performance. For this reason, only the approach of lazy envelopes and on-stack replacement is used to compare the envelope aware Jikes RVM to other execution environments for dynamic AOP in the second comparison.

5.2.1 Approaches Participating in the Evaluation

There are several publicly available AOP implementations for Java that support runtime weaving and that are directly comparable to the prototype implementation for dynamic aspect deployment presented in Section 2.5.2, called *Envelope-Aware* throughout this section.

AspectWerkz [Aspd, Bon03] provides dynamic weaving capabilities for sets of join point shadows which are specified before runtime. The weaving process [Vas04] is divided into two distinct phases: *preparation* and *activation*. In the preparation phase classes are transformed such that in the activation phase advice calls can be inserted at prepared join point shadows.

Preparation can be performed either by a post-compiler or by a special class loader. During preparation, AspectWerkz replaces each join point shadow with a call to a wrapper, which is similar to introducing envelopes. In the activation phase, advice calls are inserted into wrappers, as in our approach. Wrappers that changed during phase are replaced by means of the Java 5 standard feature Class Redefinition [Ins].

In contrast to envelopes approach, AspectWerkz' wrappers do not reduce the number of join point shadows where weaving happens. Instead of redirecting the call sites to a single wrapper generated for each callee, as done with envelopes, AspectWerkz generates one wrapper method call site respectively field access site. Furthermore, the generation of wrappers in AspectWerkz *does* affect the generated machine code, i.e., the generated machine code with and without wrappers is not the same and the optimizations applied by the JIT compiler differ in both cases.

JAsCo [SVJ03] uses a registry for aspects where all active pointcut-and-advice bindings are stored. At potential join point shadows so called traps are called which notify the registry and let it execute advice if any is applicable. JAsCo implements two optimizations of this approach [VS04]. The first optimization is implemented as the Jutta compiler which generates custom implementations of single trap methods that directly call applicable advice, thereby circumventing the registry. The second optimization is to insert call to traps only when aspects are deployed. Class Redefinition is used to replace a method without traps with a version including traps at runtime.

The lazy insertion of traps is similar to the lazy introduction of envelopes, as discussed in this thesis. Also, generating custom traps that contain woven code for specific join point shadows bears similarities to this work. The traps are generated similar to AspectWerkz's wrappers and thus affect the generated machine code.

Steamloom$_1$ [Ste, HMB+05] is an extension to the Jikes RVM [Jik] with dedicated support for aspects and dynamic aspect deployment. In contrast to other approaches, e.g., JAsCo or AspectWerkz, that only use bytecode toolkits as an external means for weaving, Steamloom$_1$ has integrated the bytecode toolkit BAT [BAT] into the VM. Similar to the prototype presented in Section 4.3, join point shadow search and advice weaving makes use of the VM's internal representation of loaded classes. Invalidating old versions of methods in which advice calls are woven makes use of standard

features of the virtual machine which are more specialized than Java 5 Class Redefinition; however, the VM's optimizations are not extended and thus do not specifically support dynamic weaving.

Similar to the Envelope-Aware prototype presented in Section 2.4, Steamloom$_1$ restricts the scope of join point shadow search for member accesses. To do so, for each member an index is stored that points to all join point shadows at which the member is accessed. When searching for join point shadows, Steamloom$_1$ only searches the corresponding index instead of the complete bytecode. However, Steamloom$_1$ does not localize the weaving operations for member accesses; still each retrieved shadow is manipulated individually, and, hence, weaving performance is still degraded. Steamloom$_1$'s weaving approach is basically that of the AspectJ compiler, except that it is performed at the VM level.

Throughout this section, AspectWerkz 2.0, JAsCo 0.8.7, Steamloom$_1$ 0.6, and PROSE 1.3.0 are used for benchmarking. For AspectWerkz, JAsCo and PROSE the HotSpot 1.5.0 JVM of Sun is used as reference execution environment. Steamloom$_1$ is based on Jikes RVM 2.3.1. JAsCo is executed using the suggested `HotSwap 2` variant and having the *inlinecompiler* and the *trapall* flags enabled. The first enables a new weaver which provides better performance, the second one allows to advise all methods, including **private** ones, as the other AOP environments also do.

5.2.2 Alternative Configurations of Envelope-Aware Jikes

Envelopes create two potential sources of overhead in the VM. Compile time is increased, as the compiler needs to take additional actions to inline the envelopes, and application runtime is increased if envelopes are not inlined properly, or if the necessary mechanisms for invalidating inlined envelopes hinder optimization. To measure this impact, the SPEC JVM98 benchmark suite is executed on top of *different configurations of Envelope-Aware* (introducing envelopes eagerly versus lazily and using OSR versus code patching as invalidation mechanism). The prototype is implemented as an extension of the Jikes RVM 2.4.1 relative to which the overhead of Envelope-Aware is reported in Table 5.1. The table summarizes the results for the different configurations in order to determine the best strategy.

In the first section of Table 5.1 the performance of eagerly introduced envelopes is presented. The second column, labeled "None Steady", shows the steady-state performance of Jikes RVM when eager envelopes are used, but no invalidation mechanism is used; this identifies the overhead introduced by the presence of envelopes separate from the invalidation mechanisms. The steady-state overhead in this setting is small, with a maximum

	Eager Envelopes					Lazy Envelopes	
Invalidation Phase	None Steady	Guards Steady	OSR Steady	OSR Start	Baseline Steady	OSR Steady	OSR Start
compress	0.0	-0.1	0.0	15.2	122.4	-0.6	5.8
jess	0.7	3.5	2.1	3.3	15.5	5.3	6.7
db	-4.8	-4.8	-4.8	2.7	8.4	1.0	-0.9
javac	3.7	5.3	4.2	18.3	17.8	4.0	3.3
mpegaudio	-0.6	6.9	-0.2	13.3	32.7	0.5	7.9
mtrt	-0.6	-1.4	-0.1	11.3	57.7	-10.6	13.4
jack	1.8	1.7	3.0	8.4	8.1	-2.0	2.9
average	0.0	1.6	0.6	10.5	37.5	-0.3	5.6

Table 5.1: Percent overhead of using eager envelopes in Jikes RVM.

of 3.7% (`javac`) and averaging 0%. In some cases envelopes actually reduce execution time (in particular `db`).

It may seem odd that some benchmark applications actually perform better on Envelope-Aware than on an unmodified Jikes RVM. However a deviation in this order of magnitude does not indicate an error in the benchmark approach. Any modification to a virtual machine can change the performance in very subtle ways [GVG04]. With every modification the memory layout is changed, i.e., the positions where objects and code are stored by the modified virtual machine differ from the positions in the unmodified one. As a result, cache locality can be changed coincidentally which results in more or less cache misses causing small deviations from the expected benchmark results.

The third column of Table 5.1, labeled "Guards Steady" reports the steady-state performance of a system using eager envelopes together with guarded inlining, code patching and splitting to enable invalidation for dynamic weaving, as described in Section 4.3.2. There is very little overhead introduced by using guards for envelopes, demonstrating that it is an effective mechanism for enabling invalidation.

The fourth column of Table 5.1, labeled "OSR Steady" shows the overhead that is present when eager envelopes are used *and* OSR points are inserted, as described in section Section 4.3.2. Guards perform roughly on par with OSR, demonstrating effectiveness of the splitting and redundant branch elimination algorithms being used.

The only exception is `mpegaudio`. Envelopes with guards result in a 6.9% degradation relative to the original Jikes RVM. After investigating the reason

of this overhead, it has been discovered that a guard is hindering optimization in a tight loop in `mpegaudio`; the loop contains a single basic block that is executed frequently, and the existence of a guard in that loop breaks up some of the optimizations performed by Jikes RVM's optimizing compiler. None of the optimizations are fundamentally blocked by the guard, but the loop optimizations in Jikes RVM are not particularly aggressive. Since the OSR-based implementation is superior in this evaluation, the guard-based approach is not further investigated in this section.

Despite eager envelopes' minimal impact on steady-state performance, start-up performance is more substantially affected. The fifth column, labeled "OSR Start" shows the performance of eager envelopes with OSR for the first run of the benchmarks. The start-up overhead ranges from 2.7% to 18.3% with an average of 10.5% degradation relative to the unmodified Jikes RVM. The start-up overhead is caused by the reduced performance of the baseline compiled code.

The sixth column in Table 5.1 ("Baseline Steady") shows the overhead caused by eager envelopes in steady-state when the optimizing compiler is disabled, thus code is compiled with the baseline compiler only. Because envelopes execute with no optimization in the baseline compiled code, when using eager envelopes the overhead is quite high, ranging from 8.1%–122.4%. When an application starts executing, all code is first compiled by the baseline compiler, thus short running programs, or "start-up" scenarios are degraded when the quality of the baseline compiled code is reduced.

Lazy insertion of envelopes enables lower start-up time overhead. The second section in Table 5.1 shows the overhead imposed on the SPEC JVM98 benchmark suite when envelopes are inserted lazily and OSR is used as invalidation mechanism. It can be seen in the "OSR Steady" column of this section that the steady-state performance is not affected at all. In fact, the performance is minimally better than that of an unmodified Jikes RVM, an explanation for such unexpected speed-up is given above. The last column shows the overhead at start-up imposed by lazy envelopes which is induced by the additional dependency tracking and additional workload of generating envelopes at class loading time. On average, this overhead is at 5.6%. All in all, at the steady-state, both the lazy and the eager envelopes approach impose no or no significant overhead. The start-up behavior of lazy envelopes, however, is superior creating only about half the overhead of eager envelopes. Thus, the implementation of lazy envelopes with OSR is further compared to other approaches below.

1. Perform n iterations of the application (*initial-phase*).
2. Deploy the aspect (*deploy*).
3. Perform n iterations of the application (*deployed-phase*).
4. Undeploy the aspect (*undeploy*).
5. Perform n iterations of the application (*undeployed-phase*).
6. Start over at step 2 for m times.

Figure 5.1: Schema of executing benchmarks with dynamic deployment.

5.2.3 Results of Deployment Evaluation

To measure the performance of deployment, the harness of the SPEC JVM98 benchmark suite has been extended to alternately deploy and undeploy an aspect. The modified benchmark harness executes in the scheme presented in Figure 5.1, where the number n of iterations in each phase and the number m of complete cycles are chosen such that the benchmark application respectively the execution environment's deployment subsystem reach a steady-state.

The prototype of envelope-based weaving based on Class Redefinition, presented in Section 2.4 is included in this evaluation. It is called *Envelope-Unaware* and HotSpot 1.5.0 is its reference execution environment.

The extended benchmark measures different properties of the execution environments. First, the time needed for the very first run during initial phase (step 1) is measured. During this run class loading and, if applicable to the AOP implementation, preparation of loaded classes takes place. Further, the time required to perform deployment and undeployment of an aspect, i.e., steps 2 and 4 is measured[2]. The aspect deployed by the harness is presented in AspectJ syntax in Listing 5.1. It advises all calls to methods or constructors within any class in any sub-package of spec.benchmarks. The advice simply increments a counter.

The median of the times for the n benchmark iterations in the initial-phase determines the steady-state performance of the execution environment when no aspects are deployed. The median is used to rule out start-up

[2]To abstract from differences in how to deploy an aspect in different AOP environments, the extended benchmark harness defines an interface Deployer with the methods deploy and undeploy. This interface is implemented for each AOP environment.

```
1 aspect Aspect {
2   static long counter;
3   before() :
4       call(* spec.benchmarks..*.*(..)) ||
5       call(* spec.benchmarks..*.new(..)) {
6     counter++;
7   }
8 }
```

Listing 5.1: Aspect used by the modified SPEC JVM98 benchmark suite.

performance as well as to level out peak runs. The median of the performance figures in the deployed-phase shows how much performance is influenced by advice calls. The median of the figures from the undeployed phase shows whether an aspect that was once deployed is removed completely from the application at undeployment or whether it leaves some footprint.

The invalidation mechanism used to undo inlining of envelopes in this thesis, ensures that invalid code is not executed after deployment. However, re-compilation does not necessarily occur at deployment, but only when the methods whose machine code has been invalidated are executed the next time—similar to the application start-up. To measure the effect of deferred re-compilation, the performance of the first benchmark application run after deployment, respectively undeployment is measured and compared to the steady-state performance in this phase.

For the performance values of deployment, undeployment, and start-up after deployment respectively undeployment, the median of the measured times of all m cycles is calculated.

Tables 5.2 through 5.5 show the results of the benchmarks for the approaches discussed in Section 5.2.1 and the Envelope-Aware prototype with lazy envelopes and OSR.

Table 5.2 shows how many milliseconds are spent for deploying and undeploying the aspect in the modified SPEC JVM98 benchmark harness, presented above. For every benchmark application Envelope-Aware provides the best performance with an average of 3 ms and a maximum of 16 ms. While Steamloom_1 performs well for some benchmark applications, the deployment time goes as high as 4,977 milliseconds and averages out at 154 ms for undeployment and 941 ms for deployment. With AspectWerkz the time for undeploying the aspect even goes up to 20,230 ms in the `jess` application.

Approach	Envelope-Aware		Envelope-Unaware		Aspect-Werkz		JAsCo		Steamloom$_1$	
Operation	depl.	und.	depl.	und.	depl.	und.	depl.	und.	depl.	und.
compress	1	0	63	126	349	442	331	–	102	3
jess	4	2	538	635	16,833	20,230	3,042	–	1,211	188
db	0	0	19	103	215	195	251	–	4	2
javac	14	16	672	1,152	–	–	4,472	–	4,977	805
mpegaudio	2	1	109	236	869	864	1,228	–	55	17
mtrt	2	1	58	152	3,784	4,247	729	–	195	53
jack	1	1	149	380	1,470	1,476	1,446	–	45	16
average	3	3	229	397	3,360	3,922	2,685	–	941	154

Table 5.2: Time in milliseconds for deploying and undeploying the benchmark aspect on SPEC JVM98 benchmarks.

For JAsCo, no undeployment times are presented because it failed during undeployment. As a result, also only *one* deployment was performed and the time required for this is reported in the table. This also made it impossible to determine the steady-state performance of JAsCo's deployment subsystem. Consequently, JAsCo's deployment figures may actually be better than what is presented here. However, the figures cannot be expected to change significantly in the steady-state.

Table 5.3 shows the steady-state performance of the different execution environments. For each AOP implementation the overheads in the initial (*ini.*), deployed (*depl.*) and undeployed (*und.*) phase are shown as compared to their reference virtual machine.

The Envelope-Aware prototype imposes no overhead to the steady-state performance of an application when no aspects are used (initial phase). JAsCo also provides a good steady-state performance in the initial phase with only 1.7% overhead. For the other approaches the overhead in the initial phase reaches from 8.6% (Steamloom$_1$) to 121% (AspectWerkz).

Like in the initial phase, Envelope-Aware imposes no overhead to the undeployed phase; for AspectWerkz the overhead drops from 121% in the initial phase to 58.5%. All other approaches perform slightly worse in the undeployed phase than in the intitial phase. For JAsCo no discussion of the performance in the undeployed phase can be given here since it crashed during undeployment in the benchmark.

Please note that the overhead presented for the deployed phase also contains the overhead for executing the additional advice functionality. The approaches have different schemes of invoking advice, ranging from calling a static method to looking-up a receiver object and calling a virtual method. Consequently, the overhead of different execution environments imposed in the deployed phase is not directly comparable and, thus, not discussed here.

AOP implementations may also imply a degradation of the start-up performance. As presented in Table 5.4, the Envelope-Aware prototype on average performs start-up up to 5.6% slower than the unmodified Jikes RVM. The impact of the other benchmarked approaches on start-up performance, though, is much higher. Of these approaches, Steamloom$_1$'s impact is the smallest with 8.9%; AspectWerkz even imposes 215.9% overhead.

When an aspect is deployed, affected methods are invalidated and recompiled the next time they are invoked. As a consequence, during the first run after deployment just-in-time compilation is performed, similar to the very first run. Table 5.5 shows the performance of the first run after deployment and undeployment compared to the steady-state performance of the deployed respectively undeployed phase. Envelope-Aware performs fairly well with an average overhead of 51.5% after deployment and 56.4% after undeployment. The Envelope-Unaware prototype performs best. Steamloom$_1$ performs roughly the same as Envelope-Aware after deployment and better after undeployment. AspectWerkz and JAsCo perform worse than Envelope-Aware.

5.3 Evaluating Control Flow Quantification

To evaluate the performance of the prototype for efficient control flow quantification, it is compared to AspectWerkz, JAsCo, `ajc` and `abc`. The prototype has actually been developed as part of the Steamloom$_1$ project, thus the performance reported for the prototype also represents Steamloom$_1$'s performance.

The versions of AspectWerkz and JAsCo and the respective reference virtual machines are those mentioned in Section 5.2.1. The static weavers `ajc` 1.5.0 and `abc` 1.1.0 have been used and their compiled code has been executed on the HotSpot 1.5.0 virtual machine. The prototype is implemented as an extension of the Jikes RVM 2.3.1.

The programs compiled with `ajc` or `abc` compilers were not run on the Jikes RVM, which would have given a direct comparison to the implementation of the prototype. This is because both AspectJ compilers produce code which exploits special optimizations of production Java virtual ma-

Approach	Envelope-Aware			Envelope-Unaware			Aspect-Werkz			JAsCo			Steamloom$_1$		
Operation	ini.	depl.	und.	ini.	depl.	und.	ini.	depl.	und.	ini.	depl.	und.	ini.	depl.	und.
compress	-0.6	53.2	2.9	22.7	23.5	23.3	38.4	54.4	5.2	0.1	27.8	–	0.7	23.5	14.4
jess	5.3	24.1	4.0	43.8	47.2	47.0	80.5	86.9	71.4	0.3	23.1	–	7.6	25.6	27.7
db	1.0	0.6	0.4	1.4	1.0	1.9	12.5	12.4	12.5	1.6	1.1	–	0.3	14.5	18.0
javac	4.0	22.6	4.5	29.2	32.0	30.7	–	–	–	4.6	29.3	–	16.7	22.1	24.0
mpegaudio	0.5	-6.7	-8.6	8.1	11.8	9.0	3.4	9.4	4.4	-0.2	10.3	–	20.3	14.0	43.4
mtrt	-10.6	19.0	0.3	3.3	12.9	9.8	585.4	685.5	245.3	1.9	336.0	–	13.1	95.4	97.2
jack	-2.0	-7.6	-8.5	25.6	28.2	27.4	5.9	14.5	12.4	3.6	12.4	–	1.2	3.2	8.9
average	-0.3	15.0	-0.7	19.2	22.3	21.3	121.0	127.2	58.5	1.7	62.8	–	8.6	28.3	33.4

Table 5.3: Steady-state performance overhead in percent measured with the SPEC JVM98 benchmark suite.

Approach	Envelope-Aware	Envelope-Unaware	Aspect-Werkz	JAsCo	Steam-loom$_1$
compress	5.8	23.7	52.4	3.1	5.5
jess	6.7	53.6	432.9	60.5	8.9
db	-0.9	3.1	15.2	1.2	-0.2
javac	3.3	47.6	–	34.3	11.9
mpegaudio	7.9	11.7	33.3	7.4	11.9
mtrt	13.4	26.8	695.3	19.5	14.0
jack	2.9	35.0	66.1	21.2	10.1
average	5.6	28.8	215.9	21.0	8.9

Table 5.4: Start-up performance overhead in percent for the SPEC JVM98 benchmarks.

Approach	Envelope-Aware		Envelope-Unaware		Aspect-Werkz		JAsCo		Steam-loom$_1$	
Operation	depl.	und.	depl.	und.	depl.	und.	depl.	und.	depl.	und.
compress	19.1	74.6	1.1	1.8	5.1	8.5	4.9	–	19.7	22.2
jess	49.2	55.2	25.9	30.4	551.4	718.0	167.6	–	98.5	46.5
db	3.4	0.1	0.0	1.2	1.5	1.3	2.9	–	0.6	-2.5
javac	68.9	54.8	35.9	44.7	–	–	162.7	–	127.3	43.7
mpegaudio	24.2	17.3	4.3	8.1	16.0	16.3	23.7	–	38.9	9.9
mtrt	173.4	160.8	16.0	14.3	40.3	87.5	29.4	–	53.3	51.8
jack	22.1	32.2	12.5	19.1	52.6	53.5	62.4	–	5.2	0.6
average	51.5	56.4	13.7	17.1	111.1	147.5	64.8	–	49.1	24.6

Table 5.5: Start-up performance overhead in percent after deployment and undeployment of the benchmark aspect.

chines; such code is bound to execute untypically slow on other VMs like Jikes RVM.

Three different kinds of evaluations have been performed to evaluate the performance of **cflow**. First, micro-benchmarks are used to compare the overhead imposed on the execution of join points that constitute a control flow respectively of shadows that depend on a control flow. Second, the impact of the **cflow** infrastructures of different approaches on real programs is measured by means of a modified version of the SPEC JVM98 benchmarks. Third, a small set of aspect-oriented benchmarks collected by the `abc` group is used for the comparison to `abc`.

5.3.1 Implementations of cflow in the `ajc` and `abc` Compilers

For the experiment, whose results are presented in Section 5.2, only AOP approaches have been relevant that support dynamic deployment. For following evaluations, also statically weaving approaches with support for **cflow** have been considered, i.e., the compilers `ajc` and `abc`.

The `ajc` compiler employs some optimizations to reduce the compilation time [HH04]. But only few optimizations are applied to make the generated code efficient.

On the contrary, the `abc` compiler [abc, ACH+04, ACH+05a] applies sophisticated static intra- and interprocedural analyses to generate efficient code. The code produced by `abc` is considerably faster [ACH+05b] than that of `ajc` in many cases. As already discussed in Section 4.4, the `abc` compiler provides three different optimization levels, `O1`–`O3`, which differ in the aggressiveness of the applied optimizations. An optimized handling of thread-safety for the **cflow** counters is already performed at the lowest level. This optimization allows the first application thread to quickly access the counters, while all other threads have to use the slow `ThreadLocal` implementation. At the highest optimization level, `abc` performs interprocedural analyses that allow to simplify or even eliminate residual dispatch and infrastructural code.

The better performance of the higher optimization levels comes at the cost of a very time- and memory-consuming compilation. The interprocedural analysis depends on a whole-program analysis that needs to know all possible entry points and the code of all reachable classes. This places Java applications under a closed-world assumption that contradicts Java's dynamic class loading capabilities.

`abc` [ACH+05b, ACH+05a, ACH+04] applies several additional optimizations. Thread-local counters are optimized for the first application thread, so that accessing the counter via a `ThreadLocal` instance is avoided for this thread. This facilitates a very quick retrieval of a counter for single-threaded

```
1  class Base {
2    static int counter;
3    static boolean callDependent = false;
4
5    void dependent() {
6      counter++;
7    }
8    void constituent() {
9      if (callDependent)  dependent();
10   }
11   public static void main(String[] args) {
12     new Base().dependent();
13     /*execute benchmark*/
14   }
15 }
16
17 aspect Aspect {
18   static int counter;
19
20   before() :
21     execution(void Base.dependent()) &&
22     cflow(execution(void Base.constituent())) {
23     counter++;
24   }
25 }
```

Listing 5.2: Micro-measurement harness.

applications. Multi-threaded applications still have to use a ThreadLocal instance for counter management. Code generated by `ajc` always relies on ThreadLocal instances. Moreover, the `abc` compiler performs intra- and inter-procedural analyses to improve the performance of code using **cflow** pointcuts; both optimization types are achieved using static analysis [SdM03]. The `abc` compiler performs the time- and memory-intensive interprocedural analysis only at its highest optimization level.

5.3.2 Micro-Benchmarks

Micro-benchmarks are performed by executing the small application presented in AspectJ syntax[3] in Listing 5.2. The aspect in the benchmark program (lines 17–25) defines a pointcut (lines 21–22) such that the beginning of the method dependent in line 5 is a dependent shadow and control flow is entered at the beginning of the method constituent in line 8 and exited at the end of the same method in line 10.

Two benchmarks are performed by this application at line 13. Both benchmarks consist of a loop performing 100,000 invocations of the method constituent, respectively dependent. Such a high number of iterations is necessary because the operations whose execution is subject to measurement execute in the magnitude of nanoseconds. In both cases, the total time required to perform the invocations is measured. Before doing so, both methods are invoked several times to ensure that they execute at a steady-state during the experiment. Also the time is measured for executing the loop without making any invocation.

To measure the overhead imposed by control flow constitution, the method constituent is called. At the beginning of this method it is recorded that the control flow is entered; the method does not execute anything because the test in line 9 always fails; thus, it is immediately recorded that the control flow is left and the method returns. To measure the overhead for testing whether a control flow is active, the method dependent is called. Since this happens from within the main method, the control flow test always fails. Thus, the advice is not executed and dependent simply increments its counter before returning.

The goal of the micro-benchmark is to measure the worst-case impact of the **cflow** infrastructure of the approaches that are subject of the evaluation. To make sure the worst-case impact is measured, the program in Listing 5.2 is written in a way that prevents different kinds of specific optimizations performed by the approaches being compared. The effects of such optimizations on efficiency are measured by the macro-benchmarks.

- The executing virtual machine can apply optimizations, e.g., when it determines that a called method is empty and the call can thus be omitted. To prevent this, the method dependent and the advice increment a counter (lines 6 and 23).
- The `abc` compiler can determine that a certain control flow will always or never be active when a given join point shadow is executed; in such

[3]For the other environments, the example has been implemented in their respective syntax, or by using appropriate API calls.

Approach	prototype	AspectWerkz	JAsCo	ajc	abc O1 st	abc O3 st	abc O1 mt	abc O3 mt
const.	166.0	7,370.0	134.0	3,839.3	505.3	497.7	1,855.7	1,883.2
check	66.5	4,240.0	$5.6{\cdot}10^6$	1,379.7	686.8	703.6	2,108.1	2,151.8

Table 5.6: Overhead in percent measured by the micro-benchmarks.

cases, the infrastructure for constituting or checking the control flow can be omitted. The method constituent contains a conditional invocation of dependent (line 9) so that abc does not conclude that the method is never called in the control flow of constituent. Similarly, if dependent is not called from outside constituent, the abc compiler would determine that the residual dispatch can be simplified to always invoke the advice. Thus, it would also cause abc to omit tracking of the control flow in constituent. To prevent this, the program in Listing 5.2 makes a call to dependent outside constituent (see line 12).

- The prototype developed in this thesis is configured not to perform inlining to avoid its interprocedural optimization.

To measure the overhead imposed by the **cflow** infrastructure of each AOP implementation the benchmark program is compiled without the aspect with the standard `javac` compiler and executed on the reference execution environment. Results for the micro-measurements are shown in Table 5.6. The row *const.* shows the overhead imposed by the **cflow** infrastructure at constituent join points; the row *check* shows the overhead imposed by the **cflow** infrastructure at dependent shadows.

For `abc`, the benchmark was executed with the weakest and the strongest optimization levels, denoted `O1`, respectively `O3` and in a single- respectively a multi-threaded environment, denoted by `st`, respectively `mt`. Single- and multi-threaded environments are simulated by measuring the performance in the first and in the second thread. It is ensured that only one thread is actually executing and the other one is sleeping during the measurement.

The results in Table 5.6 show that the stack inspection approach implemented by JAsCo imposes an extreme overhead of $5.6{\cdot}10^6\%$ at depending join point shadows. The overhead at constituent join point shadows is small compared to other approaches; yet one would expect that there is no overhead at all, since the control flow does not have to be tracked explicitly at dependent join point shadows. However, other features provided by JAsCo such as the facility for dynamic deployment also have an influence on the benchmark which cannot be eliminated. Due to its bad performance at dependent

join point shadows, JAsCo is excluded from the benchmarks discussed in the following two subsections.

The `abc` compiler produces less overhead than `ajc` in the single-threaded case. In the multi-threaded case, the code produced by both compilers performs badly. Since the benchmark is implemented in a way that `abc` cannot apply optimizations, the results for both optimization levels are about the same. The prototype implementation clearly outperforms `abc` even in the single-threaded case, while ensuring thread-safety.

5.3.3 Benchmarks Based on SPEC JVM98

As already mentioned, the micro-benchmarks prohibit many optimizations and provide a worst-case scenario. Thus, *macro-benchmarks* have also been performed to compare the prototype against `abc` in a more realistic environment that enables `abc`'s interprocedural analysis as well as the prototype's optimizations enabled by method inlining (see Section 4.4.3).

To create such an environment, the SPEC JVM98 benchmark suite has been modified by adding 15 different pointcut-and-advice pairs to each benchmark application. The pointcuts all have the form **execution**(<method1>) && **cflow**(**execution**(<method2>)) and are designed to cover a wide range of different characteristics:

- the frequency of control flow constitutions,
- the ratio of dependent join point shadow executions occurring inside versus outside of the control flow,
- and the number of methods on the call stack between the method that constitutes the control flow (<method2>) and the method that tests whether the control flow is active (<method1>).

The advice attached to each pointcut only increments a counter, so that the overhead introduced by additional functionality is minimal. Semantically equivalent aspects are defined for each approach and are applied as required by the approach. For the prototype, the harness is modified to deploy the aspect before starting the benchmark application; an aspect definition file is passed to AspectWerkz when starting the benchmarks.

For the AspectJ compilers, the benchmark applications have been compiled including the corresponding aspects. The compilation is done at optimization levels 01 and 03 for `abc`. In both cases, the resulting code has been executed on HotSpot.

Approach	prototype	AspectWerkz	`ajc`	`abc O1`	`abc O3`
compress	26.6	1,818.1	575.0	166.7	30.1
jess	-0.6	63.7	14.7	1.7	1.5
db	0.0	0.6	0.7	-0.1	0.0
javac	0.7	10.8	3.8	–	–
mtrt	11.7	72.0	31.4	27.8	22.7
jack	2.1	2.7	0.7	0.4	0.9
average	6.8	328.0	104.4	39.3	11.0

Table 5.7: Overhead measured in percent by the SPEC JVM98 benchmarks.

The `mpegaudio` benchmark is not included in the benchmarks because it is only available as obfuscated class files that could not be processed by most AOP implementations. `abc` could not successfully compile the `javac` benchmark, hence, this benchmark is omitted for `abc`.

The results of running the extended SPEC JVM98 benchmark are presented in Table 5.7. The prototype implementation exhibits the least overhead for all benchmarks but `jack`, even if `abc`'s interprocedural analysis (optimization level `O3`) is activated. This is especially obvious for the `mtrt` benchmark which is the only multi-threaded benchmark application. `ajc` and AspectWerkz exhibit a very unsatisfactory performance. While `ajc` is faster than AspectWerkz, it still inhibits a significant overhead (e.g., 14.7% for `jess` or 31.4% for `mtrt`). `abc` exhibits considerably less overhead than `ajc` already at the lower optimization level. The benefit of the interprocedural optimization employed by `abc-O3` is large for the `compress` benchmark; for all other benchmarks, `abc-O3` performs only slightly faster than `abc-O1`.

5.3.4 `abc` Benchmark Suite

While the SPEC JVM98 benchmark suite is comprised of real applications the benchmark presented in the above sub-section is synthetic in the sense that it uses pointcuts introduced only for the purpose of the benchmark. The aspects used in the benchmark do not contribute of the functionality to the application.

The `abc` team has gathered various benchmarks by collecting AspectJ programs from public sources on the web [DGH+04, ACH+05b] some of which also use **cflow** pointcuts. The prototype has been evaluated by running two applications from this benchmark suite: `figure` and `quicksort`. Both applications are fairly short, each consisting of approximately 150 lines of code.

```
1  pointcut move():
2    call(void FigureElement+.moveBy(...)) ||
3    call(void Point.setX(int)) ||
4    call(void Point.setY(int)) ||
5    call(void Line.setP1(Point)) ||
6    call(void Line.setP2(Point));
7
8  after() returning:
9      move() && !cflowbelow(move()) {
10   Display.needsRepaint();
11 }
```

Listing 5.3: Pointcut-and-advice for figure benchmark.

These benchmarks have only been performed for `ajc`, `abc` and the prototype as the previous benchmarks have already shown that the other approaches exhibit a considerably worse performance. Unfortunately, it has not been possible to run the more complex benchmarks `Law of Demeter`, `Cona`, and `ants` because they are using more advanced constructs unrelated to **cflow** which are not supported by the prototype.

The **cflow**-dependent pointcuts used in both applications are very similar, having the form <staticProperties> && !**cflowbelow**(<pointcut>) and capturing non-recursive entry-points in certain parts of the program. The `figure` application—which simulates a simple figure editor—contains the pointcut-and-advice shown in Listing 5.3. It causes a notification to Display whenever a FigureElement object is changed. However, calling, e.g., a Point's moveBy method during the execution of a Line's moveBy method does not result in a notification due to the employment of the !**cflowbelow**(move()) sub-pointcut. The `quicksort` application collects various statistics on the sorting algorithm. It uses a pointcut-and-advice to select the top-level call to the recursive quicksort method to initialize and display the statistics (Listing 5.4).

The results of running the two benchmark applications on the platforms subject to comparison are presented in Table 5.8. The **cflow** prototype developed in this thesis exhibits the least overhead except for the `figure` application when `abc` is run with the highest optimization level. A closer inspection shows that `abc` is able to completely optimize away all infrastructural code and residues are simplified to always execute advice unconditionally in this benchmark. Since the overhead for notifying the display is negligible, `abc` does not show any overhead compared to the reference execution.

```
1  pointcut sort():
2    call(void QuickSort.quicksort(...));
3  pointcut entry():
4    sort() && !cflowbelow(sort());
5  before() : entry() {
6      Stats.before_entry();
7  }
8
9  after() returning: entry() {
10     Stats.after_entry();
11 }
```

Listing 5.4: Pointcut-and-advice for quicksort benchmark.

Approach	prototype	ajc	abc O1	abc O3	abc O3 exec
figure	72.0	6,620.0	920.0	0.0	330.0
quicksort	3.8	12.7	8.8	10.2	–

Table 5.8: Overhead in percent measured by `figure` and `quicksort` benchmarks.

The fact that `abc` can optimize away all residues in this benchmark also shows that the interprocedural analysis is very effective if a pointcut has many shadows. In this example, there are multiple call sites of the move methods and `abc` can analyze each of them separately and determine that some of them always and others never are executed in the specified control flow.

If an **execution** pointcut is used instead of **call**, `abc`'s interprocedural optimizations are disabled. Using **call** rather than **execution** pointcuts, results in more shadows in the program (at every call site). If the **execution** pointcut is used instead there are less shadows in the program (only the method bodies) and these shadows may be executed both inside *and* outside the control flow, so that a dynamic check is necessary.

To make the effect of using **execution** rather than **call** pointcut designators on the measurement explicit, a version of `figure` that uses **execution** instead of **call** compiled with `abc-O3` has also been included in the benchmark (last column in Table 5.8). The semantics of the aspect does not change: the display is still notified every time a FigureElement is modified and repeated notifications are avoided. The prototype's optimizations are not vulnerable to such changes in code. Jikes' inlining optimization generates a separate

(inlined) version of a method body at every hot call site, so that the interprocedural optimizations can reason about these join points in the same differentiated way as if **call** pointcuts have been used.

Chapter 6

Related Work

Throughout this thesis its contributions are directly compared with related approaches. These approaches are discussed in detail along with the comparison. However, there is some additional related work to which a quantitative comparison, as performed in the previous chapters, is not applicable. To give a comprehensive overview of the field of AO language implementations, these related approaches are presented in this chapter.

6.1 Virtual Join Points

There are two related approaches that frame join points in terms of virtual dispatch. The more recent approach extends the prototype- and delegation-based execution model of object-oriented programs. The second one is based on reflection.

6.1.1 Prototype- and Delegation-Based Execution Model

Haupt and Schippers [HS07] present a *machine model for aspect-oriented programming*. Just like the execution semantics presented in Chapter 2, their model evolves around the notion of virtual join points. In fact, some of their core ideas have common roots [BHM06, HBMO03] with the semantics discussed in this thesis.

In a *prototype- and delegation*-based object-oriented execution model, method calls and field accesses are realized as *messages* that are sent to objects. New objects are created by "cloning" another prototypical object whereby the new object refers to the prototype by a so-called *parent* reference. Whenever a message is sent to an object, the object may either choose

to handle the message itself or to delegate it to its parent, whereas the active object, called *self*, remains the one the message has originally been sent to.

The machine model in [HS07] is a seamless extension of such a delegation- and prototype-based execution model such that objects are not referenced directly, but through a *proxy* object; the proxy does not handle any messages, but delegates all received messages to its parent which is the actual object. Potential join points in this model are the reception of messages that are sent to objects, i.e., in the extended model, messages sent to proxy objects. As message reception is handled by virtual methods in the underlying execution model join points also are virtual.

Adding advice to a join point is realized by inserting an additional proxy object in the delegation chain of parent references between the first proxy and the actual object. If the advice should, e.g., be executed before the execution of a method m, an object is inserted that understands the message m; when it receives the message, it first executes the advice and afterwards *re-sends* the message to its parent.

The execution model is further extended by making the parent reference of each object a function. Thus, the function can return a different parent proxy depending on the current runtime context, e.g., depending on the current thread. This is similar to the approach of virtual join points discussed in Chapter 2 of this thesis and its realization in FIAL presented in Section 3.3.3. In both cases, a function is evaluated to determine the applicability of advice and potentially the applicability of multiple advice is determined by just one function. In the model from [HS07], sequential advice executions are realized by means of nested message sends, whereas the approach in this thesis is to generate code containing all advice in a sequence.

In the model of Haupt and Schippers, aspect deployment is realized by inserting a proxy object into the delegation chain. This is similar to the work of this thesis, as the delegation chain embodies the dispatch function. However, in the prototypes developed in this thesis, adding new advice and thereby modifying the dispatch function, also means to re-generate the code of affected join point shadows. This is because the dispatch function gets integrated into the code generated for a join point shadow.

While the results of the model presented in [HS07] and the work presented in this thesis ultimately are very similar, the approaches tackle virtual join points from different perspectives. In this work, the Java execution model is extended to seamlessly support AOP; the extension is motivated by the realization and optimization of virtual methods in the Java virtual machine. In contrast, Haupt and Schippers base their approach on a delegation-based execution model. So far, they have not discussed optimizations in the execution environment that are specifically designed for their extended execution

semantics. While their approach for the most part uses existing concepts of the underlying execution model which are already being optimized, they do not provide a discussion of their extension's impact on runtime performance, but focus on the formal semantics of their model.

6.1.2 Reflection-Based Execution Model

AspectS [Hir03, Aspc] is a framework implemented on top of the Squeak Smalltalk environment [IKM+97, Squ]. AOP support is implemented using only Smalltalk's reflective capabilities. In AspectS, message receptions are supported as join points, which is equivalent to the approach presented in this thesis and the approach of Haupt and Schippers, since method invocations as well as member variable accesses in terms of getters and setters are implemented using messages.

All weaving in AspectS takes place at the meta-level of message sending. When a message implementation is decorated with advice functionality, its entry in the corresponding class' method dictionary is modified to instead reference a *method wrapper*. The wrapper invokes advice functionality and yields control to the original implementation, or to subsequent wrappers, if multiple advice apply.

The application of dispatch modifications to meta-level structures as met in AspectS augments the dispatching logic in the form of Smalltalk's original look-up mechanism for late-bound methods. The lookup mechanism itself is altered, along with the data that it operates on. When several wrappers are attached to a join point, the so-called *wrapper chain* must be iterated over, checking for each particular wrapper's applicability using conditional logic contained in the wrappers. Aggressive optimizations as usually found in sophisticated OO language implementations are not applicable due to the implementation of the iteration over the wrapper chain as a method with full computational power.

6.2 Meta-Models for Aspect-Oriented Concepts

Related frameworks for fast prototypical implementation of new aspect-oriented concepts are realized in different ways. The most recent approaches extend the Java language and are based on interpretation or on reflection. An older approach is does not support a real-world base language but allows to model new AO concepts by means of an operational semantics.

6.2.1 Metaspin and JAMI

In [BMN+06], a meta-model to capture the semantics of features in aspect-oriented languages is defined. The meta-model is implemented as an interpreter in the Smalltalk language, called *Metaspin*, whereby each computational step is represented as a closure and can be a join point. The meta-model and the Metaspin interpreter are suitable to experiment with new AO language features. However, this approach only targets language design and no connection to efficient execution environments is considered.

JAMI is a Java-based implementation of the same meta-model. The case study of implementing the decorator enforcing language extension presented in Section 3.6 has also been performed by the implementers of JAMI [JAM07, HBA08]. Both implementations are very similar, although the JAMI and FIAL meta-models have different objectives. JAMI's meta-model is based on reflection, while LIAM is targeted towards weaving execution environments. The JAMI-based implementation altogether has 150 lines of code while the FIAL-based implementation comprises about 200 lines of code. This smaller footprint is caused by additional re-use opportunities due to the use of reflection in JAMI.

While Metaspin and JAMI are more flexible than FIAL they are on par with respect to expressiveness. However, optimized implementations are beyond the scope of Metaspin and JAMI.

6.2.2 Reflex

The Reflex project [Tan06, TN04] provides an extensible kernel for aspect-oriented programming, based on reflection. It is implemented as a Java 5 bytecode instrumenting agent which inserts hooks into classes at load-time. Reflex provides support for structural as well as behavioral crosscutting, i.e., the pointcut-and-advice flavor; however, in the context of this thesis only behavioral crosscutting is relevant. Here, Reflex provides a meta-model for defining behavioral crosscutting which mainly consists of three parts.

- So-called *hooksets* are expressions specifying sets of *hooks*, the equivalent of join point shadows.
- *Metaobjects* are comparable to aspect classes, i.e., they are Java objects which may be implicitly instantiated and which have methods that are invoked as advice functionality.
- *Links* associate hooksets and metaobjects.

A link can also have so-called *link attributes* which, e.g., control the implicit instantiation of metaobjects or define activation conditions. An activation condition is the equivalent of LIAM's dynamic property expressions. Activation conditions in Reflex are defined in terms of a code block. Hence, they are not specified declaratively as in LIAM where primitive conditions are modeled as concrete sub-types of DynamicProperty.

Thus, the Reflex meta-model is not suitable for the implementation of an optimizing execution environment. In fact, Reflex rather aims at providing a comfortable workbench for experimenting with issues of aspect composition like ordering and nesting of aspects. The LIAM meta-model developed in this thesis also supports the specification of aspect interactions in terms of the ScheduleInfo meta-entity. Only a few special cases have been implemented in this thesis, and a more elaborate model of aspect interactions in LIAM is subject to future work.

6.2.3 Aspect Sand Box

In [WKD04] and [MKD02], the Aspect Sand Box is presented as a framework based on operational semantics models of aspect-oriented programming. This framework is intended for prototyping and studying alternative AOP semantics and implementation techniques. The Aspect Sand Box is interpreter-based and implemented in Scheme. By "implementation techniques" the authors refer to partial evaluation; while this is an important optimization in AOP language implementations it is limited to compile-time optimizations. Implementing optimizations in the execution environment, as is the approach of this thesis opens up additional opportunities for optimizations that cannot be prototyped with the Aspect Sand Box. Additionally, FIAL supports standard Java as base language whereas the Aspect Sand Box only supports a simple, non-real-world object-oriented language. Thus, no real-world projects can be used as base programs to evaluate new language concepts in the Aspect Sand Box.

6.3 Intermediate Languages and Execution Environments

Recently, the potential of providing optimizations for aspect-oriented language constructs in the virtual machine has also been addressed by other projects. Currently, there are two projects related to the work presented in this thesis with a reasonable VM integration.

6.3.1 Nu

The Nu project [DR08, RDNH06] aims at providing an interface between compilers and execution environments, just like the FIAL framework developed in this thesis. For this purpose, two new instructions are added to the intermediate language of the base language—in Nu, as in this thesis, this is Java. The new instructions are *bind* and *remove* which correspond to deploying respectively undeploying pointcut-and-advice. The pointcut-and-advice definitions themselves are realized as a composition of Java objects, like in this thesis. When bind is executed, it expects a *pattern* object and a *delegate* object on the operand stack. The pattern argument corresponds to the pointcut part, however, Nu's model is restricted to patterns that can be statically evaluated, i.e., fully evaluate to join point shadows. The delegate argument refers to a method that implements the advice functionality. Executing the bind instruction returns a handle that can be used to remove the binding later.

The Java HotSpot [Hot08] virtual machine, i.e., an industrial strength VM, is extended in the Nu project to execute the new intermediate instructions. HotSpot has three modes of executing methods of an application: they may be interpreted, JIT compiled without optimizations and JIT compiled with optimizations. In any case, each method has a stub that is executed before the method is itself executed. The stub tests whether the method has been compiled and jumps to the compiled code in that case; otherwise, the method is interpreted. Nu applies the advice dispatch mechanism within this stub, i.e., when the stub gets executed it looks up and invokes applicable advice. Furthermore, Nu implements a caching strategy, so that the look-up does not have to be performed at every method invocation.

The aspect model of Nu is less detailed than the LIAM meta-model developed in this thesis. Nu patterns cannot express conditions on dynamic properties of join points like the required type of the receiver object or **cflow**. But the effect of these conditions can be emulated. Binding values from the join point's context for use by the delegate can also not be specified declaratively in the current model of Nu.

A possible realization of **cflow** with the means of Nu is discussed in [DR08]. Consider that constituentEnter and constituentExit are patterns of join point shadows that constitute the control flow—in the Nu's join point model, it is specified as part of the pattern if the start or the end of a join point is selected—and dependent is a pattern of join point shadows that depend on the control flow. The pattern constituentEnter is bound to a delegate that, in turn, binds dependent to the actual advice; similarly, constituentExit is bound to a delegate that removes the binding of dependent to the advice.

While referring to dynamic properties of join points, e.g., by **cflow**, can be emulated in Nu, the logic is implemented in the delegates and not explicitly present in the program's intermediate representation. In this thesis, it is argued that only such an explicit representation facilitates optimizations like the one for **cflow** presented in Section 4.4.

Because of the tight virtual machine integration, Nu, like the prototypes presented in Chapter 4, exhibits a very good performance. The support for dynamic deployment comes at an average cost of 1.5% overhead whereas no overhead is imposed at all by the prototype for optimized dynamic deployment developed in this thesis. The performance of executing the bind and remove instructions has been measured in [DR08] by a small benchmark that binds respectively removes a delegate to one specific method. On average the required time is 11 μs for the bind operation and 3.4 μs for the remove operation. While this result is very good, it is not directly comparable to the results presented in Section 5.2 where deployment affected hundreds join point shadows at once. Furthermore, Nu's current approach of using method stubs for advice dispatch only works in the interpreter mode of HotSpot.

6.3.2 Lightweight VM Support for AspectJ

Golbeck et al. [GDN+08] have implemented extensions to the Jikes Research Virtual Machine to provide dedicated support to programs compiled with the AspectJ compiler. Their extended virtual machine expects that the AspectJ compiler has woven the aspects into the application code, and that the woven code is annotated with meta-information so that it can be identified. Any Java virtual machine can simply execute the woven bytecode, thus, it supports all features of AspectJ. For some features, however, the extended VM offers special support. Woven code that represents such a feature can be identified by the meta-information and optimized machine code can be generated for it.

This is in contrast to the approach followed in this thesis; here it is expected that aspects are not woven at all, but the complete weaving is performed by the execution environment. In order to also support features where no special optimizations are implemented, the work presented in this thesis employs a default code generation strategy.

Golbeck et al. claim that their approach may combine optimizations driven by static analyses as performed by the `abc` compiler with dynamic optimizations performed by the virtual machine. Such a combination is indeed promising as complex static analyses are too expensive to be applied at runtime. However, as of this writing, this possibility is only theoretical, as the `abc` compiler does not generate the appropriate meta-information.

The lightweight AspectJ virtual machine has succeeded in providing minimally invasive support for AspectJ in so far that arbitrary code compiled with the standard AspectJ compiler can be executed. Hence the compiled code does not have any dependency to the virtual machine. This is similar to the work presented in this thesis. The importer-based approach of integrating AO languages in Section 3.4 also facilitates to execute applications that are compiled with the standard AspectJ compiler. However, the work presented here is not tailored just towards one specific AO language, but aims at supporting multiple languages.

The VM optimizations for AO concepts presented in [GDN+08] target at retrieval of aspect instances, advice invocation, **cflow** and **around** advice execution. The **cflow** optimization is a partial re-implementation of the one presented in Section 4.4. It is assumed that the other optimizations can also be realized as special code generation strategies in FIAL-based execution environments; the implementation is subject to future work.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis the semantics of join points, the central concept of aspect-oriented programming languages, is conceived as a successor of virtual methods. Virtual methods are a well established and well understood concept. Consequently, highly effective optimizations have been developed for virtual methods and are implemented in all major virtual machines for object-oriented programming languages. The investigation of virtual methods in this thesis has shown that the optimizations are facilitated by the way virtual method dispatch is represented in the intermediate code. The dispatch is provided in a declarative way, making explicit on which runtime context it depends.

This insight has driven the development of an explicit and declarative meta-model for aspect-oriented constructs that makes it possible to preserve a first-class representation of these constructs even after compilation enabling optimizations comparable of those for virtual methods. The developed meta-model, LIAM, covers the core concepts of aspect-oriented languages that have been identified by investigating four different, but widely known AO languages: AspectJ, CaesarJ, Compose* and JAsCo. LIAM is extensible such that the concrete AO concepts of the investigated languages are realized as concretizations of the abstract meta-entities of LIAM. This has shown that LIAM is indeed sufficient to express aspects written in these languages. For AspectJ, Compose* and JAsCo, even automatic translators have been developed to generate LIAM models from aspects written in these languages.

A framework, FIAL, has been developed providing a generic implementation of execution environments for AO programs with LIAM models as intermediate representation of aspects. The framework also embodies the

concept of join points being virtual and maintains join point shadows, i.e., virtual invocation sites of join points, as structural entities of the executed program. Hereby, join point shadows have a dispatch function which is modified by dynamic aspect deployment. FIAL manages common tasks like aspect deployment and handling the addition of join point shadows during class loading. Concrete execution environments for AO languages can be realized as instantiation of the FIAL framework and benefit from its generic implementations. When join point shadows change due to aspect deployment, the concrete execution environment is called-back by FIAL in order to re-create the code of the modified join point shadows. Logic for, e.g., ordering advice is not required from the concrete execution environment.

In a case study, a new aspect-oriented programming language has been designed and its concepts have been realized as concretizations of LIAM meta-entities. This case study has shown that designing new languages with LIAM is a feasible approach. Since the semantics of the new concepts are implemented as pure Java code and don't have to consider, e.g., bytecode transformations, designing the concepts is also possible for designers without in-depth knowledge of bytecode and the Java virtual machine. In the case study, it is also discussed how dedicated optimizations in the virtual machine by means of a FIAL instantiation can improve the performance of executing the new concepts. This, in turn, can be realized without considering the language design. Because the LIAM entity implementations are already provided, the language implementer has an executable specification of the concepts' semantics.

A reference implementation (ERIN) of FIAL has been developed that uses the envelope-based weaving technique, also developed in this thesis, in order to realize the re-creation of join point shadows. This weaving technique imposes little complexity as the base program code is only minimally rewritten. Thus, envelope-based weaving has proven to be suitable for weaving at runtime.

Highly efficient, sophisticated optimizations as developed in this thesis, are facilitated by the declarative representation of aspect-oriented concepts as provided by the LIAM meta-model. To start with, the event of aspect deployment is an explicit event that can be handled by the execution environment. An optimization for dynamic deployment has been developed, that is based on the observed commonalities of virtual join points and virtual methods.

Envelope methods, which are the embodiment of join point shadows, are inlined when no advice is attached to them. Dynamic aspect deployment is an explicit event which potentially invalidates the inlining decision. The speculative optimization mechanisms that are applied to virtual methods are

adapted such that they also apply to envelope methods. As a consequence, the same invalidation mechanisms that are applied at class loading are used to revert the optimization of envelopes at dynamic aspect deployment. In the prototype developed in this thesis, the infrastructure required to facilitate reverting of inlining decisions does not pose any overhead. Deployment itself is performed within a few milliseconds, in a set of benchmarks of reasonable size, where execution environments with comparable support took up to twenty seconds.

Another optimization has been developed that is facilitated because a first-class representation of aspect-oriented concepts is available to the virtual machine. This optimization is for the **cflow** pointcut designator of AspectJ and comparable constructs in other AO languages. The optimization is also motivated by an established optimization technique for speculative optimizations. Furthermore, it is facilitated by engaging with different components of the virtual machine. Besides the JIT compiler, the call stack layout and the thread-switching implementation have been jointly modified.

The performance impact of the actions at join points constituting the relevant control flow as well as the performance of testing for an active control flow has been measured. In the worst case, the implementation presented in this thesis, imposes 166% overhead to a method call which constitutes as a control flow and 66% overhead to a method call join point shadow that tests for a control flow. In comparison, the other researched implementations, at least impose an overhead of 1,883% during control flow constitution and 1,3780% during control flow testing.

In real-world applications, some static optimizations are highly effective; often, they can even completely optimize away the infrastructure required for control flow maintenance and the control flow tests. Furthermore, `abc` applies an optimization beneficial to single-threaded programs. However, these optimizations come at the cost of a significantly increased compile-time and require a whole-world analysis. That means that all classes must be available at compile-time and dynamic class loading potentially leads to wrong behavior in the presence of the optimizations.

While the prototype developed in this thesis still supports dynamic class loading and does not require a costly static analysis, it is also usually more efficient than `abc` in real-world applications. Except for one of six benchmarks, the benchmarks executed by the prototype of this thesis always performed better than executing the benchmarks compiled even with `abc`'s most aggressive optimizations.

Since no benchmarks of reasonable size are currently available for dynamic aspect-oriented execution environments, a new benchmark approach has been developed in this thesis. The approach is based on extending an

established object-oriented benchmark, in the case of this thesis the SPEC JVM98 benchmark suite. This benchmark acts as the base program and aspects are deployed to it that excessively use the AO concept that is supposed to be benchmarked. The developed benchmarks have been used to evaluate the optimizations developed in this thesis from different perspectives. This has shown that the optimizations are superior to comparable implementations in nearly every respect.

7.2 Future Work

The FIAL framework and virtual machine optimizations presented in this thesis are still subject to further improvements; however, the implementation as described here is fully functional and available for download from `http://www.alia4j.org`. Some future improvements as well as some future projects that will build upon the work presented in this thesis are discussed in the following sections.

7.2.1 Optimizing the Dispatch Function

It has already been discussed in Section 3.3.3 that a dispatch function combining the entire residual dispatching logic at a join point shadow opens up the opportunity for joint optimization. Optimizations of the evaluation order of dynamic properties involved in the dispatch function are already implemented in FIAL [SBM08]. The current implementation in FIAL furthermore eliminates redundant evaluations and initial experiments have shown that the evaluation order of dynamic properties has a significant impact on the efficiency of the dispatch function's evaluation: to optimize the average evaluation time, it is, in the majority of cases, beneficial to first evaluate those dynamic properties that are the least costly.

These findings have resulted from simulations that estimate the dispatch function's average evaluation time for every possible function with up to five different dynamic properties and up to six operators. However, the effect of such optimizations has not been studied in the context of real-world applications: the evaluation costs for dynamic properties have only been approximated and it has been assumed that each dynamic property will be satisfied in exactly 50% of the cases. Furthermore, all dynamic properties have been assumed to be statistically independent.

It remains for future work to study the space of dispatch functions and dynamic properties in practice, aiming at a more realistic model of this space. This will include a detailed cost model for dynamic properties. Another

important part of this model is to identify causal dependencies between dynamic properties. Once an appropriate model is found to be used by the FIAL implementation for determining the evaluation strategy of dispatch functions, the LIAM entities modeling dynamic properties will be enhanced to reflect this model. Currently, the abstract class DynamicProperty has a method returning the expected evaluation costs, but additional methods, e.g., for determining causal dependencies, may be added.

Furthermore, it is also future work to apply adaptive optimizations to dispatch functions. The evaluation of dispatch functions will be profiled such that the model discussed above is refined according to the actual execution characteristics of the application. E.g., the actual costs for evaluating a dynamic property can be measured as well as the actual probability of a dynamic property to succeed. This information can be considered by the virtual machine and it may decide that refining the evaluation strategy will lead to a reduced evaluation time in a future execution. The expected gain has to be weighed against the cost for re-compiling the join point shadow that contains the dispatch function. If the re-compilation time is less than the expected gain, the adaptive optimization is performed, similar to adaptive re-compilation of methods [AFG+00].

The evaluation of temporal predicates over events in the program execution, as, e.g., performed by monitor-oriented programming and runtime verification [BGHS04], can also benefit from optimizations of the join point dispatch. The evaluation of such predicates is usually realized by an automaton whereby the single events cause a state transition in the automaton. In [ATdM07, BHL+07] the AspectJ language is extended so-called *trace monitors* which match the program execution with temporal predicates. If the predicate matches, extra code is executed, like throwing an exception when the program reaches an erroneous state. The authors claim that many runtime verification concerns can be realized with their trace monitors. In their implementation, AspectJ pointcuts are basic events and their associated advice it to perform state transition in the underlying automaton. The pointcuts themselves can already refer to dynamic properties and, hence, benefit from an optimized join point dispatch. Furthermore, state transition depends on the current state. In the language implementation approach presented in this thesis, the dependency on the current state can be explicitly expressed as a dynamic property to which the pointcut refers. Thereby the test for the current state can also be subject to virtual machine optimizations like the optimized **cflow** implementation presented in Section 4.4.

7.2.2 Virtual Machine as FIAL Instantiation

The implementation of a virtual machine as an instantiation of FIAL that includes all optimizations discussed in this thesis is work in progress. This virtual machine, called STEAMLOOM^{ALIA}, is the successor of the Steamloom$_1$ project and likewise is an extension of the Jikes RVM.

A goal of the STEAMLOOM^{ALIA} virtual machine is to integrate the AOP support into the VM as seamlessly as possible. This means, for example, aspects should be subject to Java's dynamic class loading semantics: an aspect class is only loaded when a join point is reached that is affected by the aspect. Similarly, the virtual machine only JIT compiles bytecode when it is actually executed. Hence, also the code for join point shadows is only compiled when they are reached. Join point shadows are, as in the envelope-based weaving approach, entities on their own, similar to methods. However, by their deep integration into the virtual machine, they have differences to methods, e.g., their resolution is performed differently.

Currently most of the LIAM entities are realized through the default code generation strategy. Only to some Context and some InstantiationStrategy entities special bytecode generation strategies are applied, like directly reading the value from the local variables instead of using reflection in terms of the Java VM Tools Interface. It is future work to also implement specific code generation strategies for other LIAM entities whereby the strategies are not restricted to generating bytecode—because of the VM integration, it is also possible to directly generate machine code. Special code generation strategies will comprise the **cflow** optimization presented in Section 4.4, and optimized aspect instance retrieval as presented in [GDN+08, Hau06].

The adaptive optimization of dispatch functions discussed in the previous section will also be realized as part of the STEAMLOOM^{ALIA} virtual machine. Furthermore, it will be investigated whether the adaptive optimization strategy of Jikes has to be modified to better cope with dynamic deployment. As presented in Section 5.2, dynamic deployment requires subsequent re-compilation of affected methods, which leads to a slow-down after deployment. It will be studied which recompilation strategy is most effective after deployment, i.e., at which optimization level methods should be re-compiled. Caching strategies for compiled code will also be investigated that may be beneficial for situations when the same aspect gets deployed and undeployed frequently. Finally, heuristics will be investigated for deciding whether to inline join point shadows or not. Since only the bytecode of join point shadows is modified at deployment, no other methods have to be re-compiled when affected join point shadows are not inlined. Hence, in

a context where deployment occurs frequently, it may be beneficial, not to inline all envelopes.

7.2.3 Static Crosscutting

Static crosscutting in AspectJ refers to the ability of an aspect to declare new members or interfaces of classes in the base program. In this thesis, static crosscutting has been left out of consideration, support for it remains future work. The static crosscutting features in the AspectJ source code can be integrated with FIAL similarly to the PA features. When an AspectJ program that makes use of static crosscutting definitions is compiled with `ajc` whereby weaving is disabled, the base classes of the program are not transformed, but annotations representing the static crosscutting definitions are inserted into aspect classes, similar to pointcuts and advice. Unlike the pointcut-and-advice related annotations, the annotations for static crosscutting definitions, unfortunately, are not officially documented. There are two conceivable approaches that will be considered in future work based on `ajc`'s intermediate representation.

Features like adding interfaces to a class require to re-write the bytecode of the class. Since the re-writing depends on the aspect definition and on the source language the aspects are written in, the FIAL framework extended to give an importer the chance to re-write classes before they are loaded. This can be used by an importer may re-write classes in order to realize the static crosscutting; this is similar to what the AspectJ load-time weaver does. In this approach, the importer can perform arbitrary re-writing of bytecode. It is future work, to research opportunities to provide support in the FIAL framework for bytecode transformations that weave static crosscutting.

A second approach, which, however, is only possible for the addition of members to classes, is to map this problem to that of supporting pointcut-and-advice [HS07, BMN+06]. Besides the annotations that serve as intermediate representation of the static crosscutting definitions, the `ajc` compiler also generates code reflecting the elements to be added in the aspect class, similar to advice methods. Code that depends on the added members, however, accesses them as if they were declared in the class to which they are added by the aspect.

To support this kind of static crosscutting, i.e., the addition of methods and fields to classes, advice units can be derived for each such method or field declared in an aspect. The join point set selects all accesses to the member and the advice action is to access the member definition in the aspect class. Furthermore, a schedule info entity is attached to the advice unit declaring that the original join point action is not executed, as the member does not

exist in the original target class. Such a facility is already provided in the ScheduleInfo meta-entity as it is required by the Compose* importer presented in Section 3.4.

7.2.4 IDE Support for Programs Executed on FIAL

Currently, an extension to the AspectJ Development Tools (AJDT) is being developed that allows the use of all its features, when compiling AspectJ applications for execution on a FIAL-based execution environment. As presented in Section 3.4 it is already possible to use the AJDT for this purpose; merely additional settings have to be made to stop the `ajc` compiler from weaving the aspects itself. However, only when weaving is performed the compiler generates the so-called *abstract structure model* which is used by the AJDT to visualize the crosscutting structure of the program. An example of the model's visualization is the "Cross References" view that shows a tree structure of advice and their advised join point shadows. These cross references are also navigable, i.e., it is possible to navigate from an advised join point shadow to the corresponding advice and vice versa.

To re-establish such features, a project is currently in progress that extends the AJDT with a new component that builds the abstract structure model from the aspects' intermediate representation. This component is realized as an Eclipse builder—the AJDT is a plugin to the Eclipse IDE [Ecl]—which is always executed after the AJDT's `ajc` compiler. The builder is being implemented as an instantiation of the FIAL framework and uses the AspectJ importer developed in this thesis to transform `ajc`'s intermediate representation to LIAM entities.

All classes in the build path of the AJDT project, i.e., all the code developed in that project, are traversed by the builder and join point shadows are identified and made known to FIAL which attaches the advice units that are defined by the LIAM model provided by the AspectJ importer. Afterwards, the builder investigates all join point shadows to check if advice are attached. For those join point shadows where advice are attached, the builder establishes a link in AJDT's abstract structure model.

To support this work, the LIAM meta-entities are extended to also store the location in the source code, where they are defined. This is similar to the debug information present in Java bytecode. This debug information facilitates to recover the file name and line number whose compilation has lead to a bytecode instruction, respectively in the case of LIAM to a model entity. The builder uses this information to establish links between source locations, as is stipulated by the AJDT abstract structure model.

As a result of this project, it will be completely transparent to the user if aspects are woven by the `ajc` compiler or if they are woven at runtime by a FIAL-based execution environment.

Scientific Career

since January 2009
Universiteit Twente
Assistant Professor on Software Composition in the Software Engineering Group of Prof. Mehmet Akşit

July 2008 – January 2009
Technische Universität Darmstadt
Post doctorate researcher in the Software Technology Group of Prof. Mira Mezini

July 2003 – June 2008
Technische Universität Darmstadt
PhD assistant in the Software Technology Group of Prof. Mira Mezini

October 1998 – June 2003
Technische Universität Darmstadt
Studies in Computer Science. Graduated as Diplom-Informatiker
(comparable to master degree in computer science)

Bibliography

[AAB+05] Bowen Alpern, Steve Augart, Steve M. Blackburn, Maria Butrico, Antony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Ngo, and Vivek Sarkar. The Jikes Virtual Machine Research Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44, 2005.

[AAC+99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2005.

[abc] Homepage of AspectBench Compiler for AspectJ. `http://abc.comlab.ox.ac.uk/`.

[ACF+01] Bowen Alpern, Anthony Cocchi, Stephen J. Fink, David Grove, and Derek Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[ACH+04] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor,

Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Building the abc AspectJ Compiler with Polyglot and Soot. Technical Report abc-2004-4, aspectbench.org, 2004.

[ACH+05a] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An Extensible AspectJ Compiler. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.

[ACH+05b] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, 2005.

[ACL+99] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño—A Compiler-Supported Java Virtual Machine for Servers. In *Proceedings of Workshop on Compiler Support for System Software*, 1999.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optmization*, 2000.

[AFG+05] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. In *Proceedings of the Institute of Electrical and Electronics Engineers*. IEEE, 2005.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of CaesarJ. *Transactions on International Conference on Aspect-Oriented Software Development I*, LNCS 3880, 2006.

[AHR02] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[AJD] Homepage of AspectJ Development Tools. `http://eclipse.org/ajdt/`.

[AR02] Matthew Arnold and Barbara G. Ryder. Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 2002.

[Aspa] Homepage of AspectJ. `http://www.aspectj.org`.

[Aspb] The AspectJ™Development Environment Guide. `http://www.eclipse.org/aspectj/doc/released/devguide/index.html`.

[Aspc] Homepage of AspectS. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.

[Aspd] Homepage of AspectWerkz. `http://aspectwerkz.codehaus.org/`.

[Asp08] The AspectJ Programming Guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`, 2008.

[ATBC+03] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, and Tatiana Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7, 2003.

[ATdM07] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making Trace Monitors Feasible. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2007.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35, 2003.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing Multiple Concerns Using Composition Filters. Communications of the ACM, 2001. Available at trese.cs.utwente.nl/composition_filters/.

[BADM06] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. In *Proceedings of the Conference on*

Object-Oriented Programming, Systems, Languages, and Applications, 2006.

[BAT] Homepage of BAT. `http://www.st.informatik.tu-darmstadt.de/BAT`.

[BCE] *Byte Code Engineering Library.*

[BFTY04] Ohad Barzilay, Yishai A. Feldman, Shmuel Tyszberowicz, and Amiram Yehudai. Call and Execution Semantics in AspectJ. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages*, 2004.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*. Springer, 2004.

[BH06] Christoph Bockisch and Michael Haupt. Taxonomy of Implementation Techniques in Relation to the Aspects of the Meta-Model. Technical Report AOSD-Europe-TUD-6, Technische Universität Darmstadt, 2006.

[BHL+05] Johan Brichau, Michael Haupt, Nicholas Leidenfrost, Awais Rashid, Lodewijk Bergmans, Tom Staijen, Istvan Anis Charfi, Christoph Bockisch, Ivica Aracic, Vaidas Gasiunas, Klaus Ostermann, Lionel Seinturier, Renaud Pawlak, Mario Südholt, Jacques Noyé, Davy Suvee, Maja D'Hondt, Peter Ebraert, Wim Vanderperren, Shiu Lun Tsang Monica Pinto, Lidia Fuentes, Eddy Truyen, Adriaan Moors, Maarten Bynens, Wouter Joosen, Shmuel Katz, Adrian Coyler, Helen Hawkins, Andy Clement, and Olaf Spinczyk. Report describing survey of aspect languages and models. Technical Report AOSD-Europe-VUB-01, Vrije Universiteit Brussel, 2005.

[BHL+07] Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. In Oleg Sokolsky and Serdar Tasiran, editors, *Proceedings of the International Workshop on Runtime Verification*, volume 4839 of *Lecture Notes in Computer Science*. Springer, 2007.

[BHM06] Christoph Bockisch, Michael Haupt, and Mira Mezini. Dynamic Virtual Join Point Dispatch. In *Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006.

[BHMMk05] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based Weaving for Faster Aspect Compilers. In *Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. GI, 2005.

[BHMO04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2004.

[BJC08] Christoph Bockisch, Andrew Jackson, and David Cousins. Second Review of Atelier Content and Performance. Technical Report AOSD-Europe-TUD-10, Technische Universität Darmstadt, 2008.

[BKH+06] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient Control Flow Quantification. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[BM07] Christoph Bockisch and Mira Mezini. A Flexible Architecture For Pointcut-Advice Language Implementations. In *Proceedings of the Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*. ACM Press, 2007.

[BMGFr07] Christoph Bockisch, Mira Mezini, Kris Gybels, and Johan Fabry. Initial Definition of the Aspect Language Reference Model and Prototype Implementation Adhering to the Language Implementation Toolkit Architecture. Technical Report AOSD-Europe-TUD-7, Technische Universität Darmstadt, 2007.

[BMH+07] Christoph Bockisch, Mira Mezini, Wilke Havinga, Lodewijk Bergmans, and Kris Gybels. Reference Model Implementation. Technical Report AOSD-Europe-TUD-8, Technische Universität Darmstadt, 2007.

[BMN+06] Johan Brichau, Mira Mezini, Jacques Noyé, Wilke Havinga, Lodewijk Bergmans, Vaidas Gasiunas, Christoph Bockisch, Johan Fabry, and Theo D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe-VUB-12, Vrije Universiteit Brussel, 2006.

[BMO05] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over Dynamic Properties of Program Execution. In *Proceeding of the Dynamic Aspects Workshop*, 2005.

[Bon03] Jonas Bonér. AspectWerkz - Dynamic AOP for Java. `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`, 2003.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35, 1986.

[BSM+08] Christoph Bockisch, Andreas Sewe, Mira Mezini, Arjan de Roo, Wilke Havinga, Lodewijk Bergmans, and Kris de Schutter. Modeling of Representative AO Languages on Top of the Reference Model. Technical Report AOSD-Europe-TUD-9, Technische Universität Darmstadt, 2008.

[BVD05] Jonas Bonér, Alexandre Vasseur, and Joakim Dahlstedt. JRockit JVM Support for AOP, Part 1. `http://dev2dev.bea.com/pub/a/2005/08/jvm_aop_1.html`, 2005.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000.

[CLI06] Common Language Infrastructure (CLI). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, 2006.

[CLW03] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized Aspect Calculus: A Core Calculus for the Direct Study of Aspect-Oriented Languages. `ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf`, 2003.

[DA99] David Detlefs and Ole Agesen. Inlining of Virtual Methods. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 1999.

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the Dynamic Behaviour of AspectJ Programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[DKM02] Christopher Dutchyn, Gregor Kiczales, and Hidehiko Masuhara. AOP Language Exploration Using the Aspect Sand Box. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, 2002.

[Dmi01] Mikhail Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.

[dR07] Arjan de Roo. Towards More Robust Advice: Message Flow Analysis for Composition Filters and its Application. Master's thesis, University of Twente, Netherlands, 2007.

[DR08] Robert Dyer and Hridesh Rajan. Nu: a Dynamic Aspect-Oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2008.

[dRHH+08] Arjan de Roo, Michiel Hendriks, Wilke Havinga, Pascal Durr, and Lodewijk Bergmans. Compose*: a Language- and Platform-Independent Aspect Compiler for Composition Filters. In *First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, Paphos, Cyprus*, Cyprus, July 2008. No publisher.

[DRL08] Homepage of Harmony Dynamic Runtime Layer Virtual Machine. `http://harmony.apache.org/subcomponents/drlvm`, 2008.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the Symposium on Principles of Programming Languages*, 1984.

[Ecl] Homepage of Eclipse. `http://www.eclipse.org`.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[FGH02] Stephen Fink, David Grove, and Michael Hind. The Design and Implementation of the Jikes RVM Optimizing Compiler. Tutorial Notes from Conference on Object-Oriented Programming, Systems, Languages, and Applications, November 2002. Available at jikesrvm.sourceforge.net/info/presentations.shtml.

[FQ03] Stephen J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

[GDN+08] Ryan M. Golbeck, Samuel Davis, Immad Naseer, Igor Ostrovsky, and Gregor Kiczales. Lightweight Virtual Machine Support for AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 2000.

[GVG04] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. In *Proceedings of the Component And Middleware Performance Workshop, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[Hau06] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Technische Universität Darmstadt, 2006.

[HB05] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2005.

[HBA08] Wilke Havinga, Lodewijk Bergmans, and Mehmet Akşit. Prototyping and Composing Aspect Languages: using an Aspect Interpreter Framework. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 2008.

[HBMO03] Michael Haupt, Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Towards Aspect-Aware Execution Models. Technical Report TUD-ST-2003-01, Technische Universität Darmstadt, 2003.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1992.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2004.

[Hir03] Robert Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In *Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer, 2003.

[HM04a] Michael Haupt and Mira Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In *Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer, 2004.

[HM04b] Michael Haupt and Mira Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. In *Proceedings of the French Workshop on Aspect-Oriented Programming*, 2004.

[HMB+05] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM Press, 2005.

[Hot08] Homepage of Java HotSpot VM. `http://java.sun.com/docs/hotspot/`, 2008.

[HS07] Michael Haupt and Hans Schippers. A Machine Model for Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 2007.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1997.

[Ins] Java Instrument Package. `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html`.

[ITK+03] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2003.

[JAM07] Homepage of Java Aspect Metamodel Interpreter. `http://jami.sf.net/`, 2007.

[Java] Homepage of Java Reflection API. `http://java.sun.com/docs/books/tutorial/reflect/`.

[Javb] Homepage of JavaGrande Benchmarks. `www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/`.

[JBo] Homepage of JBoss AOP. `http://www.jboss.com/products/aop`.

[JCC+06] Andrew Jackson, Siobhán Clarke, Matt Chapman, Andy Dean, and Christoph Bockisch. Deliver Preliminary Support For Top Priority Use Cases. Technical Report AOSD-Europe-IBM-64, IBM UK, 2006.

[JCCB07] Andrew Jackson, Siobhán Clarke, Matt Chapman, and Christoph Bockisch. Deliver Preliminary Support for Next-Priority Use Cases. Technical Report AOSD-Europe-IBM-80, IBM UK, 2007.

[Jik] Homepage of Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[JJT] Homepage of JJTree. `https://javacc.dev.java.net/doc/JJTree.html`.

[JNI] Homepage of Java Native Interface. `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`.

[JVM] Homepage of Java Virtual Machine Tool Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[Ker] Mik Kersten. AOP Tools Comparison. `http://www-106.ibm.com/developerworks/java/library/j-aopwork1/`.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

[KM05] Gregor Kiczales and Mira Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 2005.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[LY99] Tim Lindholm and Frank Yellin, editors. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.

[Mel99] Steve Meloan. *The Java HotSpot Performance Engine: An In-Depth Look*. Sun Microsystems, `http://java.sun.com/developer/technicalArticles/Networking/HotSpot/index.html`, 1999.

[Mil04] Todd Millstein. Practical Predicate Dispatch. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 2004.

[MKD02] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Proceedings of the*

Foundations of Aspect-Oriented Languages, Workshop at International Conference on Aspect-Oriented Software Development. Springer, 2002.

[MKD03] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Proceedings of the Conference on Compiler Construction*. Springer, 2003.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the International Conference on Aspect-Oriented Software Development*, New York, NY, USA, 2003. ACM.

[MR07] Suraj Mukhi and Nico Rottstädt. Survey of Open-Source JVMs. `http://www.st.informatik.tu-darmstadt.de/pages/projects/ALIA/doc/JVMSurvey.pdf`, 2007.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the International Conference on Compiler Construction*, 2003.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2003.

[PGA01] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Homogenous AOP with PROSE. Technical report, Department of Computer Science, ETH Zürich, 2001.

[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2002.

[PRO] Homepage of PROSE. `http://prose.ethz.ch`.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, 2001.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[RDHN06] Hridesh Rajan, Robert Dyer, Youssef Hanna, and Harish Narayanappa. Preserving Separation of Concerns through Compilation. Technical Report 405, Iowa State University, 2006.

[RDNH06] Hridesh Rajan, Robert Dyer, Harish Narayanappa, and Youssef Hanna. Nu: Towards an AspectOriented Invocation Mechanism. Technical report, Iowa State University, 2006.

[RS03] Hridesh Rajan and Kevin J. Sullivan. Eos: instance-level aspects for integrated system design. In *Proceedings of the European Software Engineering Conference held jointly with International Symposium on Foundations of Software Engineering*. ACM Press, 2003.

[SBM08] Andreas Sewe, Christoph Bockisch, and Mira Mezini. Redundancy-free Residual Dispatch. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages*, 2008.

[SdM03] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2003.

[SDR08] Rakesh B. Setty, Robert E. Dyer, and Hridesh Rajan. Weave Now or Weave Later: A Test Driven Development Perspective on Aspect-oriented Deployment Models. Technical Report TR 08-02, Iowa State University, Iowa State University, 2008.

[Sma] Homepage of Smalltalk. `http://www.smalltalk.org`.

[SMU+04] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, 2004.

[Soo] Homepage of Soot. `http://www.sable.mcgill.ca/soot/soot_download.html`.

[SPE] Homepage of SPEC JVM98 Benchmark Suite. `http://www.spec.org/osg/jvm98/`.

[Squ] Homepage of Squeak. `http://www.squeak.org/`.

[Ste] Homepage of Steamloom. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp`.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, 2003.

[SYK+01] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*. ACM Press, 2001.

[SYN02] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An Empirical Study of Method In-lining for a Java Just-in-Time Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. USENIX Association, 2002.

[Tan06] Éric Tanter. An Extensible Kernel Language for AOP. In *Proceedings of the Workshop on Open and Dynamic Aspect Languages, International Conference on Aspect-Oriented Software Development*, 2006.

[TN04] Éric Tanter and Jacques Noyé. Motivation and Requirements for a Versatile AOP Kernel. In *Proceedings of the European Interactive Workshop on Aspects in Software*, 2004.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1987.

[Vas04] Alexandre Vasseur. Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address It? `http://aspectwerkz.codehaus.org/downloads/papers/aosd2004-daw-aspectwerkz.pdf`, 2004.

[VS04] Wim Vanderperren and Davy Suvée. Optimizing JAsCo Dynamic AOP through HotSwap and Jutta. In *Proceedings of Dynamic Aspects Workshop*, 2004.

[VSV+05] Wim Vanderperren, Davy Suvée, Bart Verheecke, María A. Cibrán, and Viviane Jonckers. Adaptive Programming in JAsCo. In *Proceedings of 4International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *Proceedings of Transactions on Programming Languages and Systems*, 26, 2004.

[xal] Homepage of Xalan-Java XSLT processor. `http://xml.apache.org/xalan-j`.