On the design of software composition mechanisms and the analysis of composition conflicts

Wilke Havinga

Ph.D. dissertation committee:

Chairman and secretary:
Prof. Dr. Ir. A.J. Mouthaan, University of Twente, The Netherlands

Promotor:
Prof. Dr. Ir. M. Akşit, University of Twente, The Netherlands

Assistent-promotor:
Dr. Ir. L.M.J. Bergmans, University of Twente, The Netherlands

Members:
Prof. Dr. S. Mullender, University of Twente, The Netherlands
Dr. Ir. M. van Sinderen, University of Twente, The Netherlands
Prof. Dr.-Ing. M. Mezini, Technische Universität Darmstadt, Germany
Prof. Dr. V. Jonckers, Vrije Universiteit Brussel, Belgiẽ
Dr. H. Masuhara, University of Tokyo, Japan
Dr. S. Herrmann, Germany

CTIT Ph.D. thesis series no. 09-142. Centre for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede, The Netherlands.

ISBN 978-90-365-2842-9
ISSN 1381-3617 (CTIT Ph.D. thesis series no. 09-142)

Cover design: *Composition and Decomposition*, Sweden, 2009, by Wilke Havinga
Printed by Ipskamp Drukkers, Enschede, The Netherlands
Copyright © 2009, Wilke Havinga, Enschede, The Netherlands

On the design of software composition mechanisms and the analysis of composition conflicts

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 10 juni 2009 om 15:00 uur

door

Wilke Havinga

geboren op 2 februari 1980
te Assen

Dit proefschrift is goedgekeurd door

Prof. Dr. Ir. M. Akşit (promotor)
Dr. Ir. L.M.J. Bergmans (assistent-promotor)

Any problem in Computer Science can be solved with another layer of indirection
David Wheeler, ca. 1950

..except for the problem of too many levels of indirection
corollary, attributed to Kevlin Henney

Acknowledgments

Four years ago, I started my work as a PhD student in the Software Engineering research group at the University of Twente. Some months prior to that, I had finished my Masters thesis within the same research group, on a subject related to aspect-oriented programming. My experience from several part-time jobs as a student, where I used various programming languages and learned about their strong points as well as limitations, first awoke my interest in programming language research.

Another important reason, however, was the enthusiast supervision of first my Masters and later my PhD work by Lodewijk Bergmans. He first encouraged me to consider doing a PhD, and indeed without his help, enthusiasm and motivation as a daily supervisor, I might not have started this work, much less have finished it successfully. For this, I thank him first and foremost.

In addition, I want to thank my promotor, Mehmet Aksit, whose lectures on Aspect Oriented Programming first attracted me to the Software Engineering research group. His feedback and guidance are visible throughout this work, and I am sure that his advice (regarding writing style, among other things) has substantially helped the acceptance of several conference papers.

I would like to thank the other members of my PhD committee, Prof. Dr. Viviane Jonckers, Prof. Dr.-Ing. Mira Mezini, Dr. Hidehiko Masuhara, Dr. Stephan Herrmann, Prof. Dr. Sape Mullender, and Dr. Ir. Marten van Sinderen, for the time they took to read and comment on my thesis, and for traveling to Enschede, a considerable distance in some cases.

I want to thank the members of the Software Engineering group, including several former members, as well as the Formal Methods group. In particular, I want to thank Arend Rensink, who created the Groove tool set, Tom Staijen, with whom I worked in the European Network of Excellence on Aspect Oriented Software Development, Istvan Nagy, Pascal Durr, and Gurcan Gulesir, all part of the "aspect gang" with whom I shared an office for several years.

I also want to thank the people behind the European Network of Excellence on Aspect Oriented Software Development, for providing an environment for collaboration on aspect-oriented languages. Specifically, I worked most closely with several people in the Languages Lab, in particular Johan Brichau, Christoph Bockisch, Theo d'Hondt, Mira Mezini, Jacques Noyé, Kris Gybels, and several others, who helped develop a meta-model of aspect languages that also underlies a substantial part of the work in this thesis.

Finally, I would like to thank my family and friends, who supported me during the past years.

Abstract

A large variety of programming languages exists, and these languages differ substantially with regard to the modularization and composition mechanisms they support. Such differences may have an impact on important software engineering characteristics, such as reusability, flexibility, analyzability and stability. The design of software languages often entails making trade-offs between such characteristics.

In this thesis, we study several state-of-the-art programming languages, in particular focusing on aspect-oriented languages. Aspects have been proposed as a means to improve software modularization in the presence of crosscutting concerns; the aim is that such improved modularization leads to better software *maintainability*. In this thesis, we strive to improve several engineering characteristics, in particular those that are influenced by the use of aspect-oriented languages.

Specifically, to improve the *analyzability* of programs written in aspect-oriented languages, we propose several analysis techniques. First, we introduce an algorithm that ensures the declarative application of introductions by aspects. Such introductions, which specify changes to the structure of a program, can be used as a powerful composition mechanism that improves the reusability and flexibility of program modules. However, the use of introductions can also cause new kinds of composition conflicts – for example, under some circumstances it may not be possible to unambiguously resolve the compositions specified by introductions. We propose automated tools to detect such conflicts. In addition, we employ graph-based formalisms to detect several other categories of issues related to introductions by aspects. For example, aspect-based composition constructs may break other composition

rules in the language, or may lead to unintended compositions. By modeling relevant parts of aspect-oriented programs in terms of existing graph-based formalisms, we are able to employ existing analysis tools to detect the occurrence of such conflicts.

In addition to improving the *analyzability* of aspect languages, we attempt to improve *changeability*, both from the perspective of *software engineers* who write programs using aspects, as well as from the perspective of *language engineers* who prototype new (aspect) languages. To this end, we define an aspect metamodel interpreter framework, which can be used to define many aspect-oriented languages, including domain-specific aspect languages, of which we give several examples. Using this framework, it is also possible to write programs that use aspects written in multiple aspect languages, expressed in terms of the interpreter framework. This composition of aspects written in multiple languages is enabled by the common high-level structure and control flow of aspects defined by the interpreter framework, in addition to a constraint mechanism that can be used to specify resolutions to potential composition conflicts or ambiguities.

Finally, we present a powerful composition infrastructure that supports the definition of a range of composition mechanisms, such as different styles of inheritance, but also aspect-based or domain-specific composition mechanisms. The infrastructure is defined in terms of a language that defines no *fixed* composition semantics, but provides a generic mechanism for constructing composition mechanisms from primitives that are expressed as first-class objects. For this reason, the infrastructure offers more *flexibility* than existing languages that offer only a fixed set of composition mechanisms, since it enables the definition of domain-specific composition mechanisms, which in turn can provide better modularity (and hence, maintainability). In addition, the infrastructure improves *composability*, since it allows the design and implementation of composition mechanisms that are integrated and work together. Finally, since it is based on a computational model, rather than a code transformation model, it allows for (experimenting with) more dynamic compositions.

Contents

1 Introduction

In this thesis, we strive to improve the state-of-the-art modularization techniques and composition mechanisms supported by programming languages; in particular we focus on aspect-oriented languages.

This introductory chapter is structured as follows: section 1.1 discusses the context and background of this thesis. Section 1.2 gives a brief introduction to aspect-oriented programming languages, while section 1.3 explains some common terminology and concepts related to aspect-oriented programming. Finally, section 1.4 contains a detailed outline of this thesis.

1.1 Introduction and Background

Software is typically decomposed into separate parts, which we call *modules*. Separation into modules allows programmers to group pieces of the program that are functionally related, or that address closely related responsibilities of the program.

Modular programming languages support the expression of modules explicitly, e.g., by means of syntactic language constructs. For example, Modula-2 lets programmers define modules that consist of an *explicit interface* specification, as well as an implementation part [128]. These modules can *encapsulate* the implementation of a part of the program, which means that the implementation details of a module are made inaccessible to the rest of the program (see [11, chapter 3]). Such encapsulation can have the advantage that the internal implementation of a module can be changed, without affecting the implementation of other modules — as long as the explicit interface of the module remains the same.

Encapsulation is particularly useful if systems are designed such that each module addresses one important design decision (such as the choice to use a particular algorithm or data structure). Since each module encapsulates the specific means of addressing a design decision, that design decision can be reconsidered without significantly affecting other parts of the program [112].

Even in programming languages that do not support explicit constructs for defining modules, programmers may adopt a way of working that provides modularization. For example, in C programs, related functionality is typically grouped by source file, and in that sense, each source file can be considered as a module. Although the language does not include constructs that explicitly express the notion of module interfaces (except at the fine-grained level of function definitions), it is still possible to define explicit interfaces by means of header files (commonly named with the postfix '.h'), thus hiding the details of a corresponding implementation file (usually named with the postfix '.c').

Programming languages support various ways in which programmers can decompose a program into modules. As exemplified above, some languages provide explicit language constructs that support modularization, while others may support commonly accepted ways of working that serve the same goals, i.e., modularization and information hiding. However, even between languages that support modules with explicit interface specifications, substantial differences with regard to modularization support exist.

For example, languages differ in their degree of support for (and enforcement of) encapsulation. Strong encapsulation means that no (other) module can access anything that has not been made available through the explicit interface of a module. However, many languages weaken their enforcement of encapsulation under specific conditions, for example, by allowing modules that *extend* the functionality of an existing module to access some or all of the state encapsulated by the module that is being extended.

As another example of support of modularization, object-oriented languages define classes as a description of a module that encapsulates the behavior and data of a part of the program.

To integrate multiple modules into a coherent program, *composition mechanisms* are needed. An example of a *composition mechanism* is a *call* between modules. Calls allow the invocation of functionality that is defined separately (e.g., in a different module), thus integrating those modules.

Another composition mechanism, supported by object-oriented languages, is *inheritance*, which lets programmers create classes that refine the functionality of other class(es). This allows programmers to create a hierarchy of modules, where common functionality is defined at the top of the hierarchy, and specific functionality at the bottom.

The choice for a particular decomposition structure of the program influences certain qualities of the software. As an example, the complexity of call-patterns between modules can be used as a measure that gives an indication of the quality of the chosen decomposition structure; if most calls are encapsulated *within* the implementation boundaries of modules, the complexity of call-patterns *between* modules may decrease. This is an indication that each module indeed encapsulates functionality that is strongly related.

Adaptability is an example of a software quality that may be strongly affected by the chosen decomposition structure, as well as the composition mechanisms supported by a language. A program may be *adapted* by replacing one module with another, provided that the adaptation is correct and the chosen decomposition structure allows this replacement without affecting other parts of the program. As an example, consider several modules that implement different sorting algorithms; if each sorting algorithm is encapsulated by a module, that module can be replaced by another, provided that they support the same interface to access the actual data to be sorted.

Note that there often exist *trade-offs* between different software qualities. For example, one way to obtain a low call-pattern *complexity* between modules is by defining coarse-grained (i.e., large) modules. However, this may adversely affect the *adaptability* of the system, since it is typically easier to replace fine-grained (small) modules.

The potential for *evolution* may also be enhanced by the composition mechanisms available in a language. For example, inheritance mechanisms allow for the definition of abstractions that may *evolve* in several ways. As an example, consider a class that implements an input/output abstraction, allowing data to be read from or written to a data stream. While the initial version of a program may only support reading and writing of data to files, the use of inheritance enables later versions to, for example, add support for reading data over a network, without making significant changes to the rest of the program.

Modular design is a way to map requirements of a software system to modules. For this purpose, a designer has to identify the *concerns* that are relevant to the design of a software system. A *concern* is defined as a focus of interest pertaining to the development of a system under design, its operation or any other matters that are important to one or more stakeholders [125]. In other words, the relevant concerns are *identified* by stakeholders, such as users, customers, market analysts, regulators and software engineers [11, chapter 2]. Possible motivations for modular design are to reduce complexity, and to enhance adaptability and evolvability.

At the requirements level, a concern may correspond to one or more requirements that logically belong together. At the implementation level, a concern corresponds to a single module, or sometimes a set of modules, that are composed using the composition mechanisms of the language or the system.

In this context, *separation of concerns* is considered the study and realization of each concern *in isolation* for the sake of its own consistency [125, 45]. If the identified concerns do not match the modular decomposition structure of a program, the quality of the software may degrade, since there is a *mismatch* between the design (concerns) and the implementation (modules).

Specifically, in the context of software engineering, such a mismatch may cause one or more of the following issues:

(1) *Tangling* occurs when multiple concerns are mixed together in one module [125]. Tangling may negatively influence certain software qualities. For example, tangling may make it harder to *reuse* a module, since the module addresses several concerns, and a software designer may wish not to reuse all of them at the same time.

(2) *Scattering* is the occurrence of elements that belong to one concern in modules encapsulating other concerns [125]. Like tangling, this may negatively affect some software qualities. For example, when the implementation of a concern is spread over several modules, it becomes harder to *maintain* or *evolve* the implementation of the scattered concern in the future.

(3) The combination of tangling and scattering is called *crosscutting*. A concern that cannot be modularly represented within the chosen decomposition structure due to tangling and scattering, is called a *crosscutting concern*.

(4) *Code replication* is the occurrence of nearly identical code in multiple places, often resulting from a pattern of copying and editing existing source code fragments. Code replication may occur when a concern cannot be properly localized in one module. Such code replication has been indicated as a common source of errors, introduced into programs during the implementation phase [35].

Aspect-oriented programming languages aim to address the issues identified above, by allowing the modular expression of crosscutting concerns. Such a modular representation of a crosscutting concern is called an *aspect*.

Aspect-oriented programming languages enable the modular expression of crosscutting concerns by means of supporting *quantification* over a program. Quantification can be defined as follows: “In programs P, whenever a condition C arises, perform action A” [55].

Aspects may perform actions at one or more points during the execution of a program. Such points, where two or more concerns are composed by means of quantification, are called *join points*.

To express quantification, aspect oriented languages typically support *pointcuts*. A *pointcut* is a predicate that matches join points. To be precise, pointcuts can be seen as a relation from join points to booleans, where the domain of this relationship is all possible join points [125].

In general, one or more *pointcuts* can be *bound* to one or more *advices*. An *advice* is an aspect element that defines the functionality implemented by an aspect. Advices may augment or constrain (the behavior of) other concerns at a join point, matched by a pointcut expression [125].

The following section explains aspects oriented programming in more detail.

1.2 An introduction to aspect-oriented languages

In this section, we demonstrate the use of two aspect oriented languages. To illustrate the need for the composition mechanisms provided by these languages, we show the implementation of a small software system in Java. We discuss the design issues that we encounter, based on the use of only object-oriented composition mechanisms. This is followed by an implementation of the same system, using two different aspect-oriented languages. These languages, AspectJ and Compose*, are used in multiple places throughout this thesis. The examples shown here serve as a very brief introduction; for more detailed information, we refer to the respective websites of these languages [14, 10].

As an example software system, we discuss the implementation of a *centralized electronic hotel booking system*. Following common object-oriented or modular design conventions, we decompose the system into modules that represent hotels, rooms, bookings, etc. — that is, a decomposition structure that reflects important concepts from the given design domain.

However, given such a decomposition structure, several concerns that may be relevant to the design of the system cannot be expressed in a modular way. For example, access control (e.g., how to control who is allowed to make and cancel bookings), or synchronization (what if several travel offices attempt to book the same room at the same time?) cannot be expressed in a modular way, when using the above decomposition. The next subsection explains why this is the case.

1.2.1 Implementation in Java

Related to the electronic hotel booking system, we show the implementation of two concerns in Java. Based on this example, we discuss several Java-based design alternatives for such a system, and explain the issues caused by crosscutting concerns.

Below, we consider the implementation of two concerns: first, it must be possible to create or cancel *bookings*, and secondly, we want to *control access* to the booking system such that only users with appropriate access rights are allowed to modify the registered bookings. In Java, this could be implemented as shown in listing 1.1.

```java
public class Booking
{
  public boolean addBooking() throws AccessViolationException
  { // Functionality related to access control
    if (! User.getCurrentUser().isAllowed("addBooking"))
      throw new AccessViolationException
        ("Action addBooking not permitted to this user");
    // Functionality related to adding the booking
    // e.g. check for available room, then create booking
    ..
  }

  public boolean cancelBooking() throws AccessViolationException
  { // Functionality related to access control
    if (! User.getCurrentUser().isAllowed("cancelBooking"))
      throw new AccessViolationException
        ("Action cancelBooking not permitted to this user");
    // Functionality related to canceling a booking
    // e.g. check that the check-in date has not passed yet, then
        cancel the booking
    ..
  }
}
```

Listing 1.1: Partial Java implementation of a hotel booking system

Class `Booking` implements the concern *to add bookings* and/or *to cancel bookings*. In addition, we can define another class, `User`, to keep track of the current user of the system, as well as a set of actions allowed to be executed by this user. This access control concern cannot be fully separated however: to implement access control within class `Booking`, we need to include code that invokes the appropriate access checks, as shown on lines 5–7 and 15–17 of listing 1.1.

Several design issues are apparent in this code: first of all, the access control concern is not directly implemented by a single module. Instead, its implementation is scattered over several modules (at least classes `User` and `Booking`, in this example), as well as tangled with the implementation of another concern (class `Booking`, in this case).

1.2.2 Alternative solutions using object-oriented techniques

While reading the previous section, you may have wondered whether it is in fact infeasible to properly separate the booking and access control concerns, using object-oriented techniques. In this simple example, the issue of tangling can indeed be partially addressed by using inheritance or object aggregation. We show how this is accomplished, and then discuss why this still leaves several design issues to be addressed.

First of all, the tangled code related to access control can be removed from class `Booking`, as shown in listing 1.2:

```
public class Booking
{
  public boolean addBooking()
  { // Only functionality related to adding a booking
    ..
  }

  public boolean cancelBooking()
  { // Only functionality related to canceling a booking
    ..
  }
}
```

Listing 1.2: Separating the access control concern from class `Booking`

Then, listings 1.3 and 1.4 show two (partial) alternative implementations of the access control concern. These implementations are based on inheritance and object aggregation, respectively. Object aggregation means that the class of an object declares one or more other objects in its implementation part, thereby encapsulating the implementation of these objects. Even in this almost trivial example, several serious drawbacks of these alternatives are visible, however. We discuss these below.

```java
public class BookingWithAccessControl extends Booking {
  public boolean addBooking() throws AccessViolationException
  { // Functionality related to access control
    if (! User.getCurrentUser().isAllowed("addBooking"))
      throw new AccessViolationException
      ("Action addBooking not permitted to this user");

    return super.addBooking();
  }

 // idem for cancelBooking, etc.
  ..
}
```

Listing 1.3: Implementation of access control using inheritance

```java
public class BookingAccessControlWrapper {
  Booking booking; // aggregated object

  public boolean addBooking() throws AccessViolationException
  { // Functionality related to access control
    if (! User.getCurrentUser().isAllowed("addBooking"))
      throw new AccessViolationException
      ("Action addBooking not permitted to this user");

    return booking.addBooking();
  }

 // idem for cancelBooking, etc.
  ..
}
```

Listing 1.4: Implementation of access control based on object aggregation

First and foremost, when using either of these alternatives, part of the access control concern still needs to be replicated (with minor modifications) for every module to which the access control concern is applied. Such code replication is very error-prone. A report on the practical problems that are experienced in code replication, for instance, can be found in [35].

Second, in the implementation based on object aggregation, even methods for which access is unrestricted need to be explicitly forwarded. On the other hand, the implementation based on inheritance may also have serious drawbacks, as the new class, `BookingWithAccessControl`, is not a “proper” specialization of a booking. This may cause a problem if in future versions, class `Booking` is specialized using subclassing.

Finally, one may of course require to add more concerns to the example, such as a concurrency control mechanism to accessing all bookings. If the software offers various alternative implementations by enabling and/or disabling the above mentioned concerns, the number of implementation alternatives grows with respect to the number of concerns. For example, it may be preferable to have concurrency with access control, only access control, or only concurrency.

The above example illustrates that the implementation of the access control concern is tangled with the implementation of other concerns, as well as it is scattered over several modules, and parts of its implementation are replicated in several places. It is not possible to eliminate these problems fully by using object-oriented languages, since these languages do not provide explicit language constructs to separate crosscutting concerns effectively.

1.2.3 Implementation in AspectJ

Aspect-oriented programming languages address the issues identified in the previous section, by allowing the modular expression of crosscutting concerns. Listing 1.5 shows a modularized implementation of the access control concern discussed previously, using AspectJ. This solution assumes the implementation of class `Booking` as shown in listing 1.2, which does not contain any code related to the access control concern.

```
 1 public aspect AccessControl {
 2   pointcut restricted(): execution(* Booking.*(..));
 3 
 4   before() throws AccessViolationException: restricted()
 5   {
 6     String action = thisJoinPoint.getSignature().getName();
 7     if (! User.getCurrentUser().isAllowed(action))
 8       throw new AccessViolationException
 9       ("Action " + action + " not permitted to this user");
10   }
11 }
```

Listing 1.5: Modularized implementation of access control in AspectJ

In listing 1.5, an aspect named `AccessControl` is defined. Aspects, as defined in AspectJ, specify the implementation of crosscutting concerns based on *pointcuts* and *advice*. In this example, line 2 defines a pointcut that can be identified by name (`restricted`). The pattern after the colon defines points in the program where the aspect wants to influence the behavior of the program. These points are commonly called *join points*, and in this example, the pointcut is specified as a pattern that matches the *execution* of any methods within class `Booking`.

Line 4 specifies that the pointcut `restricted`, defined on line 2, is connected to an *advice* definition, specified on lines 4–10. In this example, whenever a *join point* that matches the pointcut `restricted` is encountered, the advice definition is executed *before* the original method behavior is executed. Lines 6–9 implement the behavior of the access control concern: if a user is not allowed to execute a specific action, an exception is thrown.

The modular implementation of the access control concern shown in listing 1.5, addresses the identified issues with the previously shown object-oriented implementations. To start with, the implementation of the access control concern is now fully separated from class `Booking`. This solves the issues related to tangling and scattering. In addition, the replication of code to invoke the access control checks is also addressed, since the advice specification (lines 4–10 in listing 1.5) is *parameterized* by the join point context. This can be seen in line 6, where the action that the user is attempting to execute is obtained (through the join point context, accessible via the pseudo-variable `thisJoinPoint`) from the method that is being executed, e.g. `addBooking` or `cancelBooking`. This way, the access control check on line 7 can be parameterized by this information, so that this code needs to be written only once, rather than for each action to which access control is applied.

1.2.4 Implementation in Compose*

A different aspect oriented programming language is realized in Compose*, a language and compiler that implements the Composition Filters model. A key design goal of the Composition Filters model is to improve the composability of programs written in object-based programming languages. The Composition Filters model has evolved from the first (published) version of the Sina language in the late 1980s [13, 12], to a version that supports language independent composition of crosscutting concerns [10, 23].

The Composition Filters model can be applied to object-based systems. In such a system, objects can send messages between each other, e.g. in the form of method calls or events. In the Composition Filters model, these mes-

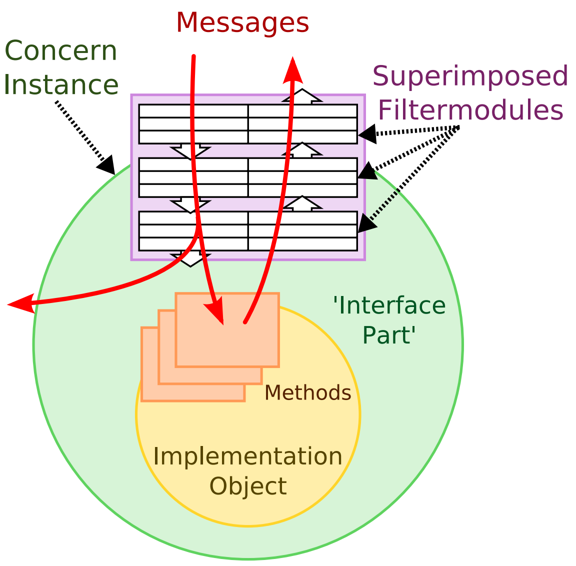


Figure 1.1: Overview of the Composition Filters model

sages can be filtered using a set of *filters*, as shown in figure 1.1. Each *filter* has a *filter type*, which defines the behavior that should be executed if the filter accepts a message, as well as behavior to be executed if a filter rejects a message. The matching behavior of a filter is specified by a sequence of *filter expressions*, which offer a simple declarative language for state and message matching. Filters defining related functionality are grouped in so-called *filter modules*. Filter modules can also encapsulate internal state, as well as references to external objects.

To indicate where filter modules should be *superimposed*, that is, to which objects a given filter module should be applied, Compose* uses *superimposition selectors*. A superimposition selector selects a set of classes based on query expressions written in a Prolog-based selector language. The result is that all messages sent to and received by all instances of those selected classes, are subjected to the filters within the filter module.

A main feature of the Composition Filters model is that it is language independent: it can be applied to any language that can be construed to support the notion of objects and messages between objects. Historically, the model has been applied to several languages, including Smalltalk [126], C++ [59], C#, Java [127], and recently even C.

We show the implementation of the access control concern in terms of the Composition Filters model, as shown in listing 1.6.

```
 1 concern AccessControl {
 2   filtermodule access_fm {
 3     externals
 4       user : User.getCurrentUser();
 5     conditions
 6       alowed : user.isAllowed;
 7     inputfilters
 8       control : Error("Action not permitted to this user") = {
            allowed => [*.*] }
 9   }
10
11   superimposition {
12     selectors
13       restricted = { C | isClassWithName(C, ’Booking’) };
14     filtermodules
15       restricted <- access_fm
16   }
17 }
```

Listing 1.6: Modularized implementation of access control in Compose*

In this listing, lines 2–9 define a filter module `access_fm`, which specifies an input filter (line 8) that raises an exception if, when the filter is evaluated, the condition “allowed” is false. This condition is defined on line 6, as a call to method `isAllowed` on object `user`. The object `user` is defined on line 4 as an object external to this filter module, initialized by a call to the static method `User.getCurrentUser()`. Similar to the AspectJ example, condition implementation methods (such as `isAllowed`) can use the join point context to determine whether the condition should currently evaluate to true or false.

Line 13 specifies a set of “restricted” classes, expressed as a Prolog predicate over the structure of the program. In this case, only the class named `Booking` is selected. Then, on line 15, the filter module `access_filter` as defined above is superimposed on the classes selected by the selector `restricted`.

In this way, the above concern definition implements the same behavior as the AspectJ example. This concludes our practical introduction to aspect oriented programming languages.

1.3 A clarification of aspect-related terminology

In this section, we explain some common terminology and concepts related to aspect-oriented programming.

1.3.1 Aspects: Quantification and Obliviousness?

One of the defining features of aspect-oriented languages is the support for quantification over the program structure and/or execution state. As described by Filman and Friedman in their paper *"AOP = Quantification + Obliviousness"*, the concept of quantification can be summarized thus: *"In programs P, whenever condition C arises, perform action A"* [55].

We consider the specification of such a *"condition C"* a generic notion of the term "pointcut" as the mechanism by which aspect languages specify quantification. Note that, although all known aspect languages support the concept of quantification, not all of them necessarily define an *explicit language construct* that can directly be identified as a "pointcut", in the sense as defined in AspectJ and similar languages. Particularly in the context of domain-specific aspect languages (DSALs), pointcuts may not appear as explicit language constructs at all. For example, in chapter 4 we discuss the language COOL, which clearly exhibits both the aforementioned features of an aspect language, but does not define an explicit pointcut construct, identifiable as such. In addition, the notion of quantification is not limited to only dynamic (run time) conditions: it is also possible to quantify over the structure of a program, e.g., some aspect languages (including AspectJ) allow the use of inter-type declarations, such as "In programs P, whenever it contains a class that implements a specific interface, add this additional method to that class".

With regard to obliviousness: this term does not necessarily imply that programmers can be oblivious to the existence of aspects that may affect their "base code". Rather, we understand this to mean that the base code does not contain any explicit symbols (such as method call statements) indicating that an aspect might be invoked, or as Filman puts it in a follow-up paper: *"Obliviousness states that you can't tell that the aspect code will execute by examining the body of the base code"* [52].

In the next subsection, we discuss the meaning of the terms "aspect code" and "base code" as used in the previous sentence.

1.3.2 Aspect vs. Base as *roles* of code

Traditionally, many aspect languages have been implemented as an extension to existing — typically object-oriented — languages. This, in most cases, leads to a conceptual distinction between the “aspect language” and “base language”. Since the existing “base language” cannot directly express the language constructs needed to address crosscutting concerns[1], aspect languages typically define their own keywords and syntax, to support (among other things) quantification over the program structure and/or execution state.

In this thesis, we also frequently use the terms “aspect” and “base” (code). We use these terms to designate the *role* of a particular code element in a given context, i.e., the “base code” is the code that is currently being “advised” by an aspect. This should be understood in exactly the same sense as, when studying a specific procedure call, we would distinguish between code that has the role of “caller” (calling side) and code that has the role of “callee” (invoked side). Note that the use of these terms is thus context dependent, as the same unit of code may fulfill the role of “caller” as well as “callee” depending on the point of view, i.e., a given occurrence of a procedure call. Analogously, code that in the context of one join point (or aspect) is considered “advice”, may very well be seen as part of the “base” code from the context of another join point (or aspect).

Thus, when talking about “aspect” and “base” code, we do *not* necessarily mean “expressions written in the aspect language” vs. “expressions written in the base language”. Sometimes, the meanings of these terms coincide, specifically in cases where a strict line can be drawn between code written in the aspect language and code written in the base language. In other cases, a distinction between aspect and base language may not exist at all. In most cases however, the line is not clear-cut, since advice is often partially or completely specified in terms of “base language” statements. This is the case in AspectJ, for example.

The discussion about the meaning of “aspect” vs. “base” code above is important for the following reason: one way to see aspects is as a strict “meta-layer” on top of an existing “base language”. In this view, aspects observe the behavior of the “base code”, and may influence that behavior. However, in such a view, the specification of how aspects influence the “base code” (typ-

[1]This is the motivation to develop aspect oriented programming languages in the first place

ically called "advice"), is not itself observable by (other) aspects[2], since the behavior of the aspectual "meta-layer" would not be observed by that layer itself. We call this an asymmetric model, as aspects can observe and influence the base code, but not vice versa. In addition, aspects cannot typically influence other aspects. In other words, this model contains two distinct kinds of modules (i.e., classes and aspects). We believe that historically, this perception of aspects was often a de facto choice based on implementation concerns.

As an alternative to the above view, aspects can be seen as mainly providing a different (inverse) "calling convention" as compared to, e.g., the explicit calling conventions found in object-oriented languages. In this view, aspects are seen as a new *composition mechanism*, rather than as a new *kind* of module altogether, i.e., in addition to objects. Advice is seen as any other object-based behavior specification, and can thus potentially be advised by other aspects. In this view, there is no conceptual distinction between aspects and objects, thus we call this view symmetric.

1.4 Motivation and Overview

"The concept of separating concerns is fundamental in software development. Although the separation of a concern can bring many benefits, such as making it easier to understand a concern or to reuse a component, separation can also entail costs, such as making it more difficult to understand the behavior of the system as a whole." [36]

In this thesis, we search for composition and modularization techniques that improve on the state-of-the-art in programming languages. In particular, we focus on aspect-oriented languages, which address the modularization of crosscutting concerns. However, this improved modularization may come at a cost. For example, assuming that all concerns of a design are expressed as properly separated modules (such as aspects), these modules have to be composed again into a coherent program that exhibits the desired behavior. At that point, the improved separation of concerns may lead to new kinds of composition issues. For example, it may not be possible to unambiguously resolve how modules should be composed, or several modules may specify conflicting additions or changes to the program.

[2]At least, this is not possible without defining additional machinery to let "meta-aspects" reflect on aspects.

Thus, while aspects may at one hand improve the modular structure of programs in the presence of crosscutting concerns, they may at the same time make the program less comprehensible or less stable (i.e., more prone to unexpected effects from modifications to the software). In this thesis, we discuss such trade-offs, and try to improve such engineering characteristics of aspect languages, as well as to mitigate potential negative effects, e.g., with regard to analyzability or comprehensibility.

Below, we describe the contents of each chapter in some detail.

1.4.1 Identification and Detection of Composition Conflicts

Chapter 2 studies the use of *annotations* and *introductions* in the context of aspect-oriented programming. As we will show, the combined use of annotations and introductions can help to make design intentions explicitly visible, thus improving the conceptual relation between a design and its implementation. This has the practical benefit that pointcut expressions become more robust, since in many cases they no longer need to depend on accidental program properties that may change when the program evolves. However, the use of introductions can also lead to situations where the composition specified by aspects can be interpreted in multiple ways, as we exemplify using examples based on the introduction of annotations. This is highly undesirable. In chapter 2, we discuss how such cases can be detected, and what can be done to prevent or address such situations.

Chapter 3 introduces a graph-based model of aspect-based composition mechanisms. This model can be used to analyze and detect several types of composition conflicts related to aspects; specifically, issues related to the composition of the structure of a program, as specified by aspects. For example, in addition to the issues identified in chapter 2 (which focuses in particular on the use of annotations), the application of aspects may also break existing rules of the base language, or may lead to unintended (although technically viable) compositions. We design a model that can be used to detect such issues, and use graph-based tools to automate this detection.

1.4.2 Models for Modularization and Composition

Chapter 4 introduces a meta-model of aspect-oriented languages. Since this model supports a wide range of aspect-oriented concepts, it can be used to model diverse aspect languages, including domain-specific aspect languages. We show that languages implemented in terms of this common meta-model of aspects can be composed, in the sense that aspects written in several such languages can be used in the same program. Although this can be applied to general-purpose aspect languages, it is particularly useful in the context of domain specific aspect languages (DSALs), since each DSAL can be used to address concerns within a specific domain, in a way that closely reflects the recurring design concerns in that domain.

Chapter 5 presents a language (’*Co-op*’) with a powerful composition infrastructure that supports the definition of a range of composition mechanisms, such as different variants of inheritance, but also aspect-based or domain-specific composition operators. The purpose of this language is to provide a programming environment that does not fix the composition semantics (as part of the language), but provides a generic mechanism for constructing new composition operators, which are expressed as *first-class* objects. A first-class abstraction may be passed as an argument of a call, it may be returned as a result of a call, it may be stored or retrieved, etc. The ease of modifying or introducing new operators is useful for research and education purposes, the development of domain-specific languages, as well as expressing application-specific compositions.

Finally, chapter 6 concludes this thesis with a summary and list of contributions, as well as an evaluation of the engineering characteristics affected by the work presented in this thesis.

Resolving introductions and detecting ambiguous compositions

2.1 Introduction

Since the introduction of aspect-oriented programming, aspect-oriented languages have evolved in several respects. In this chapter[1], we investigate the consequences of this evolution, focusing in particular on the following:

(1) Pointcut languages have become increasingly *expressive*. Whereas early (implementations of) aspect languages often supported only relatively simple expressions over the structure of a program, today, many pointcut languages support expressive composition mechanisms for pointcut (sub)expressions. Several publications discuss the need for and implementation of such highly expressive pointcut languages [65, 111, 91, 90, 107, 69].

(2) Pointcut languages have also become more *comprehensive* in covering the domain of static and dynamic program properties, allowing more and more kinds of program properties to be used as a selection criterion in pointcut designators. In this chapter, we focus specifically on *annotations* as an important example of a static program property. Other examples indicating this trend are pointcuts based on dynamic (runtime) properties such as control flow [47], data flow [102], or based on multiple events during the execution of the program (stateful aspects, [46]).

(3) Several languages and frameworks support the "introduction" of new program elements based on a crosscutting specification, thus provid-

[1]The contents of this chapter are based on a paper published at the 5th International Conference on Aspect-oriented Software Development, AOSD 2006 [71]

ing new and additional mechanisms for composing the structure of a program. For example, several languages (including AspectJ and Compose*) effectively allow aspects to add ("introduce") additional methods to existing classes.

In this chapter, we particularly study the introduction of *annotations* based on crosscutting specifications, although the results of our research are not limited to only this subject, as we discuss at the end of this chapter (in section 2.8). Annotations are a mechanism that enables the attachment of meta-data to program elements. Various object-oriented programming languages (such as C# [51] and recent versions of Java [119]) support such annotations. Annotations do not have any direct influence on the execution of an application, but they can be used by compile-time tools or meta-facilities — hence the term *metadata*.

The combined use of aspect oriented programming and annotations is a natural match [87, 94, 95], as annotations can be used to explicitly express the *design intention* of elements in a program. Without annotations such information is usually implicit, and therefore invisible in the implementation. This is why the combination of AOP and annotations can be especially useful, since the dependence of pointcuts on accidental properties of the program often leads to fragile pointcuts, i.e., pointcuts that easily "break" when the program later evolves [118]. The explicit expression of design intentions (e.g., by means of annotations) enables programmers to write pointcuts that are much less fragile than those that are based on, for example, naming conventions or structural patterns [107].

In this chapter, we investigate how annotations can be used by aspects, how aspects can *introduce* annotations based on crosscutting specifications ("pointcuts"), and also, how the introduction of annotations can be derived from other annotations. We identify issues caused by the use of powerful pointcut languages in combination with the introduction of elements (such as annotations) in the same structure that is queried by such a pointcut language. We perform an extensive analysis and discuss approaches that can solve these problems. We present a solution within the context of a concrete implementation of the *Compose** language [23, 10]. Finally, we argue that these problems are inherent to the use of expressive pointcut languages, and are not limited to specific tools, AOP languages, the use of annotations, or even to only introductions that change the structure of a program.

2.2 Using annotations in AOP

In this section, we explain how annotations can be *used* by aspects. We also show how aspects can *introduce* annotations based on crosscutting specifications. In addition, we show how this enables the *derivation* of annotations based on properties of a program. Finally, we describe several types of situations in which these techniques can be used to improve the modularization of a program.

2.2.1 Using annotations in pointcuts

Pointcuts that directly refer to the structure or syntax of a program are generally quite fragile: they often break when the base program is refactored [118]. To avoid such fragile pointcuts, we can decorate the program with annotations that explicitly represent design information about the program. These annotations can then be used as a selection criterion in pointcuts. We demonstrate this use of annotations by means of a small example: listing 2.1 shows a simple Java class.

```
@PersistentRoot class User {
  String name;
  String email;
  SessionID session;
  ..
}
```

Listing 2.1: Annotations in Java

The notation `@PersistentRoot class User` specifies that the class `User` has the annotation `PersistentRoot` attached. Annotations can be attached to several kinds of program elements, most notably classes, methods and fields.

The following AspectJ (version 5) example shows a pointcut that selects the execution of all methods within classes that have the annotation `PersistentRoot`:

```
  pointcut isPersistent():
    execution(* (@PersistentRoot *).*(..));
```

Listing 2.2: An annotation-based pointcut

Within the `execution` part of this pointcut, the first `*` indicates that any (method) return type will match. The next part, `(@PersistentRoot *)`, means that the class containing the executed method must have the annotation `PersistentRoot` attached. The remainder specifies that this pointcut definition matches methods regardless of their specific name, number of arguments and argument types.

By writing pointcuts that refer to annotations, these pointcuts depend on explicitly expressed design intentions, rather than (often accidental) syntactical or structural features of a program. For this reason, such pointcuts can be more robust [107].

However, for some cases the usefulness of annotations can still be improved. For example, by directly binding the annotation `PersistentRoot` to every persistent class, those annotations are scattered over the application. In fact, we can state that the annotation `PersistentRoot` is therefore tangled with base classes (such as the class `User` in this example), because their code cannot be fully separated. This may hinder the reusability and evolvability of the classes involved.

For example, it may be desired to make annotations application specific: suppose the class `User` above would be reused in two applications. In one application, it is part of a set of classes that should be made persistent, but not in the other. By directly attaching an annotation to the source code of class `User`, this otherwise reusable class is “polluted” with application-specific information.

Fortunately, several AOP languages support a technique that addresses this, by supporting the *introduction* of annotations based on a crosscutting specification. The next section discusses this technique.

2.2.2 Introduction of annotations

An intuitive AOP solution to the problem of scattered and too strongly bound annotations is to *introduce* annotations (from within an aspect) on program elements selected by a pointcut. In AspectJ, the introduction of annotations can be expressed as follows:

```
declare @type : SessionID+ : @TransientClass();
```

Listing 2.3: Introducing an annotation

This example specifies that the class `SessionID` and all its subclasses should be marked with the annotation `TransientClass`. This annotation expresses some information about the design: presumably, in this application, it does not make sense to permanently store session-ID’s, as they represent volatile information that expires after a limited amount of time. By explicitly specifying this information separate from the base classes, the information is localized in one place. The AspectJ construct `declare @type` means "introduce an annotation on a type". The part between the colons is a standard AspectJ Type-pattern [15], in this case matching the type `SessionID` and its subclasses. The part after the second colon specifies that the annotation `TransientClass` should be attached to the types that match the specified pattern.

Note that the language used to designate places where annotations have to be introduced can be different from the normal pointcut language[2]. In practice, the expressiveness is typically similar to that of the normal pointcut language - and we show that this expressive power is indeed desired, as it enables programmers to express their intended designs more directly.

In the case of AspectJ, we note that the pattern-based pointcut expressions used to introduce annotations have a limited expressiveness. To demonstrate this based on listing 2.3: when introducing annotations on types (classes), it is only possible to express constraints (selection patterns) that are very directly related to the type(s) that we want to attach an annotation to. For example, it is not possible to attach an annotation to types based on the criterion that they should implement a particular method, since the pattern-based selection language used for introductions in AspectJ does not support such expressive selection criteria. Still, AspectJ allows us to express some conditions on the introduction of annotations. As these condition-patterns can include restrictions based on annotations, we can *derive* the existence of one annotation based on the existence (or absence) of others. The next subsection explains how this can be useful.

2.2.3 Deriving annotations

Annotations may be relatively bound to each other: an annotation can be attached to a certain program element, if another —related— program element has a certain annotation.

To illustrate this, we extend the previous example by defining the following rule: "If a field is of a type that has the annotation `TransientClass`, it should be marked by the annotation `TransientField`". This way, we can specify an exception to the general rule that all fields within a class marked by the annotation `PersistentRoot` will be kept in a persistent datastore.

In AspectJ, it is possible to express such a rule as follows:

[2]In AspectJ terminology, the selection part of an introduction specification is called *pattern*. We use a more general notion of the term *pointcut* to also indicate such selection patterns.

```
declare @field: (@TransientClass *) * :@TransientField();
```

Listing 2.4: Deriving an annotation

This declaration specifies that the annotation `TransientField` should be introduced on all fields that match the field-pattern between the colons. Here, the pattern specifies that such fields should be of a type that has the annotation `TransientClass` attached, and that we do not put any constraints on the name of the type or the name of the field. As you can see, the introduction of the annotation `TransientField` *depends on* the fact whether the annotation `TransientClass` is attached to the type of a field.

Another example can be found in [37], where the following *constraint* is specified and enforced: "If a class is decorated with the annotation `WebService`, its public methods (that constitute the webservice) *should have* the annotation `WebMethod`". The difference with the approach we discuss here is that we actively *generate* the dependent annotations, i.e. "If a class is decorated with the annotation `WebService`, its public methods *are marked* with the annotation `WebMethod`". As such rules can express the intention of a programmer more directly, the ability to express such rules has a clear added value.

One might ask why it is useful or necessary to introduce the annotations `TransientField` or `TransientClass` if their places can also be designated directly by the pointcuts that could express the rules above. Using such pointcuts can in fact be sufficient when these annotations are only used within pointcut expressions of aspects. However, (derived) annotations could be used by third party tools or frameworks as well. An example of a non-aspect-oriented tool that could benefit from the mechanism of superimposing and deriving annotations is the Annotation Processing Tool (APT) [1] for Java 1.5 (and newer), which can be used to write so-called "annotation processors". A processor can be registered with APT to match one or more types of annotations, and is then called by APT when a matching annotation occurs in the code of an application.

2.2.4 Summary: scenario's for the use of annotations

In the preceding sections, we have shown several techniques that combine the use of annotations and aspects. We summarize several types of scenario's in which these techniques can be used to improve the modularity of a program:

(1) Directly attaching annotations to the source code of a program is useful to express inherent properties of a given program element. For example, a method that has no side effects could be decorated with an annotation `NoSideEffects`. The presence of this annotation can then be used in pointcut specifications; for example, aspects implementing persistence or caching may want to select methods based on whether or not they have side effects.

(2) Using a crosscutting specification (pointcut) to specify the introduction of annotations is useful to explicitly express a *role* of a given (set of) program element(s). The same class may fulfill different roles depending on the application in which it is used. For example, it may be the case that instances of a class should be persistently stored in one application, but not in another. Thus, by separating such application-specific information from the elements to which it is attached, said elements (e.g., classes) are made more reusable.

(3) Using pointcuts that depend on annotations that are introduced (based on cross-cutting specifications) by other aspects can be useful to further separate generic (i.e., reusable) parts from application-specific parts of a program. For example, we could implement a generic “Observer” aspect, which observes classes that have the annotation “subject” attached. Note that whether or not a class should be a “subject” of observation is a prime example of an application-specific *role*, in this case defined as part of a well-known software design pattern (see [57]). Which classes should be designated as “subject” can be specified by an “application binding” aspect, thus separating the generic and reusable part of the “observer” pattern from the application-specific designation of classes to be observed.

    Note that the designation of classes to be observed has now become a 2-step process: the generic observer aspect depends on the presence of annotations that first have to be introduced by the application-specific binding aspect. Hence, the term *derivation*.

2.3 Problem statement

In the previous section we described how annotations can be used in combination with aspects, by writing pointcuts that refer to annotations. In addition, we have shown how aspects can *introduce* annotations based on crosscutting specifications. In cases where such specifications refer to other annotations, we speak of *deriving* annotations, based on the existence or absence of other annotations. We discussed scenario's where each of these techniques can be useful. We have shown how the use of annotations can help to make pointcut expressions less fragile, and how their use can improve the reusability of modules. However, basing introductions on derived information in this way may result in dependencies between introductions (e.g. of annotations) and the crosscutting specifications used to designate places where introductions should be applied. This section describes the problems that such dependencies may cause.

The main problem caused by dependencies between introductions of annotations is that the order of applying the introductions may matter. If this order is unspecified, ambiguities may occur. For instance, consider the combination of the examples in listings 2.3 and 2.4:

```
1 declare @type : SessionID+ : @TransientClass();
2 declare @field: (@TransientClass *) * :@TransientField();
```

Listing 2.5: Inter-dependent declarations

Both declarations (see listing 2.5) specify the introduction of an annotation. The first declaration (line 1) specifies a selection pattern based on the structure (inheritance hierarchy) of the program, while the second declaration (line 2) specifies a pattern that depends on the existence of another annotation. We assume that, without further specification, no ordering is implied between these two introduction declarations; the declarations may actually be part of different aspects. In this example, there are two possible orders in which these introductions can be applied. The first is to start by evaluating the types matching the condition `SessionId+` and attaching the annotation `TransientClass` to these types (i.e. apply the first introduction), and then to apply the second introduction, which attaches the annotation `TransientField` to fields of a type that have the annotation `TransientClass` attached. The other possibility is to switch the order of applying the two introductions. In that case the annotation `TransientField` will not be introduced anywhere, as the annotation `TransientClass`, on which this introduction clearly depends, has not been attached yet. In this case, it is intuitively clear which order is intended by the programmer: the first order is desired, since the declaration on line 2 clearly depends on the one on line 1. Thus, it makes sense to apply the introduction on line 1 first.

However, such a conclusion cannot always be drawn. Depending on the expressiveness of the pointcut language used to specify where annotations should be introduced, it may not be trivial to see which dependencies exist. For example, when using a Turing-complete pointcut language, it is generally impossible to infer such information by analyzing only the textual representation of pointcut expressions. Using a Turing-complete pointcut language to specify pointcuts can be useful, as it allows for powerful reasoning within pointcut expressions [65, 31]. However, even when a more restricted pointcut language is used, there are still several issues to be addressed.

To start with, there can exist multiple levels of dependencies. This can already be seen in the example above, as the attachment of the annotation `TransientClass` depends on the (static) structure, while at the next level, the attachment of the annotation `TransientField` depends on the attachment of the annotation `TransientClass`. In general, there can be “chains” of introductions that depend on each other. This suggests that some kind of iterative algorithm is needed to resolve the dependencies between introductions of annotations, as it may not be possible to resolve all dependencies in a single pass over all the introductions. However, dependencies between introductions can even be circular. If circular dependencies occur, such an algorithm might never terminate.

In addition to this problem, the existence of dependencies implies that the ordering of evaluating the declarations can lead to different results, as seen in the example above. In the given example it is intuitively clear which ordering is desired. However, given the occurrence of circular dependencies and/or negative dependencies (i.e. where an annotation should only be attached if another annotation is *absent*), there is no intuitive way to tell which order of evaluation is intended by the programmer. In such cases, the specification of annotation-declarations may be ambiguous, unless certain ordering constraints are implied or can be specified by the programmer.

In AspectJ, the ordering of introductions within a single aspect is unspecified. Based on some small experiments, we conclude that the current implementation[3] detects simple dependencies such as in listing 2.5, but yields ambiguous results (i.e. depending on the ordering in the aspect sources in seemingly arbitrary ways) when negative or circular dependencies are involved. When introductions are divided over several aspects, the ordering is also unspecified. Thus, the current implementation does not address the problems

[3]AspectJ 5

related to ambiguous (non-declarative) specifications and the expression of circular dependencies.

We stress that these types of problems are not necessarily limited to introductions of annotations only. Any type of introduction that changes parts of the structure that can also be referred to by pointcuts describing other introductions can potentially lead to similar problems. Whether problematic cases can actually occur depends on the expressive power of the pointcut language. In this chapter, we discuss potential solutions to these problems. We discuss the trade-offs between supporting expressive pointcut languages combined with introductions on one hand, while on the other hand offering a predictable, intuitive and non-ambiguous programming interface. Also, we want our AOP language compiler to be able to resolve dependencies whenever possible, and to detect cases where the specification is ambiguous.

The remainder of this chapter describes how the problems that we observed can be addressed, as illustrated by the aspect-oriented language *Compose**.

2.4 The use of Annotations in Compose*

In this section, we explain the use of annotations within the context of Compose*, our aspect-oriented programming language based on the concept of Composition Filters [21, 23], implemented on the .NET platform. In the Composition Filters paradigm, advice is specified by filter modules, which can be superimposed (woven) on any number of classes. The selectors (cf. pointcuts) that specify where filter modules (advices) should be superimposed are written in Prolog, using a predefined set of predicates that can be used to query the structure of a program.

2.4.1 Annotation-Based Join Point Selection

Listing 2.6 shows part of an example Compose* concern implementation handling `ObjectPersistence`. The selector `persistentClasses` (lines 5–7) selects all classes that have the annotation `PersistentRoot` attached. Selectors in Compose* are expressed using a predicate-based selector language that can query the static structure of an application. In this example, the selector has an output variable `C` that selects all program elements, provided that they are classes that have the annotation attached that is indicated by variable `A` (line 6). This variable `A` must be a program element of the annotation type that is named `PersistentRoot`, as specified on line 7. The selector `persistent-Classes` is thus equivalent to the AspectJ pointcut in listing 2.2.

```
 1
 2 concern ObjectPersistence {
 3   superimposition {
 4     selectors
 5       persistentClasses =
 6        {C | classHasAnnotation(C, A),
 7             isAnnotationWithName(A, 'PersistentRoot')};
 8   ...
```

Listing 2.6: Using annotations in Compose*

2.4.2 Superimposition of Annotations

We introduce a simple extension to the existing mechanism used to superimpose filtermodules in Compose*: a new language construct that specifies the superimposition of annotations on a set of selected program elements. The selector mechanism itself is exactly the same as the one used for superimposing filtermodules, and thus has Turing-complete expressiveness. Program elements can be selected based on their name, properties and relations to other program elements (i.e. based on the static structure of the application), including annotations.

Listing 2.7 shows the same example as used for AspectJ in listing 2.3, now implemented in Compose*:

```
 1 concern AppSpecificPersistence {
 2   superimposition {
 3     selectors
 4       transientClasses =
 5         { AnySess | isClassWithName(S, 'SessionID'),
 6                 inheritsOrSelf(S, AnySess) };
 7     annotations
 8       transientClasses <- TransientClass;
 9   }
10 }
```

Listing 2.7: Superimposition of annotations

In listing 2.7, the class `SessionID` and its subclasses are selected by the selector `transientClasses` (line 4-6). The annotation `TransientClass` will be *superimposed* (introduced) to this set of selected classes (line 8).

In principle, annotations can be superimposed on any kind of identifiable program element. In practice, there can be limitations in the underlying implementation language. In our case, .NET and Java both limit the attachment of annotations to classes, fields (instance variables) and methods. In addition, annotation types themselves can restrict the target program element types to which they can be applied. Conceptually it would be possible to annotate AOP constructs as well, such as concerns or filter modules. AspectJ does in fact support the use of annotations on e.g. aspects and advice. This topic is discussed in more detail in [107]; section 3.2.1 (regarding the language model) and 5.2 (regarding the use of annotations on AOP-specific program elements).

2.4.3 Derivation of annotations

In the previous sections, we extended the selector language of Compose* to use annotations as a selection criterion and introduced a language construct to superimpose annotations on a set of selected program elements.

These two features can be combined to achieve the derivation of annotations. Again, in listing 2.8 we show the same example as expressed in AspectJ in listing 2.4:

```
1 superimposition {
2   selectors
3     transientFields = { F |
4       typeHasAnnotationWithName(T, 'TransientClass'),
5       fieldType(F,T) };
6   annotations
7     transientFields <- TransientField;
8 }
```

Listing 2.8: Deriving an annotation

Here, the selector `transientField` selects all fields `F`, as long as they are of a type that has the annotation named `TransientClass` attached (line 3-5). The annotation `TransientField` is superimposed on these fields (line 7).

There are two big differences between the pointcut language used when superimposing annotations in Compose* and the pointcut language used when introducing annotations in AspectJ. First, Compose* supports predicates that express all (relevant) properties of program elements, as well as relations between them. For example, the predicate `typeHasField(T,F)` expresses the relation between a field and its containing type. Using this relation, we can for example select only fields in classes that have the annotation `PersistentRoot` attached, and introduce an annotation on those fields (listing 2.9 gives an example). In AspectJ we cannot express such a pointcut, as annotation introductions on fields follow the pattern: `declare @field : FieldPattern : @AnnotationToAttach()`. The FieldPattern can place restrictions on only the type and the name of the field itself (see e.g. the example in listing 2.4), and not on anything else, such as for example the type containing a specific field.

```
superimposition {
 selectors
  persFields = { F |
   typeHasAnnotationWithName(T, 'PersistentRoot'),
   typeHasField(T, F) };
 annotations
  persFields <- PersistentField;
}
```

Listing 2.9: Using relations between program elements

The second difference is the inherent expressive power of the pointcut languages. Compose* uses a Turing-complete general-purpose language (prolog) with a predefined library of predicates, whereas AspectJ uses a more strictly defined, less expressive pointcut language. Basically there is a trade-off between supporting powerful reasoning *within* pointcut expressions (here, Compose* offers more power) as opposed to reasoning *about* pointcut expressions (which is easier in AspectJ).

Clearly, the problems described in section 2.3, based on the ASpectJ examples, also apply to our annotation introductions in Compose*: there can be dependencies between the superimposition of annotations and the selectors used to specify these superimpositions, because these selectors can also be based on annotations. In the next section we analyze these problems.

2.5 Problem analysis and Approach

As discussed in the problem statement, introductions based on expressive pointcut languages can cause issues related to the order in which these introductions have to be applied. Specifically, the existence of dependencies between introductions can cause ambiguities, i.e. when different orderings of applying introductions leads to different results.

In this section, we analyze concrete cases where such problems occur, and introduce our approach to solve these problems.

2.5.1 Detecting dependencies

Preferably, we want to address the ordering problems described in the problem statement for the most general case, i.e. for a Turing-complete pointcut language, such as the one used in Compose*. Although such a language allows for powerful reasoning in selector expressions, it makes it impossible to reason statically about the outcome of their evaluation. The cause is that an expression in a Turing-complete language is, in general, not statically decidable, i.e. we cannot (generally) predict the possible results without evaluating the expression. We give a small example to demonstrate this.

In the context of .NET, annotation types are represented as normal types. This means it is also possible to build (inheritance) hierarchies of annotations. For example, we can create a generic annotation type called `PersistentRoot`, and another annotation type `XMLPersistentRoot`, extending `PersistentRoot`. In listing 2.10, we show how to select all classes that have *any* kind of `PersistentRoot` annotation attached.

```
ClassPersistent =
 { C | isAnnotationWithName(PRAnnot, 'PersistentRoot'),
     inheritsOrSelf(PRAnnot, AnyPRAnnot),
     classHasAnnotation(C, AnyPRAnnot) };
```

Listing 2.10: (in)visibility of dependencies

The problem with this selector is that its evaluation may depend on the existence of a number of annotations, i.e. in this case any annotation that extends the annotation `PersistentRoot`. However, by looking at only the source code of the selector expression, we cannot generally predict which annotations will become bound to the free variable `AnyPRAnnot`. Therefore, we cannot infer (from statically analyzing the selector expression) that there exists a dependency between this selector expression and the superimposition of e.g. the annotation `XMLPersistentRoot` somewhere else.

However, we can look at the *results* of selector evaluation; if the result of evaluating a selector changes after we have superimposed a certain annotation, there obviously exists a dependency. The reverse is not true: the fact that the result does not change after superimposing an annotation does not imply that there is no dependency. It just does not occur given the current combination of selectors, program elements and annotations in the application under consideration. In other words, by performing the evaluation of the selectors, we can observe when dependencies occur, but we cannot detect *potential* dependencies that are independent of a particular application.

The approach we propose to determine a correct order of evaluation is based on trying all possible orders of evaluating the selectors and superimposing annotations, and then observing the results. This approach can also solve the problem of multi-level (or even circular) dependencies, as we explain in the next section. In section 2.7 we discuss alternative approaches that have been proposed in related work.

2.5.2 Circular dependencies

Circular dependencies between introductions may occur intentionally. For example, consider the combination of the following two rules (taken from [37]): (1) If a class contains a public method that has the annotation *WebMethod*, the class itself should have the annotation *WebService*, and (2) vice versa, if a class has the annotation *WebService*, all of its public methods are *WebMethods* that should be annotated accordingly. Both rules can easily be expressed by annotation introductions in Compose*. However, since these rules depend on annotations that are introduced based on the other rule, iteration over those introductions is required.

This problem can be addressed by iterating over the evaluation of selectors and the introduction of annotations, until a fixpoint is reached. That is, the algorithm iterates over the introduction of annotations until the state (the set of selected program elements per selector) does not change between two iterations. An annotation can only be superimposed once on each program element—if it is attached a second time in a later iteration, the results are considered idempotent. In each iteration step, the superimposition of an annotation is performed and the selectors are reevaluated to reflect the changes caused by the superimposition. Because the resolution is based on the reevaluation of selectors between the iterative application of introductions, we do not need to know where dependencies occur beforehand.

Circular dependencies may also occur unintentionally, for example by combining several aspects that derive annotations from the existence or absence of other annotations. In certain cases such circular dependencies may cause infinite loops within an iterative resolution algorithm. To illustrate such a case, we show an example in listing 2.11. In this listing, we express the rule that the annotation *PersistentRoot* should be introduced on classes that (1) do not already have this annotationand (2) have at least one field that should be made persistent.

```
selectors
 isPersistentRoot = {C |
   not(classHasAnnotationWithName(C,'PersistentRoot')),
   classHasField(C,F),
   fieldHasAnnotationWithName(F, 'PersistentField') };
annotations
```

```
 isPersistentRoot <- PersistentRoot;
```

Listing 2.11: Example of a circular dependency

The problem with this rule is that, in an iterative approach, after executing the superimposition of the annotation *PersistentRoot* on classes that matched the selector, we have to reevaluate the selector *isPersistentRoot*. However, after reevaluation it will no longer match the classes on which we just introduced the annotation *PersistentRoot*, because it no longer fulfills the first condition. Because the rule on which the superimposition was based is now violated, a declarative approach would suggest that the superimposition should not have been executed in the first place. However, if we would ‘roll back’ that decision, the rule would apply again. Hence, we would create an infinite loop caused by what we will call *negative feedback* between selectors and the process of superimposing annotations. In this case it is relatively easy to recognize the problem, as the selector and superimposition are specified in one place; but in practice, the selector and superimposition specification may be distributed over different aspects.

This example demonstrates that we cannot ensure that iteration over selectors and superimposition of annotations will terminate, given the occurrence of *negative feedback* - that is, introductions that depend on other annotations being *absent*, while these annotations are subsequently also introduced (possibly in a completely different location).

However, many pointcut languages do support exclusion operators, as this can be very useful. A good example (that can be easily expressed in both AspectJ and Compose*) would be to specify that fields that do *not* have the annotation *TransientField* automatically get the annotation *PersistentField* superimposed. However, as we will show in the next section, a combination of such rules (even though they can easily be expressed in existing languages) can even lead to specifications that are inherently ambiguous (in the absence of explicit ordering constraints). Obviously, we would like to detect the occurrence of such situations.

2.5.3 Ambiguous selector specifications

The existence of exclusion operators and dependencies in the selector language leads to another problem: we prefer superimposition specifications and selectors to be declarative. Thus, their specification should not imply any ordering of superimposition – in particular, since introductions may be specified by different modules (aspects), and we prefer not to have to specify explicit composition ordering constraints between modules, for reasons we discuss in detail in section 2.7. However, different orders of evaluating the selectors and executing the superimposition of annotations do exist. As a consequence, this may result in different sets of program elements with different annotations attached. If this happens, the concern specification is ambiguous and non-declarative. As an example, consider the ambiguous combination of rules specified in listing 2.12.

```
1 selectors
2  notTransient = { F |
3   not(fieldHasAnnotationWithName(F, 'TransientField'))};
4  notPersistent = { F |
5   not(fieldHasAnnotationWithName(F, 'PersistentField'))};
6 annotations
7  notTransient <- PersistentField;
8  notPersistent <- TransientField;
```

Listing 2.12: Ambiguous selector specification

Listing 2.12 lists two rules: if a field is not transient, it should be made persistent, and vice versa. These two derivation rules express the design intention that each field in a program should be marked either transient or persistent. However, the design intention is not expressed *precisely* in this example: suppose there exists a field that has neither annotation *TransientField* nor *PersistentField* attached. This field is then selected by both selectors, *notTransient* and *notPersistent*. If we superimpose the annotation *PersistentField* on it first (line 5) and then reevaluate the selectors, the field no longer matches selector *notPersistent*, so the annotation *TransientField* will not be attached. However, if we first superimpose the annotation *TransientField* (line 6), the field no longer matches the selector *notTransient*, so the annotation *PersistentField* will not be attached. In this case, the end results are different depending on which introduction is executed first. The specification is syntactically valid, but also clearly ambiguous, as there is no way to discern which order of applying the introductions was intended by the programmer.

This issue can be resolved by (1) forbidding the use of such ambiguous specifications, or (2) by letting the programmer specify an ordering that would resolve the problem. In either case, it is necessary to be able to *detect* the occurrence of such issues. Given solution (1), it is unreasonable to expect that programmers would never make such mistakes, and given solution (2), it is unreasonable to always specify a complete module ordering, even in cases where the ordering makes no difference as to the results. Thus, given either approach, it is necessary to implement mechanisms that detect situations where the composition cannot be resolved unambiguously, such that the compiler can generate error messages, or to warn the programmer that an ordering specification should be provided.

2.5.4 Summary of the problem analysis

Based on the observations made in the previous subsections, we draw the following conclusions with respect to the ordering problems:

- An important issue is that selectors and superimposition should be declarative, which means that the order of attaching annotations should not matter (otherwise, the specification is imperative rather than declarative). Implying an ordering compromises the declarative nature of selector specifications, which leads to problems regarding evolvability: introducing additional selectors may change the implied ordering. In addition, this makes it harder for programmers to understand what is actually selected by a particular selector, as this may depend on statements in seemingly unrelated modules.
- The expressiveness of the selector language in Compose* does not allow static reasoning about dependencies based on the textual representation of the selectors. In addition, there can be multiple levels of dependencies (including circular dependencies).
- Iterative resolution of dependencies will not terminate (i.e. it leads to infinite loops) in cases where a circular dependency occurs in combination with an exclusion operator. We illustrated such a case in section 2.5.2.

These conclusions determine the core scope of our solution proposal: an algorithm that considers all different orderings in which annotations can be superimposed, and iterates over every possible ordering, until the set of selected elements for each selector reaches a fixpoint. To address the problem of infinite loops, we disallow the occurrence of negative feedback between selectors and superimposition of annotations. This means that selecting based on the *absence* of an annotation that is attached by another aspect will be considered an error, as this causes negative feedback (hence, the case described in listing 2.11 would be detected as problematic). Note that this does not directly limit the expressiveness of the selector language itself: it is still possible to use ‘not’ and other types of exclusion constructs. However, it does limit the possibilities of introductions, as it is no longer allowed to introduce annotations that are used to *exclude* program elements in selectors.

If such *negative feedback* occurs, the programmer has two options to make the specification declarative (apart from changing the design): (1) do not derive annotations that are used as part of exclusion conditions in selectors, but attach them manually (in the source), or (2) do not base the exclusion condition in the selector on a derived annotation, but directly on the (static) conditions that where also used to specify the derivation of the annotation. Both solutions are workarounds that do not yield very elegant code, but can at least always resolve the issue. The discussion section explains why we think the alternative (i.e. specifying an ordering to resolve the problem) is less attractive.

Additionally, we disallow ambiguous specifications by considering the occurrence of different results based on different orders of attaching the annotations as an error. To detect such cases, our algorithm tries every possible ordering of introducing the annotations.

In fact, we observed that the second restriction we impose on the use of introductions (i.e. different orderings are not allowed to render different end results) may be superfluous, because we observed that by disallowing negative feedback, the order of superimposing the annotation can never lead to different end results. In other words, cases that would be detected as being an ambiguous specification would always involve negative feedback (and already be detected as such). This would mean that just disallowing (and detecting) negative feedback is actually sufficient to ensure the declarativeness of selectors in Compose*. Proving that this observation is true for all cases is one of our future works. In terms of implementing an iterative algorithm this observation does not make a big difference: checking for negative feedback already involves trying all the possible orderings of introductions.

Finally, it is important to note that the problems related to negative feedback and circular dependencies can occur in any sufficiently expressive pointcut language, even though it may not be Turing-complete. For example, AspectJ supports introductions based on the *absence* of other elements (i.e. a ‘not’ operator), as well as dependencies on other introductions. Such a pointcut language is sufficiently expressive to share the problems described in this chapter, although more efficient or easier solutions may be viable if it is possible to reason about dependencies based on the textual representation of pointcut expressions.

In the next section we explain the algorithm used to resolve dependencies and to apply introductions.

2.6 Dependency algorithm

This section describes an algorithm that implements the iterative resolution of dependencies, as discussed in the previous section. This means it tries every possible ordering of superimposing the annotations specified in each concern source, while checking for negative feedback and ambiguous end results.

To describe this algorithm, we define a few terms that are used to describe the algorithm more precisely:

- **(Superimposition) selector**: a selector expression (e.g. $S = \{C| isClass(C)\}$) that returns a set of program elements when evaluated
- **Selector result**: a set of program elements selected by a superimposition selector (i.e. the result of evaluating a selector)
- **(Superimposition) action** (e.g. $S \leftarrow A$): the act of attaching a specific annotation (A) to a set of program elements (selected by S)[4]. An annotation can only be attached once; if a program element already has the annotation A attached, it will not be attached a second time by executing a superimposition action.
- **Iteration**: in every iteration step, exactly 1 superimposition action is executed. All selectors are then reevaluated, rendering new (possibly different) selector results.
- **Negative feedback**: occurs when for any selector result there exists a program element that was selected in iteration i-1 but not in iteration i. (For i=0 this is never the case, as by definition nothing has been selected before the first iteration.)
- **State**: the current set of selected program elements for each selector, and a list of actions executed to reach this situation.
- **Endstate**: a state where the execution of any superimposition action will not change any of the selector results.

[4]In this context superimposition is always about annotations.

2.6.1 Inputs, outputs, variables

To describe the algorithm, we define its inputs and outputs first.

Inputs are modeled as follows:

$$
\begin{aligned}
selectors &\triangleq [s : selector]\\
actions &\triangleq [a : action]\\
programAST &\triangleq [programElement : element | subTree : programAST]\\
\\
selector &\triangleq (id : identifier, expression : predicate)\\
predicate &\triangleq (expr : prologExpression \rightarrow ast : programAST \rightarrow\\
&\quad\; [selectedElements : element])\\
action &\triangleq (sel : selector, annotation : identifier)\\
element &\triangleq (nodeType : identifier, name : identifier, annotations : identifier)\\
\\
state &\triangleq ([(selector : identifier, [selectedElement : element])],\\
&\quad\; previousState : state, executedAction : action)
\end{aligned}
$$

In this notation, $(id : identifier, expression : predicate)$ denotes a tuple containing id and $expression$, of types $identifier$ and $predicate$, respectively. $[programElement : element]$ denotes a list containing items of type $element$, whereas $(expr : prologExpression \rightarrow resultVar : identifier \rightarrow [selectedElements : element])$ denotes a function with input parameters $expr$ and $resultVar$ and output $selectedElements$, being a list of $elements$.

Thus, the algorithm takes a list of selectors and actions as its inputs, as well as an abstract syntax tree (AST) representation of a base program. A selector consists of an identifier (selector name) and a predicate that can be evaluated against a program AST, yielding a subset of the elements in the program. Predicates are expressed in terms of Prolog queries as shown earlier in this chapter. When evaluated, they yield a set of program elements matching this predicate[5]. A (super-imposition) action specifies an annotation to be attached to elements matched by the selector indicated by this action. Finally, program elements are represented by type, name and attached annotations.

Finally, we define an additional data-type $state$, used to represent the internal state of the algorithm. For each state, the algorithm stores the elements matching each selector, the action that resulted in the current state, as well as

[5]The details of how this is implemented can be found in [69]; here, we represent the process as an opaque function that can be assumed to have no side effects.

a reference to the previous state, such that the order of executed actions can be traced back to the initial state.

The output of the algorithm consists of either:

- An error condition (exception thrown) when the algorithm detects that negative feedback occurred, or that several end states resulted in different selections of program elements, meaning the superimposition of annotations cannot be resolved unambiguously.
- The abstract syntax tree of the program, with annotations attached according to the resolved annotation introduction specifications.

2.6.2 Algorithm description

The algorithm implements a breadth-first search, starting with an initial state that represents a program structure on which no annotations have been superimposed yet. From this state, it performs all of the possible superimposition actions one by one. For each action that generates selector results different from the current state, it adds a new state to a list of states. If a state with exactly the same sets of selected elements for each selector already exists in that list, the new state is discarded instead. If any of the selector result sets shrinks by executing an action (i.e. it misses at least one element in the new state that was selected in the previous state) the algorithm stops, because this is an error condition (negative feedback between selectors). If, from a given state, none of the actions executed from that state render different selector results, that state is marked as an end state. If several end states are encountered by the algorithm, it checks whether they all have the same selector results, and throws an exception if this is not the case. After it has handled each state, the algorithm tries the next from the list of states, until there are no states left to handle.

The listing below describes this algorithm in pseudo-code:

```
1: function $dependencyAlgorithm(selectors, actions, programAST) : annotAST$
2: begin
3: $stateIterator \leftarrow 0$
4: $endState \leftarrow -1$
5: $states[0] \leftarrow evaluateSelectors(selectors, programAST, \textbf{Nil}, \textbf{Nil})$
6: while $stateIterator < \#states$ do // Any more states to handle?
7:     $currentState \leftarrow states[stateIterator]$
8:     $isEndState \leftarrow true$
```

```
 9:     for all $a \in actions$ do // From current state, try all actions
10:         $annotatedAST \leftarrow setAnnotations(programAST, currentState, a)$
11:         $newState \leftarrow evaluateSelectors(selectors, annotatedAST, stateIterator, a)$
12:         if $newState.selectedElems \neq currentState.selectedElems$ then // Any changes?
13:             $isEndState \leftarrow false$ // Selected elements differ -> current state not an end state
14:             for $sel = 0..\#selectors - 1$ do // Check for shrinking selector result sets
15:                 if $currentState.selectedElems[sel] \not\subset newState.selectedElems[sel]$ then
16:                     return $NegativeFeedbackException$
17:                 end if
18:             end for
19:             $isNewState \leftarrow true$ // Same state seen earlier? Assume no
20:             for $state = 0..\#states - 1$ do
21:                 if $newState.selectedElems = states[state].selectedElems$ then
22:                     $isNewState \leftarrow false$ // Same state already handled before; discard
23:                 end if
24:             end for
25:             if $isNewState$ then
26:                 $states \leftarrow states : newState$ // New state found, add it to list to be handled
27:             end if
28:         end if
29:     end for
30:     if $isEndState$ then // No action led to different selected elements -> end state found
31:         if $endState = -1 \vee states[endState].selectedElems = newState.selectedElems$
            then
32:             $endState \leftarrow stateIterator$
33:         else
34:             return $AmbiguousSelectorsException$ // Results differ from other end state(s)
35:         end if
36:     end if
37:     $stateIterator \leftarrow stateIterator + 1$
38: end while
39: return $setAnnotations(programAST, states[endState], endState.action)$
40: end
```

The algorithm uses two helper functions:

$$\begin{aligned}
evaluateSelectors \quad &::= \quad ([s : selector] \rightarrow ast : programAST \rightarrow \\
&\qquad fromState : int \rightarrow lastAction : action \rightarrow newState : state) \\
setAnnotations \quad &::= \quad (ast : programAST \rightarrow accordingToState : state \rightarrow \\
&\qquad execute : action \rightarrow annotatedAST : programAST)
\end{aligned}$$

The first of these functions, *evaluateSelectors*, takes a set of selectors and a program AST, as well as the current state and superimposition action most recently executed. The implementation of this function evaluates all selectors, yielding a new state. The second function, *setAnnotations*, takes a program AST and then attaches annotations according to a given state, and in addition, executes one new superimposition action. This yields a new AST with annotations according to the given state and action.

2.6.3 Example run of the algorithm

To demonstrate how the algorithm works in practice, we show an example run for a simple combination of selectors and introductions introduced earlier in this chapter.

We take the class `User` as it is introduced in listing 2.1, and assume the existence of a class `SessionID`. Now, we consider an application that combines the selectors and introductions in listings 2.7 and 2.8. In this case, the inputs for the algorithm look as follows (S=Selector, A=Action):

```
S0 : transientFields = { F |
      typeHasAnnotationWithName(T, 'TransientClass'),
      fieldType(F,T) }
S1 : transientClasses = { AnySess |
     isClassWithName(S, 'SessionID'),
     classInheritsOrSelf(S, AnySess) }
A0 : transientFields <- TransientField
A1 : transientClasses <- TransientClass
```

Listing 2.13: Example algorithm inputs

Figure 1 represents a sample run of the algorithm. The different states that the algorithm encounters are displayed from left to right. Vertically, we can see which program elements are matched by each of the selectors in a given state. The arrows indicate the transitions between states, while the labels next to the arrows indicate what action (introduction) was executed to get from one state to the other.

In the initial state, no annotations have been superimposed. The algorithm begins by evaluating all selectors. In this state, S0 (`transientFields`) does not select any fields, as there are no classes that have the annotation `TransientClass` attached. S1 (`transientClasses`) selects the class `SessionID`.

Consequently, the algorithm tries to generate new states by applying the superimposition actions one at a time. In this case, the algorithm starts by executing A0, which superimposes the annotation `TransientField` on fields

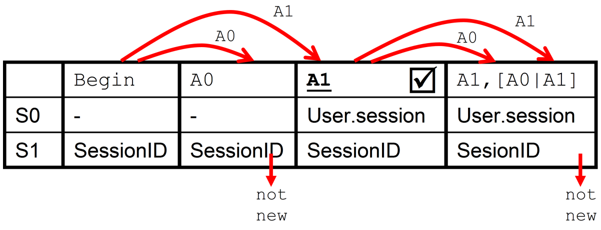


Figure 2.1: Example run of the algorithm

selected by S0. As S0 does not select anything in the begin state, the resulting state (after applying A0 and reevaluating S0 and S1) is of course identical. If a newly generated state already occurs in the list of states, we do not need to handle it a second time (therefore, it has no outgoing arrows - in practice, it is not even added to the list of states to be handled).

Next, the algorithm tries to superimpose A1, still from the begin state. After it reevaluates S0 and S1, it turns out that S0 now selects the field `User.session` - because executing A1 superimposed the annotation `TransientClass` on the class `SessionID`, and S0 selects fields that have a type with this annotation attached. From the begin state, there are no other possibilities, because we already tried to apply all possible actions (A0 and A1). This state is not an end state, as the application of A1 led to a new state.

The second state is skipped, as it is identical to the begin state. The third state repeats the same process: A0 and A1 are executed again, one at a time. It turns out both actions lead to the same, unchanged, state. Because the algorithm cannot find any action that leads to a new state, this is an end state. The fourth state is skipped, as it is not a new state. At this point the algorithm has reached the end of the list of states, which means it has generated all different introduction orderings and selector results. In this case, we can see that no negative feedback occurred anywhere during the execution of the algorithm (i.e. there is no state where any selector matches less than in its predecessor state). Also, there is only one end state, so there are clearly no ambiguities. Thus, the selector results for S0 and S1 are returned according to the situation found in the end state.

2.6.4 Efficiency of the algorithm

We chose to use a breadth-first search rather than a depth-first (recursive/backtracking) solution for several reasons:

- The calculation of a new 'state' involves evaluating all selector predicates, which is a (relatively) expensive operation. Hence, we sacrifice memory to gain speed by avoiding duplicate state calculations. We can achieve this by storing states that have been encountered already - a breadth-first algorithm already keeps such a list.

- We have to consider all (different) orderings to check whether the results are the same and to make sure that no negative feedback occurs for any possible ordering. Therefore, it does not matter that backtracking would find a first solution faster. For the same reason, the use of optimizations that would find a first solution faster (e.g. an A*-algorithm) would not make a difference.

- Breadth-first searching can cause problems related to state space explosion (resulting in excessive memory usage). However, we know that most actions will generally return the same selector results, even when using a different ordering. Because we check for duplicate states, most realistic cases are unlikely to cause such a state-space explosion, even if they involve many selectors and introductions. However, our algorithm is essentially a brute-force approach, so it is always possible to construct a worst-case scenario that consumes a lot of time and memory.

2.6.5 Termination of the algorithm

It is not obvious that the algorithm described in the previous section will terminate in all cases. In this section, we show that it does.

Non-termination could be caused by the while-loop in the algorithm. This loop has the exit condition $stateIterator < numberofstates$ (both values are positive integers). In each cycle, $stateIterator$ is incremented. However, the number of states can potentially increase repeatedly within a single cycle. This could cause the algorithm to never terminate. Therefore, we inspect the circumstances under which the number of states actually increases. There are three possible cases when executing each action within a cycle:

1) An action was executed that made at least one program element disappear from a selector result set (i.e. negative feedback occurred). This is an error condition that will terminate the algorithm.

2) An action was executed that did not change any selector result. This case will be ignored, because it has been handled already. In this case, the number of states does not increase.

3) An action was executed that added at least one program element to at least one selector result. If this results in a case that has not been handled yet (the worst case), the number of states is incremented.

Only in the third case is the number of states incremented. However, in that case we are dealing with a monotonically increasing result set (as seen over several iteration cycles). Also, there is a finite set of program elements that can be part of each selector result set (the number of program elements does not grow during the execution of this algorithm). Therefore, case 3 will eventually cease to occur, as there will simply be no program element left to add to any selector result. This means that eventually only cases 2 or 1 will occur.

Because $stateIterator$ is increased in every cycle, and eventually no new occurrences of case 3 can be found, the algorithm will always terminate eventually, when $stateIterator$ equals the number of states.

To summarize the argument:

- Each variable ranges over a finite domain, i.e., a subset of all program elements (of which there are a finite number)

- The state consists of a fixed set of variables, and therefore also ranges over a finite domain.

- The state changes in an increasing order (i.e., in a transition from one state to another, each variable in the original state contains a subset of the program elements that the same variable contains in the new state), because the algorithm immediately terminates if this is not the case.

- Since the state changes in an increasing order over a finite domain, the algorithm always reaches a fixpoint in a finite number of steps.

Note that, since the algorithm evaluates pointcuts as part of its behavior, the above argument is only valid if the termination of the evaluation of pointcut expressions can be guaranteed – this issue is however orthogonal to the use of the algorithm itself, since non-terminating pointcut evaluations would be an issue in any case, regardless of the use of this algorithm.

2.7 Related work

We discuss two kinds of related work: (1) work related to the use of annotations in the context of aspect-oriented programming, and (2) work discussing alternative solutions to the issues observed in this chapter.

2.7.1 On the combined use of AOP and annotations

The benefits of explicitly describing dependencies between annotations are described in [37]. The paper introduces a technique to describe dependencies between annotations, as well as a tool to enforce such dependency relations using a dependency checker tool. The work motivates how the concept of declaring and enforcing dependencies between annotations can be used to model and enforce domain-specific restrictions on top of a general purpose programming language such as C#. Our work focuses on the *derivation* of related annotations, by introducing a technique to not only declare relations between annotations, but also realize the automatic derivation of such relations.

R. Laddad investigates the application of meta-data in combination with AOP in [94, 95]. In these articles, he gives practical hints in what situations the application of annotations in combination with AOP (in particular, AspectJ) can be useful. In this chapter, we investigated several additional ways of using annotations in combination with AOP, e.g. by allowing the superimposition and derivation of annotations.

Several AOP-languages as well as AOP-supporting frameworks support the use of annotations. In AspectJ [14, 15] and JBoss [3, 42][6], join points can be designated based on annotations that are present at a join point shadow (i.e., a location in the source code). Similar to the superimposition mechanism in Compose*, the introduction of annotations is also supported in these

[6]JBoss itself is a framework rather than a language; however, it uses specifications written in JAXB (an XML to Java binding standard) to support the introduction of annotations. We here consider the schema of such specifications the “language”.

languages. The main differences lie in the expressiveness of the pointcut languages and the way of specifying the introduction of annotations. Both AspectJ and JBoss have only a limited language to select program elements on which annotations can be introduced. In contrast, the selector language of Compose* allows for specifying arbitrary complex queries to select program elements for introductions, as exemplified in listing 2.9. Another important difference is that the weaver of AspectJ implicitly uses the order of the declarations of introductions, whereas Compose* does not rely on any ordering information, by ensuring the declarativeness of the introduction specifications. Other examples of frameworks that support the use of annotations for purposes of addressing crosscutting concerns are Hibernate [2], an object-relational mapping framework for Java as well as .NET, and Spring [5], an enterprise-level application development framework for Java.

2.7.2 Alternative solutions

The problem of resolving multiple levels of dependencies also occurs in the domain of (automated) source code transformations, such as supported by (for example) automated refactoring tools [89]. JTransformer [4] is a transformation tool that uses a language named Conditional Transformations to specify source code transformations. The expressive power of this language to specify transformations (i.e. expressing which elements should be transformed) is intentionally limited: it allows for reasoning about dependencies between transformations based on the syntax of the transformation specification, without the need to consider the application context. This enables the detection of *potential* conflicts between transformation specifications, even if a conflict may not occur in all applications to which such a (potentially conflicting) combination of transformations could be applied. However, the use of a Turing-complete selector language in Compose* did not allow for using a similar approach. Note that there is a trade-off here: the approach of Conditional Transformations allows for detecting inherent (application independent) conflicts in the specification by offering a transformation language with a limited expressive power. On the other hand, Compose* offers an expressive selector language for superimposition, however, our dependency resolution algorithm can detect only application specific conflicts.

A radically different approach to resolving aspect composition problems is taken in [99]. Here, aspects are defined as a declaration of changes to a program (i.e. as transformations). The paper defines a clear approach for determining the order of applying such changes: a global composition specification of all aspects is to be provided, and advices at shared join points are applied in their order of appearance in the aspect definition (source code). Such an approach clearly solves any possible ambiguities and works well in situations where aspects are defined as incremental changes on top of an existing program (and on top of each other). However, to write such a global composition specification a programmer has to know about all the dependencies in an application. This can be troublesome when combining (existing) aspect libraries in a new application. It also implies that the behavior of an aspect may depend on its location in the composition specification. This reduces the ability for a programmer to understand an aspect as a module on its own, i.e. without knowing what prior transformations may or may not have influenced the pointcuts written in this particular aspect. For this reason, we prefer an approach that keeps pointcuts declarative rather than depending on an imperative ordering specification. When a pointcut specification is ambiguous in combination with other aspects or introductions, we detect this problem.

Finally, the combination of logic languages, negation and (non-)termination has been the subject of much research within the domains of logic programming and databases. There exist examples of logic languages that guarantee declarativeness and termination of predicate resolution, even in the presence of negation. However, such languages always have to impose restrictions on their expressiveness. An example of such a language is DATALOG, which is a subset of Prolog; an extensive discussion on its expressiveness, features and extensions to the language can be found in [62].

2.8 Discussion

2.8.1 On declarativeness vs. explicit ordering specifications

Our solution is aimed at keeping the specification of introductions and pointcuts declarative. An alternative is to allow precedence specifications between introductions. Even though this is definitely a solution that can be considered, it has several drawbacks: if pointcut specifications might no longer be declarative in all cases, the programmer needs to look at the context (i.e. other aspects within the same application) in order to understand what elements might match a particular pointcut expression. Such explicit dependencies can hinder the reusability of aspects. More importantly, even when using such an approach, it would still be desirable to *detect* ambiguities, as allowing for precedence specifications does not ensure that programmers will always know when it is necessary to use them. Thus, detection mechanisms such as discussed in this chapter would still be useful.

2.8.2 On the generality of the issues and solutions in this chapter

In this chapter and elsewhere [107], we, as well as other authors [87, 94], have argued that the use and introduction of annotations can have important benefits with regard to the reusability and evolvability of software modules.

Even so, the issues identified in this chapter also occur in other situations, as we discuss below.

First, we have already shown that the identified issues related to dependency resolution and the potential ambiguity of composition specifications are not limited to the use of specific tools or languages — for instance, we have shown how the same kinds of conflicts occur in both AspectJ and Compose*.

In addition, we have seen the same issues occur in the context of normal pointcut-advice specifications [56]. In the cited paper, the authors show examples where pointcuts (and hence, the execution of associated advice) depend on the execution or non-execution of other advice[7]. If circular or negative application conditions are specified between such pointcut-advice combinations, the same issues as identified in this chapter occur. The solution, described in more detail in a later paper [27], is to order aspects from high- to low levels, where dependencies are considered to occur in only one direction ("stratified aspects"). This effectively prevents circular dependencies and resolves any potential ambiguity, although at the same cost as discussed in subsections 2.7.2 and 2.8.1. Thus, it is clear that the same issues also appear in contexts unrelated to annotations or structural introductions.

In principle, our solution could also be applied to this situation, as the number of pointcuts and advices is finite, as was the case with annotations (thus guaranteeing our algorithm can run in a finite amount of time). This is assuming the pointcut matching criteria can be determined statically; although there is no inherent reason why our algorithm cannot be used at runtime based on dynamic properties, it would probably be unpractical for performance reasons.

[7]This can be done in AspectJ by using the `adviceexecution` and `within` pointcut specifiers.

2.9 Conclusion

The technology for using annotations together with AOP is readily available; several aspect-oriented languages support the designation of join points based on references to annotations. The introduction (superimposition) of annotations is not a new idea either; several aspect-oriented languages support the introduction of annotations over multiple program elements. However, as we discussed in the section on related work, these AOP languages offer relatively simple static pointcut languages to express the locations where annotations can be introduced.

In this chapter, we consider the application of an *expressive* static pointcut language for introductions. This pointcut language is a predicate-based, Turing-complete query language that allows for specifying complex queries as a means to select program elements on which annotations can be introduced. Queries can also select program elements based on the annotations that are already associated to program elements. By introducing annotations through queries that select program elements based on other annotations, we obtain the *automatic derivation* of annotations. By supporting the derivation of annotations, dependent annotations (and complete annotation hierarchies) can be automatically introduced. By ensuring the *declarativeness* of annotation introductions, we believe that we can keep the use of this mechanism as straightforward as possible for the programmers.

However, to ensure declarativeness, it is necessary to handle the dependencies between the evaluation of pointcuts and the introductions of annotations. The main contributions of this chapter are related to this issue:

- We analyzed the possible dependencies among introductions and identified cases where dependency problems may arise (section 2.5).
- Based on this analysis, we developed an approach and designed an algorithm to resolve the above mentioned dependencies, and detect the possible dependency problems (section 2.6).
- We showed that this algorithm will always terminate either by providing a correct resolution of the dependencies, or detecting ambiguity in the weaving specification (section 2.6.5).
- As a proof of concept, we have also implemented and tested our approach in Compose*[10], our platform for aspect-oriented programming language research, developed at the University of Twente.

This approach can also be applied in other aspect-oriented languages to the weaving of introductions. For instance, the latest AspectJ weaver implicitly chooses the order in which introductions are applied, in some cases depending on arbitrary criteria such as the ordering of declarations within a source file. Our approach ensures the declarativeness of introduction specifications; this means that by adopting our approach, the weaver would not need to rely on the order of these declarations.

Finally, our approach is also *generic* in the sense that it is applicable to other types of introductions, not only the introduction of annotations. For example, the selection language can be applied to introduce methods in the same way we introduced annotations. When a method is introduced and this method is referred to by another superimposition specification (i.e. a pointcut), the same dependency issues will arise that we identified at the introduction of annotations. The next chapter discusses this, among several other issues related to the use of introductions.

A Graph-based Approach to Modeling and Detecting Composition Conflicts related to Introductions

3.1 Introduction

Aspect-oriented programming languages allow for the modular specification of crosscutting concerns. To facilitate this, aspect-oriented languages offer new kinds of composition mechanisms. *Introductions* are one such mechanism, also referred to as *structural superimpositions* or *inter-type declarations* [40]. Introductions are constructs that affect the structure of a program, for example by adding a method to a class or by changing the inheritance structure. In chapter 2, we have discussed these mechanisms in the context of introducing annotations, and demonstrated their use in detail.

In this chapter[1], we focus on analyzing conflicts related to introductions. For example, introductions may have unintended effects, cause a program to fail to compile, or cause the program to be ambiguous (i.e. the program source can be interpreted by the compiler in more than one way).

To facilitate a precise analysis of such composition conflicts, we express the *structure* of Java/AspectJ programs (source code) as a graph-based model, which we refer to as *program model*. In addition, we express the *introductions* that are part of the AspectJ source code as *graph transformation rules,* which can be applied to such a program model.

To automate the analysis, we implement a tool that automatically converts the relevant parts of Java/AspectJ program sources to the above models, i.e., modeling the structure of a program as a graph, and modeling introductions as transformations on that graph. Our tool stores these models in a standardized

[1]The contents of this chapter are based on a paper published at the 6th International Conference on Aspect-oriented Software Development, AOSD 2007[73]

format, supported by an existing toolset that we use to perform the actual analysis of our models.

Specifically, we explicitly model several kinds of composition conflicts as graph matching patterns. Using a graph rewrite tool to match these patterns against our program models, we can automatically detect the occurrence of composition conflicts. The same tool is used to apply the introductions (represented by graph transformation rules) to the program model. This tool is able to detect situations in which the transformations can be applied in different orders, leading to potentially different transformed program structures.

This chapter is structured as follows: in section 3.2, we identify three categories of composition conflicts related to introductions, and illustrate these through various examples. In section 3.3, we introduce a graph-based model of introductions. In section 3.4, we use these models to detect and visualize the conflicts. In section 3.5, we discuss how aspect language developers can address or avoid particular types of conflicts. Section 3.6 discusses alternative approaches to detecting conflicts related to introductions. We finish with a discussion of related work and a conclusion.

As compared to the previous chapter, we here focus on classifying different types of issues, finding their root cause and defining a generally applicable framework for the analysis of these issues, whereas chapter 2 focused specifically on the use of annotations.

3.2 Issues related to introductions

Many aspect-oriented programming languages offer various composition mechanisms to adapt the structure of a program, for example by changing the inheritance structure or by introducing additional program elements, such as methods, instance variables or annotations. In this section, we distinguish three categories of composition conflicts that can occur when using introductions.

3.2.1 Violation of language rules

Introductions may cause violations of basic language rules. By “basic language rules”, we mean rules that are part of a language definition, either explicitly or implicitly, such as basic assumptions underlying the language. We give some examples of language rules in Java:

- Each class, apart from the predefined root class, extends exactly 1 "parent" class (single inheritance)[2].
- Interfaces may consist of method *declarations* and constants, and may not define instance variables or method *implementations*.
- A class can define exactly 0 or 1 methods with a specific method signature, potentially overriding a definition in a parent class, but never defining 2 or more distinct methods with the same signature in a single class.

Although such rules may appear as rather obvious to anyone who has ever used the language, in the presence of aspects it may not be straightforward at all to determine whether the program adheres to such rules. This is caused by the dependency inversion [108] introduced by aspects - that is, aspects may *superimpose* elements (e.g. methods) on existing program elements (e.g. classes). However, this change is not visible in the source code of the class on which the method is superimposed ("obliviousness"). Thus, (manual) inspection of the base code might not indicate any problem. In addition, we will argue why potential issues may not be obvious from inspecting (just) the aspects either.

It has been argued[3] that violations of basic language rules are already detected by the base language compiler, and hence, that an AOP compiler does not need to detect them. In addition to giving examples of such violations, in this section we explain why this is not necessarily the case. Moreover, we show how aspect language implementations sometimes break existing language rules, for example related to the type system, in ways that may confuse programmers.

Example: Multiple, conflicting method definitions

Consider the AspectJ application fragment in listing 3.1.

```
public interface Persistent { ... }

public class BusinessObject implements Persistent { ... }

aspect PersistenceImplementation {
```

[2]If unspecified, the parent class defaults to the root class, `java.lang.Object`

[3]i.e., in private communication as well as during discussions in the European Workshop on Aspects in Software 2006 (EWAS), regarding an early version of this work [70].

```
 6   void Persistent.saveChanges() { db.update(...); }
 7 }
 8 
 9 aspect ObjectCache { ...
10   void BusinessObject.saveChanges(){ cache.setVal(...); }
11 }
```

Listing 3.1: Conflicting method introductions

In this example, two unrelated aspects both introduce a method named `saveChanges` to classes that match the specified type. The aspect `PersistenceImplementation` introduces such a method on all classes[4] that implement the interface `Persistent` (line 6), whereas the aspect `ObjectCache` introduces a method with the same name on the class `BusinessObject` (line 10). In an actual application, where these definitions would likely be distributed over several files, it may not be immediately obvious that these aspects conflict with each other, because the type patterns used to introduce these methods are different, and appear unrelated at first sight. However, because class `BusinessObject` implements the interface `Persistent` (line 3), both aspects introduce a method with the same name to the same class, which leads to a naming conflict.

One might assume that such conflicts will be detected by existing (base) language compilers. However, this is the case only when aspects are woven into base-language source code (such as Java) and then compiled using an existing base language compiler (e.g. *javac*), i.e., implementing a source-to-source weaving strategy. In practice this is not always feasible, because aspect languages may introduce constructs that cannot comfortably be expressed in terms of base language source code – for example, because the aspect language extends the type system in a way that would not be accepted by the (strict) type checker of the base language compiler.
In addition, the interpretation of base language rules may be extended by the aspect language. For example, in AspectJ projects, base classes may contain calls to methods that are introduced by an aspect. This means that even though the base code needs to be *parsed* before weaving (to accommodate pointcut evaluation), the compiler-level *semantic checks* may have to be postponed until the weaving is done. So, some phases of aspect and base code compilation may become interleaved, leading to various degrees of integra-

[4]Here, we assume that the method is introduced on classes implementing the interface, rather than on the interface itself. The latter would violate the language model of Java, since interfaces are supposed to only declare methods, not implement them!

tion between aspect and base code compilers. As a consequence, the implementation of compiler-level semantic checks (i.e. enforcement of language rules such as those described in this section) may have to be reconsidered. For example, such rules should take introduced methods into account, even if no source-code level representation of these methods exists during any phase of the compilation.

Moreover, design choices made by aspect language developers may sometimes lead to language semantics that may be different from what programmers expect. We demonstrate this using the example above. According to the actual semantics as implemented by the AspectJ compiler, the introduction on the interface `Persistent` (line 6) effectively supplies a default implementation that can be overridden by base classes implementing this interface. This means the above example compiles and runs in AspectJ; the method that is declared directly on the class itself (line 10) overrides the one declared on the interface (line 6). In other words, the way in which the interface `Persistent` now functions is similar to an abstract base class, except that the programmer is not bound by the single inheritance restrictions imposed by the Java base language - as classes may implement multiple of such “abstract class"-like interfaces. We believe that many programmers will expect aspects to add behavior or (maybe) override existing base behavior; supplying a default implementation that can be overridden by a base class is at least not the semantics we would have expected. On top of this, additional issues may be caused by this *de facto* introduction of multiple inheritance in Java. AspectJ enables the use of multiple inheritance in Java to some extent, since classes may implement multiple interfaces, and interfaces can now define method *implementations* rather than only *declarations*. In addition, Java also allows multiple inheritance between interfaces. Fortunately, the AspectJ compiler generally detects the issues this may cause, as we show below.

Even though the AspectJ compiler does not consider our example to be a language rule violation, it still has to implement other (additional) language rules because of its adapted interpretation of Java interfaces. To demonstrate this, listing 3.2 shows a revision of listing 3.1. In this example, two aspects declare a method `saveChanges` on two different interfaces (lines 5 and 9). To deploy these generic aspects in our particular application, a binding aspect (line 12-15) declares that the class `BusinessObject` implements both these interfaces. This is a commonly used technique to deploy a generic aspect in a specific application context, as for example demonstrated by AspectJ implementations of several design patterns [66].

```
public interface Persistent { ... }
public interface Cache { ... }

aspect GenericPersistenceImplementation {
 void Persistent.saveChanges() { db.update(...); }
}

aspect GenericObjectCache {
 void Cache.saveChanges(){ cache.setVal(...); }
}

aspect BindingAspect {
 declare parents: BusinessObject implements Persistent;
 declare parents: BusinessObject implements Cache;
}
```

Listing 3.2: Conflicting method introductions revisited

In listing 3.2, the class `BusinessObject` now effectively inherits two competing "default implementations" of the method `saveChanges`. It is undefined which method definition should take precedence. The AspectJ compiler recognizes this situation as an error and gives a message accordingly.

Based on these observations, we conclude that the definition and enforcement of even such basic language rules is not as straightforward as one might have expected.

Example: Cyclic inheritance

To show that many existing language rules are in some way affected by aspects (or, more specifically, introductions), we show some more examples. Consider accidentally declaring a circular inheritance structure as in listing 3.3:

```
public class Ellipse extends Circle { ... }

aspect CircularShapes {
 declare parents: Circle extends Ellipse;
}
```

Listing 3.3: Cyclic inheritance (caused by an aspect)

When AOP languages support a construct to change the inheritance structure, it is possible to define a circular inheritance structure (using that construct). However, OO languages demand that the inheritance structure does not contain cycles. It is interesting to note that AspectJ defines an even more strict language rule, which also effectively prevents an aspect from introducing circular inheritance (given that we already know that the base program does not have circular inheritance itself). The AspectJ compiler enforces the following rule: given that `A extends B`, it is only possible to declare `A extends C` (thus *overwriting* the original superclass of A, as Java does not support multiple inheritance!) when C is itself a subclass of B. Clearly, this rule is always broken when we try to introduce a cycle in the inheritance tree (i.e. the AspectJ compiler detects this as an error). Thus, we see that aspect-oriented languages may also introduce additional language rules.

Example: Extending a final class

As another example, it is not allowed in Java to inherit from a *final* class. Final classes are sometimes used in libraries to prevent applications from accessing protected fields or methods (they could do so by extending a library class); the AspectJ compiler applies this rule also to the `declare parents` construct, preventing programmers from using it to extend final classes. E.g., when compiling listing 3.4, the AspectJ compiler yields the error message *“cannot extend final class AccessControl”*.

```
public final class AccessControl {
  public boolean hasAccess(User u) { ... }
}

aspect CircumventProtection {
  // Would allow class MyAC to override method hasAccess(..)
  declare parents: MyAC extends AccessControl;
}
```

Listing 3.4: Attempt to extend a final class using “declare parents”

To conclude, the examples in this section have shown how basic object-oriented language rules can be violated by introductions: first, a class cannot contain two distinct program elements (e.g. methods or fields) with the same name/signature. Second, we usually do not want to “break" existing language mechanisms such as inheritance or the finalization of classes. Note that the kind of rules mentioned in this section are in principle language specific, although the examples mentioned here apply to most object-oriented languages. We have shown that developers of aspect languages have to make non-trivial decisions as to the exact (re)interpretation of existing language rules in the presence of aspects.

3.2.2 Unintended effects of introductions

There exist situations where an introduction does not break any (base) language rules, but still has unintended effects. Consider the example in listing 3.5. Here, the aspect `Printing` introduces a method named `getSize` on the class `AlertDialog`. However, this introduction has another effect: it overrides a method with the same name, which the class `AlertDialog` already inherited from its parent class `DialogWindow`.

```
public class DialogWindow {
  public Rect getSize() {
    // return window dimension..
  } ..
}

public class AlertDialog extends DialogWindow {
  public AlertDialog(String alertMsg) {..}
}

aspect Printing {
  public Rect AlertDialog.getSize() {
    // return paper dimension..
  } ..
}
```

Listing 3.5: Method introduction overrides an existing method

There is a difference between this kind of conflict and the previous category: in this case, there is no inherent technical reason (such as violation of language rules) why this composition would be invalid. However, the implicit effect of effectively overriding an existing method may be unintended and undesired. A compiler or checking tool cannot generally judge whether overriding is deliberate or unintentional. Even so, a compiler should preferably flag such situations and issue a warning, as many compilers already do in similar situations that may indicate programmer errors. In section 3.5, we discuss this issue in more detail.

3.2.3 Ambiguous aspect specification

A third kind of composition problem that we study is caused by composition specifications referencing *and* modifying the same program model. The issues that result from this where discussed in detail in chapter 2, at least within the context of annotations. We briefly summarize the discussion here, and show how the same kind of issue can occur when using method introductions.

When using introductions, the program structure is changed by the application of introductions, but also “queried" by pointcut designators. It is definitely possible that a pointcut refers to the same program elements that are also changed or introduced by an introduction. Therefore, introductions can influence the composition as specified by (other) pointcuts.

```
1 declare parent: BusinessObject implements PersistentRoot;
2
3 pointcut persistence():
4   execution(* PersistentRoot+.*(..));
```

Listing 3.6: Pointcut depends on an introduction

Listing 3.6 specifies a pointcut (line 3+4), which selects all classes that implement the interface `PersistentRoot`. However, classes can be adapted to implement this interface using the `declare parents` construct (line 1). This way, the pointcut depends on this change to the program structure. This can be an intended effect, but such dependencies can also lead to ambiguous code. A simple example of such a case is given in listing 3.7.

```
 1 /* ..all types defined in package MyApp.. */
 2 public interface Persistent { ... }
 3 public interface SensitiveData { ... }
 4
 5 public class User {
 6   private String pwd;
 7   ...
 8 }
 9
10 aspect SensitiveDataHandling {
11   declare parents:(MyApp.* && !Persistent) implements SensitiveData;
12
13   void SensitiveData.clear() { ..clear values.. }
14 }
15
16 aspect PersistenceHandling {
17   declare parents: User implements Persistent;
18 }
```

Listing 3.7: Ambiguous introduction

In this example, a method `clear()` is introduced on all classes that are supposed to contain sensitive data (line 13). Line 11 indicates the classes concerned: all classes that are part of our application package, but do *not* implement the interface *Persistent*, are supposed to contain sensitive data. However, another aspect may declare specific classes to be persistent (line 17). In this case, the specification is ambiguous, because the order in which the two parent declarations can be evaluated and applied lead to different results. Whether or not the class *User* should implement the interface *SensitiveData* depends on the arbitrary[5] choice whether the declaration in line 17 is applied before or after the declaration in line 11. Hence, we cannot determine whether the method `clear()` should be introduced on the class `User`.

3.3 A graph-based model of introductions

In the previous section, we identified several types of conflicts related to introductions. To facilitate a precise analysis of such conflicts, we first present a concrete and precise model of introductions.

In general, we can say that a composition construct involves two parts: a selection (what to compose, e.g. two objects) and an action (how to compose, e.g. by sending a message from one object to the other). In the case of aspect-based composition, we can think of the selection mechanism as pointcuts or structural patterns (such as type patterns in AspectJ), whereas the actions in AOP terminology correspond to e.g. superimposition of advice or (structural) introductions (see [105], chapter 2 for a detailed reference model of AOP constructs). In the remainder of this chapter, we use this simple model of composition for the analysis of structural introductions in AspectJ.

3.3.1 Graph-based program representation

To analyze the effects of introductions, we first need to model the elements to which introductions can be applied, i.e. a concrete program model.

In this section, we define a simple mapping of program structures to a graph-based representation. Next, we model introductions as transformation rules on such graphs. In the next section, we will use these mappings to analyze the identified types of composition conflicts in detail.

Figure 3.1 shows a graph-based representation of (relevant parts of) the structure of the program in listing 3.1. This graph is constructed from the source code according to a simple meta-model of program structures: programs are modeled as consisting of *program elements* of a specific *kind* and with a specific *name*. Program elements can be related to other program elements through *named relations*.

[5]i.e. no explicit ordering constraints are specified

Two concrete examples of program elements are “Class User” and “Method getUserName()”. These program elements may be related by a “hasMethod” relation, indicating that the class implements (“contains”) this method.

A program that is represented using the above meta-model can be mapped to a graph-based representation that consists of the following kinds of elements:

- *id*-nodes, defining a unique object identifier for each program element
- *kind*-nodes, representing a kind of program element, e.g. *Class*, *Method*, *Interface*, *Aspect*.
- *name*-nodes, representing a program element name or signature, e.g. *Persistent*, *saveChanges*, *ObjectCache*.
- *isa*-edges (between *id* and *kind* nodes), indicating the type of a specific program element node.
- *named*-edges (between *id* and *name* nodes), indicating the name of a specific program element.
- Any other edges between two *id* nodes, representing relations between program elements. These edges are labeled using the name of the relation, e.g. *implements*, *extends*, *hasMethod*, etc.

A program is mapped to this graph representation as follows: each program element to be represented is mapped to a node with a unique identity (node label), e.g. *method1*, *method2*, *class1*. These generated labels are used to uniquely identify that node. Each program element node has (at least) two outgoing edges: one edge labeled *isa*, pointing to a node that represents the kind of program element (e.g. class or method), and an edge labeled *named*, pointing to a node that represents the name of this element. If several program elements have the same name, their *named* edges point to the same node. If nodes are of the same kind, their *isa* edges also point to the same node. All other edges model the relations between program elements in the given language model. For example, class nodes may have *implements* relations (edges) to interface nodes.

To demonstrate the mapping, the left-hand side of figure 3.1 represents the AST of the base program in listing 3.1. It contains only two program element

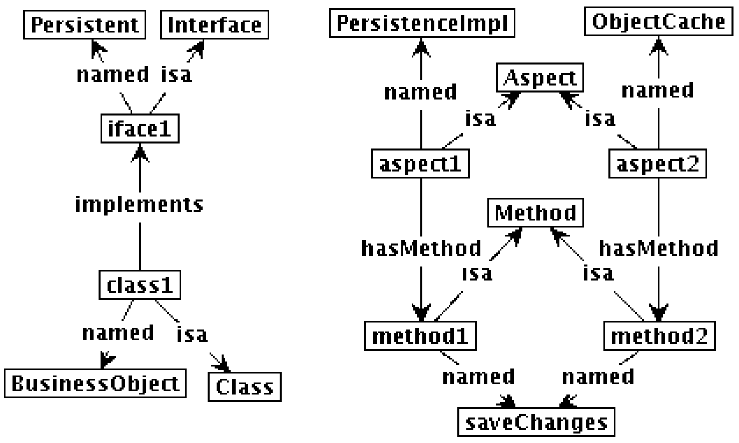


Figure 3.1: Graph representation of the program in listing 3.1

nodes[6], labeled *iface1* and *class1*. These nodes have edges to nodes representing their name and kind, and in addition, node *class1* (BusinessObject) has an *implements*-relation to node *iface1* (Persistent).

Similarly, the right-hand side shows a representation of the structure of the aspects defined in listing 3.1. Note that both aspects have edges labeled *hasMethod* to distinct method-nodes (as these methods are distinct program elements). However, both method-nodes have a *named* edge that points to the same *name*-node.

We wrote a prototype of a tool that automatically maps (relevant) parts of Java/AspectJ programs to this representation. As the model imposes no restrictions on the particular types of relations or kinds of program elements, it should be straightforward to create mappings from other languages, including UML class diagrams.

3.3.2 Introductions as graph transformations

The graph-representation presented above was chosen because it enables us to express the selection part of introductions (type patterns, pointcuts) as simple graph matching patterns.

[6]for simplicity, we show only the nodes that are relevant to the explanation of our examples. This applies to all diagrams in this chapter.

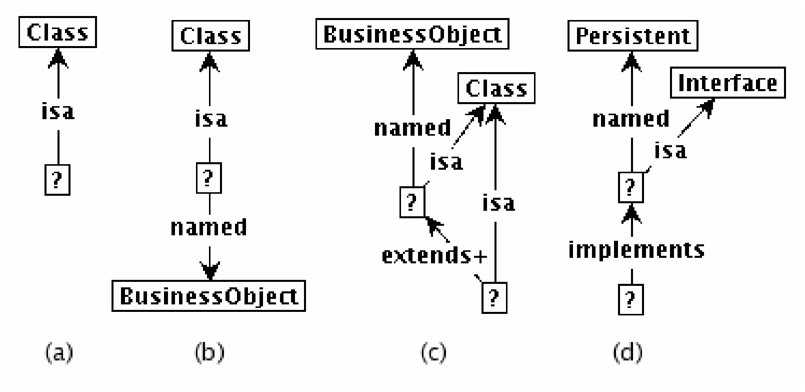


Figure 3.2: Examples of graph matching patterns

Figure 3.2 contains four examples of such matching patterns. Given a program graph, the graph matching pattern in figure 3.2a matches all subgraphs consisting of nodes that have an *isa*-edge to the node labeled *Class*. In other words, it matches all program element identifier nodes that represent a class in the system under consideration. A question mark in a graph pattern means the label of that node or edge is a “don’t care" in the matching process (i.e. it may contain any value). A single graph matching pattern can potentially match multiple distinct subgraphs within a single program graph; the pattern discussed above yields a subgraph match for *each* class node that is found in a given program graph.

The pattern in figure 3.2b selects subgraphs consisting of nodes that have an *isa*-edge to `Class`, and a *named*-edge to `BusinessObject` - i.e. it selects the class `BusinessObject`. The pattern in figure 3.2c extends this example, by selecting subgraphs consisting of nodes that can reach the `class BusinessObject` node through one or more *extends*-edges, i.e. any subclasses of `BusinessObject`. The plus sign here signifies one or more occurrences of an edge with the given label. Finally, the pattern in figure 3.2d selects any node that has an *implements*-edge to the node representing the interface `Persistent`, i.e. it selects all classes implementing that particular interface.

To model a complete introduction specification, we use a single graph that specifies a selection (a matching pattern as described above) as well as an action (the actual introduction).

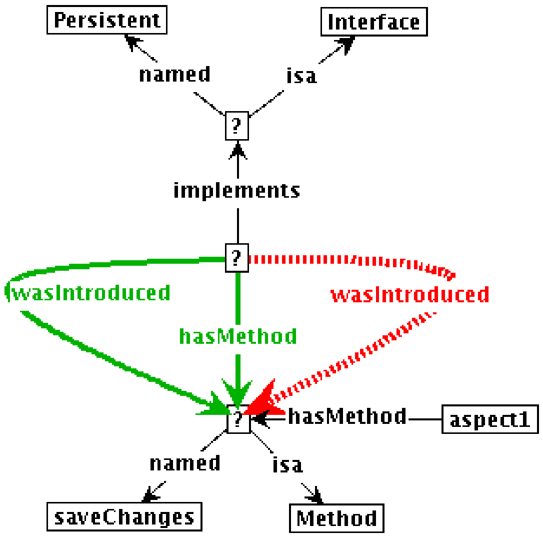


Figure 3.3: Transformation: method introduction (1)

Figure 3.3 shows the composition specification rule that corresponds to line 6 of listing 3.1. In this figure, all the nodes and edges with solid lines together specify a selection pattern as described above. This example specifies the type pattern `Persistent`, i.e. it selects all elements that implement the interface `Persistent`, as in figure 3.2d. Also, we select a node named `saveChanges`, which is a `Method`, and is contained (`hasMethod`) by the program element `aspect1`, which is the unique identifier of the aspect that declares this introduction.

The thick gray edges are not part of the selection pattern, but specify the action (introduction) that should be executed when this rule is applied. Here, we specify the introduction of an edge labeled `hasMethod` between the selected class(es) and the method defined within the aspect. Finally, the dotted edge specifies an *embargo*. In the graph, there must *not* exist an edge labeled `wasIntroduced` between the selected nodes. As part of the transformation, we introduce an edge with this label. This prevents the same introduction from being applied at the same location more than once; i.e. after we perform the introduction, the rule will not again match the same location in the transformed program model.

This pattern matches the program model of listing 3.1, as shown in figure

3.4 - the nodes and edges involved in the match are in bold-face in this figure.

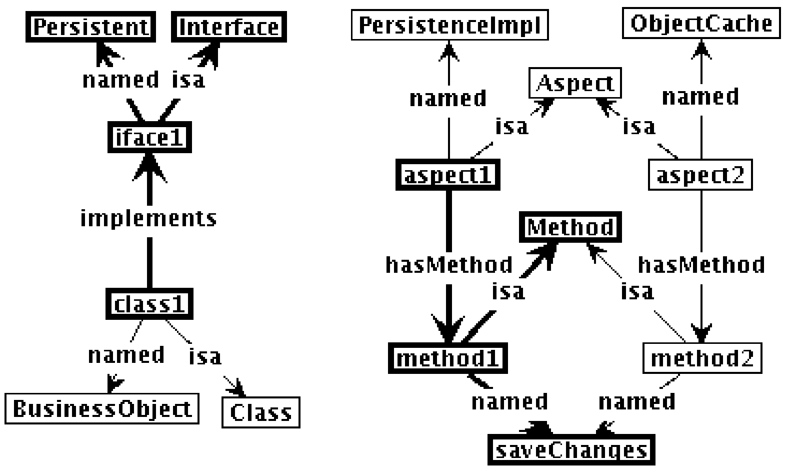


Figure 3.4: Selection: matching program elements

After application of the rule in figure 3.3, the program model from figure 3.1 is transformed into a new "state", as depicted in figure 3.5. The edges between nodes `class1` and `method1` have been added by application of the introduction rule.

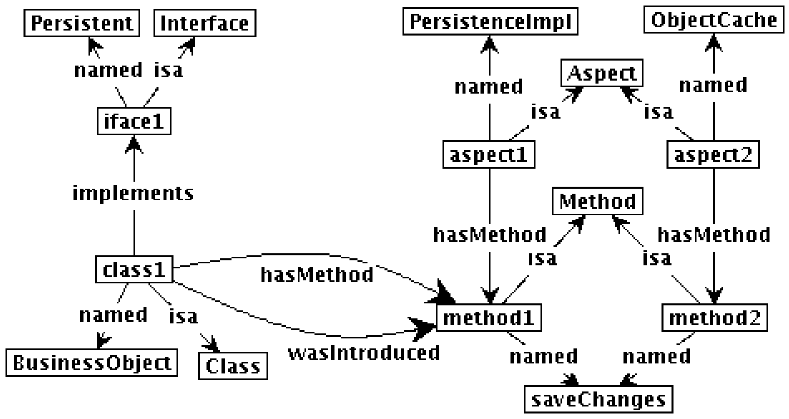


Figure 3.5: Introduction: applied to the model from figure 3.1

3.3.3 Automatic model generation and conflict detection

The diagrams in this chapter are visualizations made using the Groove (Graphs for Object-Oriented Verification) tool set [63, 114]. We explain the functionality of these tools as they are used in this chapter.

The *Groove Editor* can be used to create graph representations, e.g. of program models and transformation rules as described above. The graphs are stored as XML files using the GXL (Graph eXchange Language [77]) format - a (de facto) standard used by many graph-based tools. As there exist standard libraries to read and write GXL files, it is easy to create tools that generate graphs in this format.

We have implemented a research prototype of such a tool that automatically maps the structural model of Java/AspectJ programs to program graphs (as in figure 3.1). In addition, it can map several kinds of AspectJ introductions to graph transformation rules (as in figure 3.3). This tool is available for download [68].

The *Groove Simulator* implements a single-pushout graph rewriting algorithm with negative application conditions[7]; i.e. it can execute the matching and application of transformation rules (introductions) such as those presented above. Given a begin state (a program model such as figure 3.1) and a set of transformation rules (representations of introduction constructs such as figure 3.3), the simulator tries to match the transformation rules. Each transformation (introduction) is then applied for each match found in the current state. Each transformation can lead to a new state (modified program representation). The simulator can explore the state-space of transformation applications, in other words, it can generate all orderings of matching and applying a given set of transformation rules. It detects states that are isomorphic, i.e. have the same configuration of nodes and edges. Optimized algorithms are used to ensure that graph matching and duplicate state detection can be done in polynomial time in most cases.

Although in this chapter we show graph visualizations generated using the graphical user interface (GUI) of the Groove Simulator, this tool can also be run in command-line interface (CLI) mode, not showing any visible representation of the graphs whatsoever. This would be useful when integrating conflict detection in compilers or checking tools. To implement this, detected conflict patterns would have to be converted to a textual error message which is linked back to the original source code. This approach, based on the inte-

[7]For a detailed explanation of the algorithms and the tool itself, please see [114].

gration of Groove analysis in existing compiler technology, has already been successfully implemented in the context of semantic conflict detection [48].

To demonstrate the use of the Groove simulator, we complete the representation of listing 3.1, including the application of both introductions listed in that example.

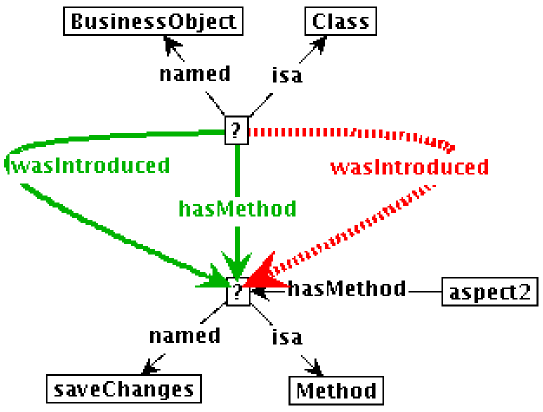


Figure 3.6: Transformation: method introduction (2)

Figure 3.6 represents the selection pattern in listing 3.1, line 10. It is very similar to figure 3.3, but directly selects the class `BusinessObject` instead of referring to an interface.

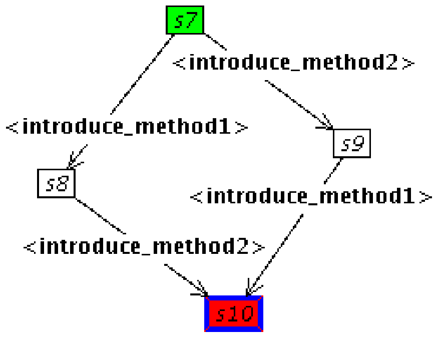


Figure 3.7: Transformation state space: applying the introductions in list. 3.1

Figure 3.7 shows a state-space exploration of our example case. In this figure, each node represents a particular state of the program model. Node s7 corresponds to figure 3.1, which is the initial program state, before any introductions have been applied. For every transformation rule that is applicable in this state, figure 3.7 contains an outgoing edge. In this example, two transformations (introductions) are applicable from the initial state: the edge labeled <introduce_method1> denotes the application of the rule depicted in figure 3.3, whereas the edge labeled <introduce_method2> denotes the application of the rule in figure 3.6. Node s8 refers to the state of the program model as depicted in figure 3.5. In this state, only the transformation depicted in figure 3.6 is applicable, since the other transformation pattern that was applied on the transition from s7 to s8 prevents a method from being introduced at the same location twice (this is caused by the wasIntroduced embargo edge in figure 3.6, which was added to the program graph when that transformation was applied).

In this case, we see that the subsequent application of both introductions eventually leads to the same (=isomorphic) state (program model), independent of the order in which they are applied. The final state s10, in which both introductions have been applied, is depicted in figure 3.8. In this state, no more introduction rules can be applied.

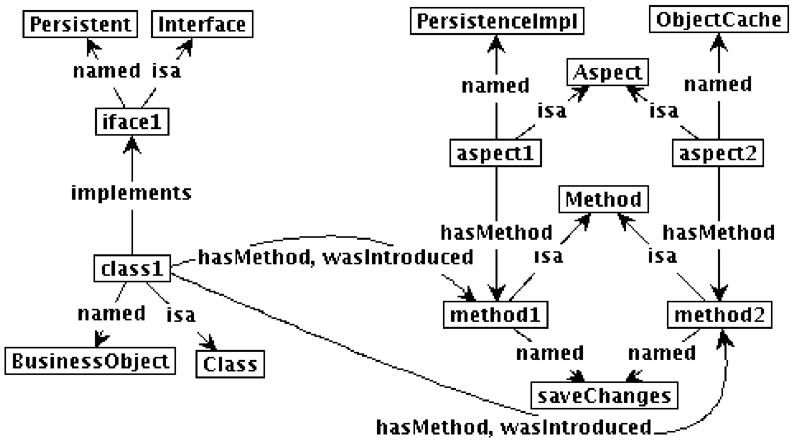


Figure 3.8: Final program model after applying introductions

We can use the diagram shown in figure 3.7 for two purposes: first, because there is exactly one final state[8], we can conclude that any order of applying the introductions led to the same result in this case. This means the program is unambiguous. Second, we can inspect each state for the occurrence of conflicts. The next section discusses this in detail.

3.4 Analysis of conflicts related to introductions

In this section we use the graph representation of compositions as described above to visualize examples of different types of composition problems.

3.4.1 Violation of language assumptions

As we have observed in section 3.2.1, the example in listing 3.1 violates a basic language assumption[9]. By expressing the violation of such language assumptions as matching rules over the program model, we can detect and visually represent the exact location of the problem. Figure 3.9 depicts a rule that matches violation of the first rule mentioned in section 3.2.1: if the program model contains a program element that is a class, which has two *distinct* method elements that have the same name, it violates this language assumption. In this diagram, the dotted (red) line labeled '=' means that the nodes connected by this edge must be distinct nodes in the graph (i.e. the two nodes must not have the same identity).

Figure 3.10 shows that this rule indeed matches in the final state of our previous example (see figure 3.8). We can see exactly which elements are involved in the conflict: the involved elements and edges are represented in bold in the figure. Also, we can trace back (by looking at figure 3.7) which combination of introductions led to the matching of this rule, and are thus involved in causing the conflict.

It is possible to define such rules for all kinds of language assumptions - which can often be found in or derived from the language specification. As another example, figure 3.11 depicts a rule that matches circular inheritance between classes - another type of language rule violation mentioned in section

[8]To be exact, all final states are isomorph graphs, i.e., they all represent the exact same structural model of the program.

[9]In this chapter, we assume that methods introduced on interfaces should not be overridden by implementation classes. As noted before, this is slightly different from actual AspectJ semantics.

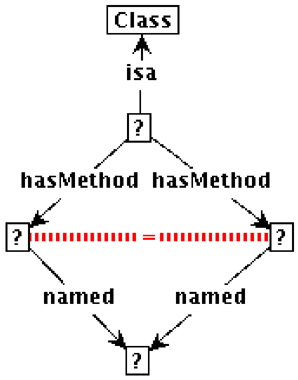


Figure 3.9: Violation rule: Naming conflict/double definition

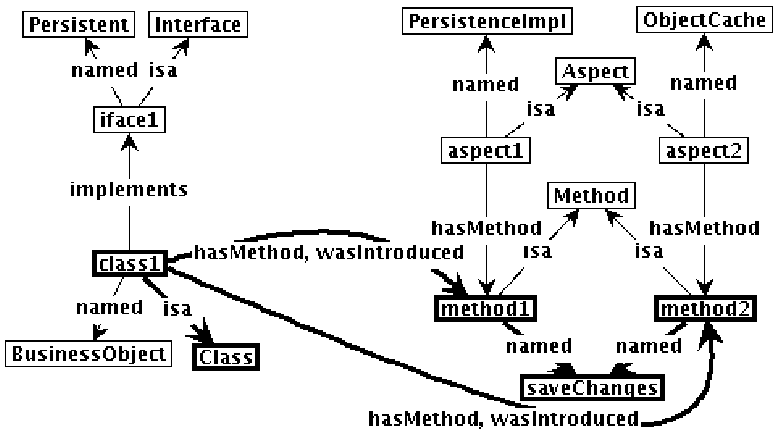


Figure 3.10: Program matches the violation rule in figure 3.9

3.2.1. Note that the edge labeled *extends+* will match one or more such edges and (arbitrary) intermediate nodes.

Thus, we have shown how graph matching patterns can provide a generic model for the detection of language rule violations.

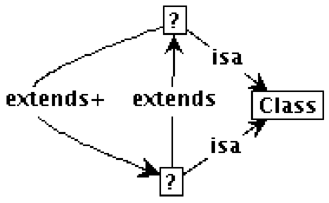


Figure 3.11: Violation rule: circular inheritance

3.4.2 Unintended effects of introductions

To detect unintended effects of introductions, it is necessary to define rules that match situations in which such effects occur. Tool- or compiler-developers can define such rules for their (AOP) language and make their tool issue warnings (or even errors) if these rules are violated. Using the graph-based approach we can trace back why the situation occurred (e.g. which introduction caused it), which could help a programmer decide whether the effect is desired or not.

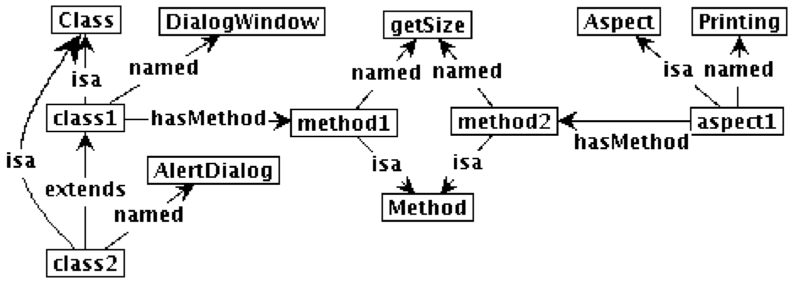


Figure 3.12: Program model of listing 3.5

Figure 3.12 shows a graph representation of the program in listing 3.5. Figure 3.13 specifies an introduction that (when applied) superimposes the method *getSize* as defined by the aspect *Printing*, on the class *AlertDialog*. As discussed before, performing this introduction effectively overrides the method *getSize* that class *AlertDialog* already inherits from class *DialogWindow*.

By defining a rule that matches such situations, the state-space exploration will show us when a state matches this rule, and allow us to trace back the

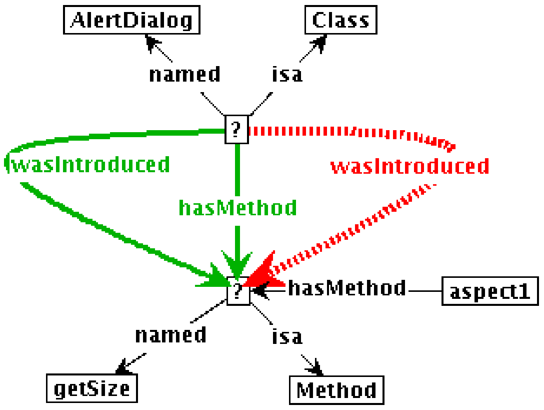


Figure 3.13: Introduction: AlertDialog.getSize()

introductions that led to this situation.

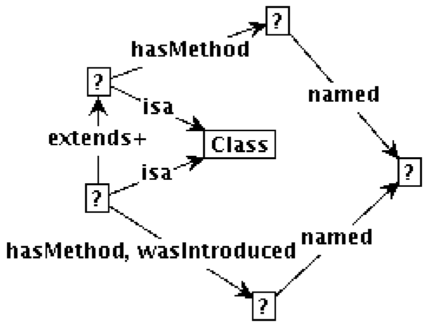


Figure 3.14: Rule matching method overriding by introductions

Figure 3.14 depicts such a rule for overriding methods. It looks for a combination of 2 nodes that are both classes, of which one extends the other, directly or indirectly. The notation *extends+* means there may be other nodes in between, as long as there are *extends*-edges between them. So effectively, this selects all the superclasses of a *class*-node. If the parent class has a method with the same name as the child class, and the method was introduced to the child class (by an introduction), the pattern matches. This means an existing method was overridden by an introduction, which may be an unintended effect.

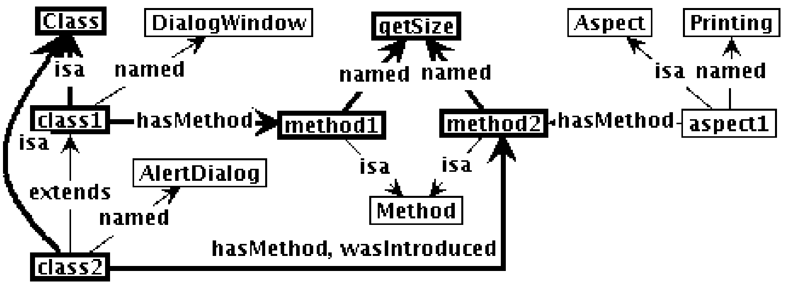


Figure 3.15: Matched rule: method override by introduction

Applying the introduction in figure 3.13 to the original program model in figure 3.12 results in the transformed program model shown in figure 3.15. As we can see, this transformed model matches the pattern specified in figure 3.14. The elements involved in the match are represented with thick lines and in a bold typeface.

Thus, we have shown how the graph models presented in this chapter can also be used to describe "code smells" that are related to the structural configuration of program elements, indicating when composition constructs are used in potentially unintended ways.

3.4.3 Ambiguous aspect specification

We can also use the state-space exploration offered by the Groove toolset, to detect whether a given combination of aspects and base program can be interpreted in more than one way.

To visualize the problem, we first represent the example in listing 3.7 using graphs and transformations. Figure 3.16 represents the (relevant) structural elements of the example.

Listing 3.7 contains two *declare parents* constructs, which are both depicted in figure 3.17. The first rule, figure 3.17(a), simply selects the class named *User* and the interface named *Persistent*, and introduces an *implements*-edge between the two. The second rule, figure 3.17(b), uses an embargo-edge as part of the selection pattern. The pattern selects every class that does *not* have an *implements*-edge to the interface *Persistent*. If such classes are found, an *implements*-edge is added to the interface named *SensitiveData* (but, as in all examples, only if this edge was not already introduced by a

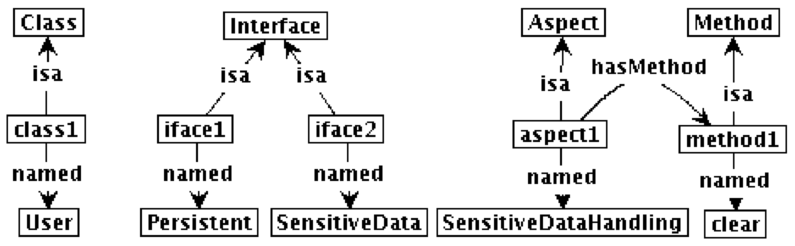


Figure 3.16: Program model, representing listing 3.7

prior application of the same rule). Note that, for the sake of clarity, we left out the additional constraint of selection by package name (listing 3.7, line 11). Finally, the introduction of the method *clear* defined by the aspect *SensitiveDataHandling* is a rule analogous to figure 3.3, except with different *name*-nodes.

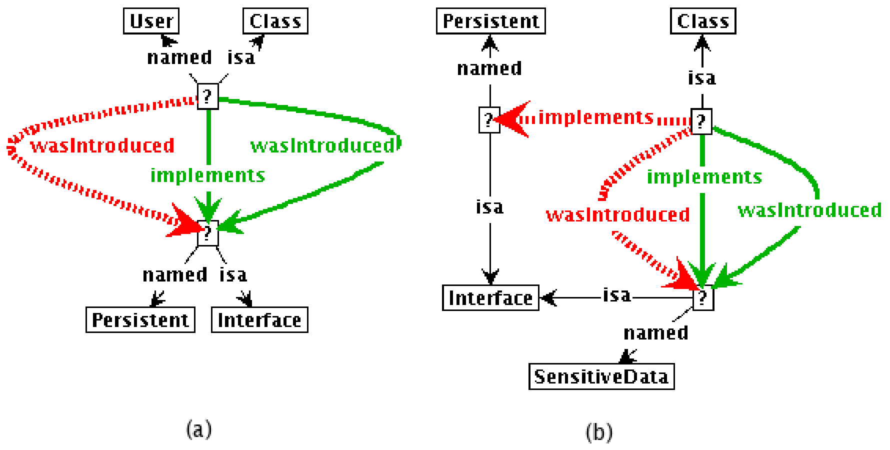


Figure 3.17: Two examples of "declare..implements"

Next, we can use the Groove Simulator to explore the possible orderings of matching the transformation rules and applying the introductions. Figure 3.18 shows that there are different orders in which the introductions can be applied.

Node *s37* represents the original program model (figure 3.16). Node *s39*

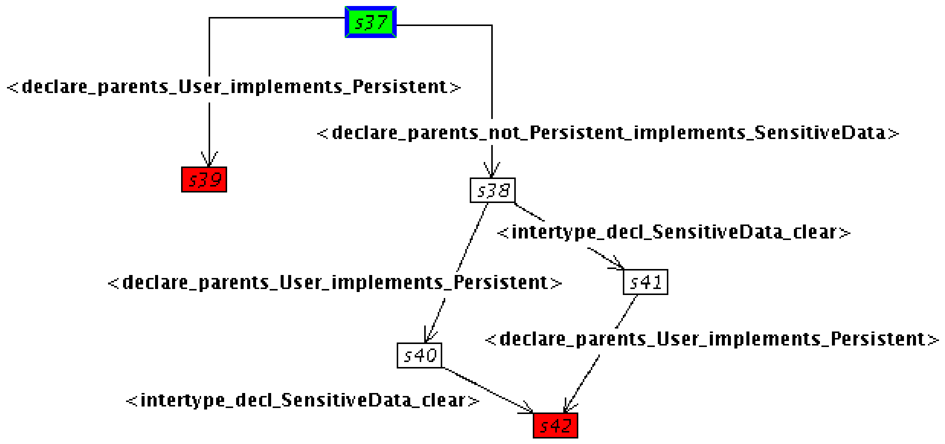

s37
<declare_parents_User_implements_Persistent>
<declare_parents_not_Persistent_implements_SensitiveData>
s39
s38
<intertype_decl_SensitiveData_clear>
<declare_parents_User_implements_Persistent>
s41
<declare_parents_User_implements_Persistent>
s40
<intertype_decl_SensitiveData_clear>
s42

Figure 3.18: Transformation state space: applying introductions in different orders

represents the state after the rule in figure 3.17(a) has been applied. Node *s38* represents the state after the rule in figure 3.17(b) has been applied. As can be seen in the diagram, in state *s39* there are no more applicable rules. As the class *User* implements the interface *Persistent*, the pattern that selects classes that do not implement this interface does not match anything. However, if the declaration of the interface *SensitiveData* is applied first (as is the case in state *s38*), the other rules still match and can be executed in two orders, which lead to the same state (*s42*) when both have been applied. In state *s39* as well as in state *s42*, no more transformations (introductions) can be applied. Therefore, these states are considered "end states", as is indicated by their different background color (gray). The fact that there is more than one (non-identical) end state indicates that the aspect specification can result in different programs (and hence, behavior), depending on the order of applying the aspects.

To show the different interpretations depending on the order of executing the introductions, figure 3.19 shows the final program model resulting from taking the first alternative (state *s39*), whereas figure 3.20 shows the second alternative (state *s42*). Clearly, the resulting program models differ considerably, which indicates that the specification is ambiguous.

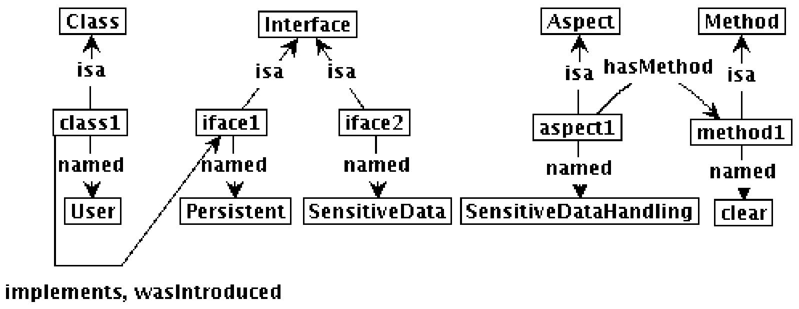


Figure 3.19: Final program model, alternative 1

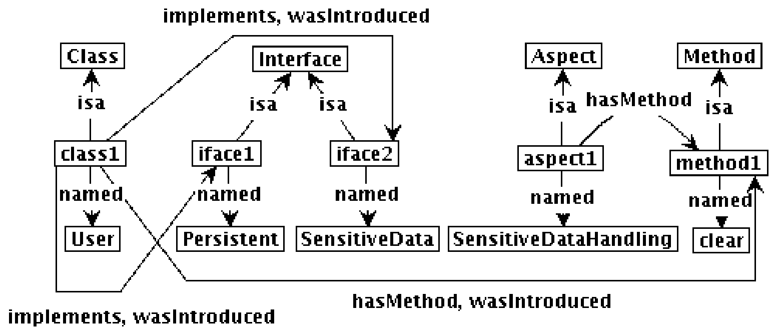


Figure 3.20: Final program model, alternative 2

To conclude, in this section we have shown how graph-based representations of programs can be used in combination with graph-based analysis tools, to detect several kinds of problems related to structural compositions specified by aspects.

3.5 Addressing composition conflicts

In this section, we discuss various design alternatives of preventing and handling the different types of composition conflicts that we identified previously.

3.5.1 Violation of language rules

A violation of a language rule always has to be detected by a compiler, and should result in an error message. When new language constructs are introduced into an existing language, or an existing construct is modified, language developers should take care of updating the checks executed by existing language rules, if necessary.

3.5.2 Unintended effects of introductions

Many compilers or checking tools warn their users when they use language constructs in a way that may lead to unintended effects. However, a drawback of this solution is that warnings are also generated in cases where the behavior is in fact intended. Such warnings may eventually become meaningless to the programmer, because distinguishing too many intended and unintended warnings is too cumbersome.

Alternatively, some language developers might want to forbid language constructs that can lead to an unintended effect. For instance, they would allow only introduction constructs that are guaranteed to be free of potentially undesired effects. This, however, restricts the expressiveness of the language.

Another solution to this problem is to make design intentions [107] (e.g. whether the overriding of an existing element is intentional) explicitly known to the compiler. Such design intentions can be explicitly indicated, for example, by using keywords or annotations that specify whether methods may be overridden by aspects. In fact, several (non-AOP) languages adapted such techniques for regular (object-oriented) method overriding. An example of using a keyword is the keyword *virtual* in C++ or C#. If a method is not explicitly declared to be virtual, an error message is issued by the compiler if subclasses try to override that method. An example of using an annotation is *@Override* in Java (documented in [79], see *java.lang.Override*). If a method is indicated as *override* but does not actually override a method of a superclass, compilers are required to generate an error message. An aspect language compiler could similarly generate a warning or error message if an introduced method overrides an existing one, but the introduction specification was not marked by such an annotation or keyword.

3.5.3 Ambiguous aspect specifications

There are several ways to ensure that an aspect specification is unambiguous. First, the aspect language could introduce an ordering mechanism, which could be used to *explicitly* specify the order of applying introductions.

One could argue that an application where this is necessary is probably not well-designed. In an example given earlier in this chapter, listing 3.7, the knowledge that "data is sensitive" is inferred from the fact that "it should not be stored persistently". However, there can obviously be many other reasons why data should not be stored persistently, other than it being sensitive. For example, some data may just represent intermediate results, which can be recalculated and hence do not need to be stored persistently. By changing the design such that the knowledge whether "data is sensitive" is not inferred in this way, the ambiguity can also be resolved. Thus, a compilers needs to detect situations where the design is ambiguous because of such "incorrect" specifications, and present the user with an error message.

Another alternative is to make the semantics of the language "smarter" (i.e. by defining fixed *implicit* ordering rules within the compiler) to resolve the composition specification in an unambiguous manner, or generate an error/warning if this is not possible. Considering our example in section 3.2.3, this would mean that the parent declarations are evaluated and applied in a manner that never leads to an ambiguity. For example, the compiler could always handle introductions that involve negations before any others. However, this could sometimes lead to counter-intuitive results.

Even if one prefers to apply ordering rules between introductions (implicitly as part of the compiler or explicitly specified as part of the aspects), it is probably undesired to force programmers to supply a total ordering specification. That approach would be rather inflexible in large systems and hinder the evolution of the program, since writing such a specification (or updating it when the program evolves) requires an understanding of the entire system. Instead, the programmer should only supply a (partial) order specification when and where it is needed, i.e. if the program would otherwise be ambiguous. Thus, a compiler should always detect cases where the programmer should have provided an ordering spec (or used a different -declarative- design), but failed to do so, leading to an ambiguous specification.

3.6 Discussion

There are several approaches to detect conflicts related to aspects. In the examples presented in this chapter we reason about aspects in the context of a base system. Some approaches pursue a more modular way of reasoning by looking only at the aspects (i.e. without considering a particular base program). However, aspects make quantifications over the base program and can superimpose behavior or additional program elements in (potentially) multiple places, which makes it hard to reason about aspects without considering the base program, in general.

To illustrate this, we take another look at listing 3.1. In this example, it is not possible to accurately detect that a conflict occurs by looking only at the aspects (line 5-11): the base context (line 1-3) is needed to determine that the problem exists. Without such a base context, it is not apparent that the specified patterns may match the same classes. This is especially the case because the interpretation of (AspectJ) type patterns depends on the (base) application under consideration. In this example, there is no way to discern (without a base system) whether a type pattern will match a class or an interface implemented by multiple classes, because the same kind of textual pattern (e.g. the fully qualified name of a type) is used to select interfaces, as well as classes.

Even if we assume we can somehow discern interfaces from classes while looking only at the type patterns, we still cannot generally be certain whether there is a conflict by looking only at the aspects. In this case, it depends whether the *base code* defines a class *BusinessObject* that implements the interface *Persistent*. We could detect this case as a *potential* naming conflict based on only the aspects. However, this approach would lead to many “false positive" detections of potential problems. This is partly caused by the limited amount of information that can be inferred by inspecting only the aspects. Because of this, we would have to assume the worst-case scenario (i.e. in the example, that the base system will define a class *BusinessObject* that implements the interface *Persistent*), even if the problem would not occur in many base programs, in practice.

In this chapter we present a graph-based approach. It would be possible to achieve the same results with other formalisms. However, we feel that for our purposes, the graph-based approach is very suitable:

- Our aim is to reason about — the structure of — programs. Graphs are convenient to represent program structures; for example as an Abstract

Syntax Tree. The descriptions of conflict patterns are also expressed using a graph-based model that closely matches the structural representation of the program.

- Since introductions change the structure of the base program, modeling introductions as program (hence graph) transformations is very intuitive. However, it may be less optimal for other types of aspect compositions.

- Graphs are relatively suitable for human viewing and reasoning. This is important for research purposes and experimentation. However, the selected graph representation is primarily optimized such that the transformation patterns we want to represent can be expressed in a straightforward manner, and can even be generated in an automated way. The graph representation is *not* optimized for human understandability, and layout becomes a critical issue for larger graphs.

- The reuse of algorithms defined as part of an existing graph matching- and rewrite system leads to a more *concise* description of the detection pattern (e.g., as compared to duplicating such matching and transformation code based on specific situations).

Almost all of the graph representations shown in this chapter are generated in an automated way[10]. All inputs, i.e., program graphs and transformation representations, are generated using a tool (developed by us) that automatically converts Java/AspectJ sources to models of the form as discussed in this chapter. All analysis results (i.e., state space representations, graphs incorporating the results of applying transformations) are generated using the (existing) Groove tool set.

It should be noted that for the sake of clarity, we manually fixed the graphical layout of the graphs and removed all elements that are irrelevant as to the discussions in this chapter.

Because we use an existing, well-modularized[11] set of tools and libraries, our approach is suitable to be included in compiler technology, while writing

[10]An exception are the patterns that codify language rule violations in section 3.5, which where written manually.

[11]i.e., the tool provides a graphical user interface, but all functionality can also be invoked programmatically and incorporated as a library in compiler technology. This has been done, in a different context [48].

only a minimal amount of integration code. However, when limiting oneself to specific languages and specific conflicts, it may well be possible to write more efficient “hardcoded" detection algorithms. Our primary goal is to offer a *generic* model that precisely and understandably describes conflicts, and can also be used to detect these conflicts in an automated way. Running the conflict detection does not take a noticeable amount of time in the (admittedly small) examples described in this chapter.

3.7 Related Work

In [104], Mehner et. al. propose a graph-based interaction analysis approach for aspect-oriented models. Although the graph formalisms they use are very similar as in our approach, the models to which they apply interaction analysis are specified at the requirements level (specified using UML). In this chapter, we proposed a way to describe models at the implementation level (i.e. program sources) using graph-based formalisms, in an automated way. Because of the different models, the approaches can address (or detect) issues at different stages of the software engineering life cycle.

In addition, our work also shows how graph matching patterns can themselves be used to describe and detect conflict patterns (e.g., as described in sections 3.4.1 and 3.4.2). Although this is likely also possible in the approach by Mehner et. al., the paper does not mention or exemplify such cases.

In [115], Rinard et. al. propose a classification system for aspect-oriented programs. This system characterizes two kinds of interactions between advices and methods: (1) control flow interactions between advices and methods; (2) indirect interactions that take place as the advice and methods access object fields. The classification system is supported by program analysis tools that automatically identify classes of interactions and hence help developers to detect potentially undesired/problematic interactions. However, this is not intended as a conflict detection (or warning) tool as such; it is left to the interpretation of the user what is, or might be, a conflict. Also, Rinard’s work focuses on the interactions among woven advice, while we focus on conflicts caused by introductions in this chapter.

In [83], Kessler and Tanter identify structural conflicts similar to our proposal. To this aim, the authors propose a dependency analysis technique. This technique is based on querying a logic engine (connected to their AOP platform) to infer dependencies between what has been looked at (while interpreting the pointcuts) and what has been modified in the structural model of a program. The proposal suggests to report the detected interactions to the programmer, who should then decide about an appropriate resolution.

In [88], Kniesel and Bardey analyze aspect interference, and propose a solution to resolve it. They observe conflicts related to unintended interactions (interference), which may be caused by incorrect or incomplete weaving. They represent the weaving of aspects as conditional transformations (expressed by logic predicates). Then, they analyze these transformations for *potential* interference. In their approach, only the conditional transformations (i.e. the aspects) are analyzed to detect potential conflicts, independently of any base program. Such modular reasoning clearly is a big advantage when reasoning about large systems. However, as we discussed in section 3.6, this may in some cases lead to the detection of many *potential* conflicts, most of which would only occur in “worst case” base systems. Only when the aspects are considered in combination with a concrete base system, it is possible to verify whether the potential conflict *really* occurs in that particular case. Hence, we suggest that a combination of early “aspects-only” checks (especially for the category of “certain” problems) and more detailed analysis (as we propose in this chapter) that includes the base program, can be useful.

In [82], Katz shows how to identify situations in which aspects invalidate some of the already existing desirable properties of a system. The paper emphasizes the importance of specifications of the underlying system. To detect interactions that invalidate desirable properties, he recommends regression verification with a possible division into static analysis, deductive proofs and aspect validation with model checking. We do not focus on the formal specification of desirable system properties, and do not require the programs to be augmented with such specifications.

In [46], Douence et. al. analyze interactions between aspects written in a formally defined stateful aspect language. They model the transformations done by aspects by precisely defining the semantics of the aspect weaver. Then, they detect interactions between aspects using static analysis. When conflicts are detected, they can be resolved by extending the specification of the aspects, i.e. by supplying the desired ordering. In this chapter, we use graph transformations to simulate aspect compositions, thereby also modeling part of the semantics of the language. However, we focus on the detection of conflicts related to introductions.

In [17], Aßmann and Ludwig present a weaving approach based on Graph Rewrite Systems (GRS). In this approach, aspects, joinpoints and weaving have well-defined and precise semantics in terms of graph-rewriting. In GRS-based aspect-oriented programming, aspect composition operators correspond to graph rewrite rules, weavings are direct derivations, and woven programs are normal forms of the rewrite systems. In our work we focus on composition conflicts rather than weaving; we introduce a graph notation as a means to precisely model the composition, with the goal of helping the understanding and detection of composition conflicts.

More recently, the authors of StrongAspectJ [43] have analyzed AspectJ with regard to how it influences the Java type system, and determined that it causes new classes of typing errors that are not detected at compile-time. They propose typing mechanisms to resolve these issues, and formally define this typing system and its type safety properties. This underlines our observations in section 3.2.1, stating that existing language properties (such as the typing system) need to be reconsidered in the presence of aspects, potentially leading to non-trivial changes to the compiler infrastructure. For example, such issues may impact the order in which particular modules (such as parsing, type checking, pointcut resolution, advice application etc.) within compilers have to be executed.

3.8 Conclusion

The aim of this chapter is to contribute to the understanding of aspect-oriented composition conflicts, in particular within the scope of structural composition (introductions). To this extent we propose and illustrate a systematic approach to analyze such composition conflicts in a precise and concrete manner. We employ graph-based formalisms to represent aspect-oriented programs, to represent introductions (as a graph transformation), and to express conflict detection rules. These formalisms have been introduced to deliver a precise explanation why and when some forms of composition cause a conflict, and to ensure that the categories of identified conflicts are not overlapping. Also, the precise formulation makes it possible to perform the conflict detection fully automatic, for example as part of an aspect language compiler or consistency analyzer.

To summarize, the main contributions of this chapter are:

(a) It proposes a general approach to the systematic and precise analysis of aspect composition conflicts.

(b) It presents a classification of composition conflicts related to introductions as caused by either: violation of language rules, unintended effects, or ambiguous aspect specifications.

(c) It offers a precise specification of the underlying causes for each of these conflict categories.

(d) We have shown that the proposed techniques are suitable for the automatic detection of composition conflicts; we have implemented a prototype that performs automatic conflict detection for each of the three categories. It can handle AspectJ source code and works for introductions (inter-type declarations). The implementation can detect and reveal conflicts in the graph representation of the program, using the Groove tool set.

(e) We discuss several alternatives of how to avoid or deal with the composition conflicts.

Although this is not the scope of this chapter, we believe that this approach is general enough to be able to model other types of composition conflicts, for example related to advice weaving.

Prototyping and Composing Aspect Languages

4.1 Introduction

As discussed in the introduction of this thesis, an important consideration in programming language design is that programs should clearly reflect the design decisions they embody. From this perspective, domain specific languages (DSLs) are an interesting phenomenon, as they can be designed to represent recurring concerns from a particular domain, in a way that is optimized specifically for that domain. This yields programs that more closely reflect the design domain, as compared to expressing the same design in terms of a general-purpose programming language.

Some DSLs address concerns from domains that typically crosscut an application. Examples of such domains are persistence, synchronization, caching, and access control. The benefits of using domain specific aspect languages (DSALs) to address such concerns are widely recognized [44, 98, 121]. In fact, the idea of expressing each crosscutting concern using a dedicated domain-specific language was at the very heart of the first proposals called “AOP” [86].

In this chapter[1], we introduce a meta-model of aspect-oriented languages. Since this model supports a wide range of aspect-oriented concepts, it can be used to model diverse aspect languages, including domain-specific aspect languages. In addition, we introduce an interpreter for this model, such that aspect languages can be prototyped and used (in an interpreted style) with real programs. Since aspects are mapped to a common meta-model when using this approach, this also enables us to use aspects written in several aspect languages, within the same program.

[1]The contents of this chapter are based on a paper published at the 22nd European Conference on Object-Oriented Programming, ECOOP 2008 [72]

Although this can be applied to general-purpose aspect languages, it is particularly useful in the context of domain specific aspect languages: most applications will need to express concerns from several problem domains, making it desirable to write programs using multiple DSALs. That way, each DSAL can be used to effectively address the concerns within its specific domain. In general, it is not trivial to compose aspects expressed in several DSALs however, as each language would typically construct its own model of the program; unless a lot of care is taken, the effects of one aspect may not be reflected in the models constructed by other DSALs. In addition, aspects written in several DSALs may interact with each other, possibly in undesirable ways (depending on the situation).

To summarize, this chapter contributes the following:

(1) We propose an aspect interpreter framework that can be used to prototype diverse aspect languages. To demonstrate that our framework supports a wide range of aspect language concepts, we use it to prototype several domain-specific aspect languages. This is discussed in section 4.4.

(2) Using our approach, aspects written in several (domain-specific) aspect languages are mapped to a common model. As a result, we can compose applications that are written using multiple aspect languages, as we will illustrate in section 4.5.

(3) The framework provides explicit mechanisms to specify composition of advices, even if advices are written in several aspect languages. This is also discussed in section 4.5.

In this chapter, we show implementations of only three DSALs. However, our work is based on a thorough study of aspect oriented languages [105] including general-purpose aspect languages, as well as the modeling of their possible implementation mechanisms using an interpreter, as presented in [33].

In section 4.3, we introduce the framework itself. Section 4.4 demonstrates the framework by showing the implementations of several DSALs using our framework. Section 4.5 discusses the composition of aspects written in multiple DSALs, including specifications to resolve (potential) interactions between aspects. Section 4.6 discusses several design and implementation considerations related to our framework. We discuss related work in section 4.7, and conclude this chapter in section 4.8.

4.2 Common aspect language concepts

One of the defining features of AOP languages is the support for "implicit invocation" of application behavior based on quantification ("pointcuts"). That is, behavior can be invoked without an explicit reference (such as a *method call* statement) being visible in the (source) code at the point of invocation. Implicit invocation is a key feature of the interface between the base program and the aspect program. The framework to model aspect language mechanisms we present in this chapter is strongly based on implicit invocation as the connection between the base program and the aspects.

In this section, we discuss the concepts used in the aspect language domain, based on a reference model proposed in [105], as well as an early version of an aspect metamodel presented in [74].[2] We then propose a framework that provides common implementations of these concepts, while supporting variations on these concepts found in different aspect languages. We discuss the high-level design and architecture of the framework, which was first proposed in [33] and [75]. Finally, we briefly outline the workflow used to prototype DSALs using this framework.

First of all, aspect languages support the concept of *pointcuts*. Pointcuts define the circumstances under which an aspect influences a program – for example, at locations that can be identified in the source code, such as when entering a particular method, or under particular runtime conditions, such as when a particular variable is assigned a specific value. Thus, pointcuts can be compared to call statements in an object-oriented language, in the sense that they determine *when* and *where* "externally" defined behavior (such as a different method) should be invoked. Pointcuts can be seen as predicates

[2]Regarding the history of the metamodel and interpreter presented in this chapter: a preliminary version of the metamodel presented in this chapter was designed in collaboration with Tom Staijen, Arend Rensink, Lodewijk Bergmans and Klaas van den Berg at the University of Twente. This model was used as input to define a metamodel of aspect languages within the European Network of Excellence on Aspect-Oriented Software Development, where we collaborated with Johan Brichau, Jacques Noye, Mira Mezini, Christoph Bockisch, Vaidus Gasiunas, Johan Fabry and Theo D'Hondt. Based on these discussions, Johan Brichau first implemented an interpreter for this metamodel, called MetaSpin [33]. Thus, the model presented in this chapter reflects the input of many researchers working on the design and implementation of diverse aspect oriented languages.

or conditions over the *execution state* of a program. The execution state may, in a broad sense, include information about the call stack, objects, or even the execution trace and structure of the program. As pointcuts may match at several places or moments during the execution of a program, the concept of *join points* is used to model references to the relevant execution state (e.g., which method is being intercepted) whenever a pointcut matches. In this sense, join points can be compared to a meta-representation of a call.

Pointcuts can be bound to *advices*, which specify behavior to be executed. Thus, advices can be seen as very similar to object-oriented methods, with the difference that they can use the join point information to adapt their behavior based on the current runtime context (whereas methods in object-oriented languages do not usually have the same level of meta-information available).

Bindings specify how pointcuts are connected to advices. This allows for the reuse of both pointcut and advice specifications, since pointcuts can be attached to several advices, or vice versa. In addition, by disabling or enabling bindings at runtime, dynamic deployment (of aspects) can also be modeled. Bindings can be compared to the method lookup mechanism in object-oriented languages, connecting call statements (pointcuts) to method implementations (advices).

Similar to methods in object-based languages, advices may want to share *state* among each other. In object-oriented languages, variable binding is relatively straightforward, since the desired context is usually specified on the calling side. That is, calls are invoked on a specific object, which determines the "this"-context and thereby the binding of variables in the invoked context. When using aspects, however, there is no visible "calling side" due to the use of implicit invocation. Instead, variables can be bound to specific instances based on properties found in the join point context. This way, variables can be instantiated based on e.g. the active object identified by the current join point (i.e. the "caller"), the current thread context, the type of the active object, as a singleton (a global variable shared by each advice), or based on any other property available through the join point context.

All of the above elements can be grouped into modules, commonly called aspects, thus representing the *structure* of aspects. As is the case with classes in object-oriented systems, the structure may be used for grouping and encapsulation of related behavior and data.

Finally, *constraints* may be imposed, so that the desired behavior can be determined in cases where several aspects want to execute behavior at the same join point. This can be compared to the resolution of multiple-inheritance conflicts: if multiple parent classes define a method with the same name, some languages offer explicit mechanisms to resolve the order in which the composition mechanism should consider these methods.

4.3 JAMI - an aspect interpreter framework

In this section, we introduce the Java Aspect Metamodel Interpreter (JAMI) [8]. This interpreter framework uses the concepts defined in the previous section as a metamodel of aspects, and allows to run aspects defined in terms of this metamodel together with regular Java programs, in an interpreted style. The high-level concepts defined in the previous section are used to define the high-level structure of and control flow within aspects; these concepts can be refined to model a wide range of concepts found in aspect languages.

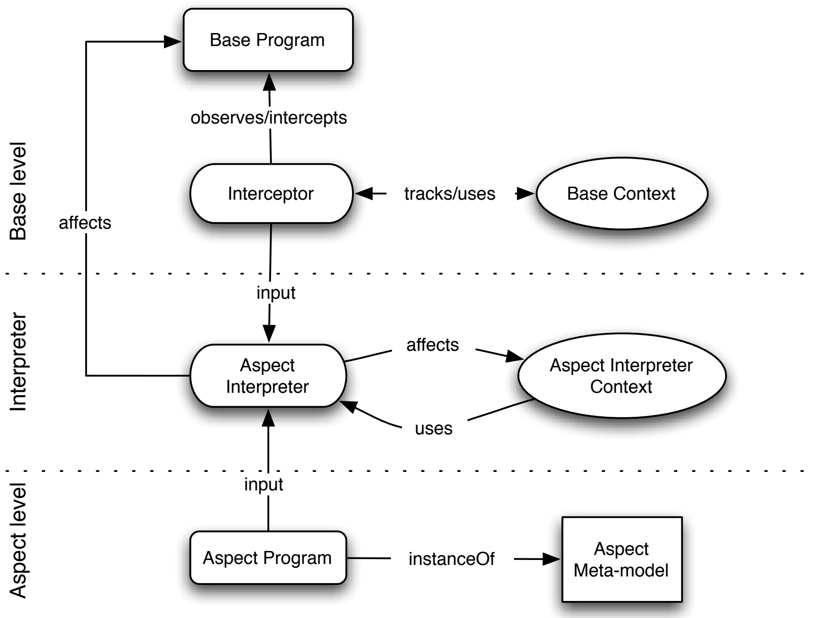


Figure 4.1: Aspect Meta- model Interpreter - Conceptual overview

Figure 4.1 shows a conceptual overview of the framework. We assume a *base program* that defines the basic units of behavior (e.g., methods) of an application. An *interceptor* component is used to observe the base program while it is running, and to intercept its control flow at any moment during the execution that is of potential interest to the aspect interpreter. The interceptor creates and keeps track of a (partial) representation of the state of the base program, called the *base context*.

Whenever the execution of the base program is thus intercepted, the control flow is diverted to the *aspect interpreter*. This interpreter determines whether, and if so in what way, the control flow or behavior of the base program should be affected. These decisions are specified by an *aspect program*, which is expressed in terms of the aspect language concepts – or refinements thereof – discussed previously. While the aspects are being evaluated, the interpreter uses the *aspect interpreter context* to keep track of its internal state.

4.3.1 The high-level structure of JAMI

Figure 4.2 shows a high-level class diagram of the Java Aspect Metamodel Interpreter. Below, we explain the structural elements shown in this figure.

When using the aspect meta-model interpreter, a (Java) program in principle runs as it would without the interpreter. However, the *interceptor* intercepts the control flow at any point that is of potential interest to the aspect interpreter. Apart from intercepting method calls, the mechanism keeps track of *context* information that may be of interest to the framework. For example, it keeps track of the call stack, including senders, targets, and method signatures of all calls on the stack, as well as field assignment instructions. Upon interception of the control flow, the mechanism creates a *join point* object representing the current point of interception. Refinement classes exist for different join point types, such as `MethodCallJoinpoint`, `MethodReturnJoinPoint` or `AssignmentJoinpoint`. Each of these join point objects keeps references to the relevant context information - e.g., the method that was executing upon interception, etc.

Subsequently, the aspect *evaluator* determines (1) whether any aspects apply at this join point, (2) if so, which behavior (advice) should be executed, and (3) if several advices apply, how their evaluation should be handled — for example, (ordering) constraints may apply to advices at shared join points. The next subsection describes the control flow within the evaluator in detail.

As shown in figure 4.2, the aspect evaluator has references to a set of active aspects and constraints. Typically, we initialize the aspect evaluator and the list of active aspects at program startup; since we use an interpreter-

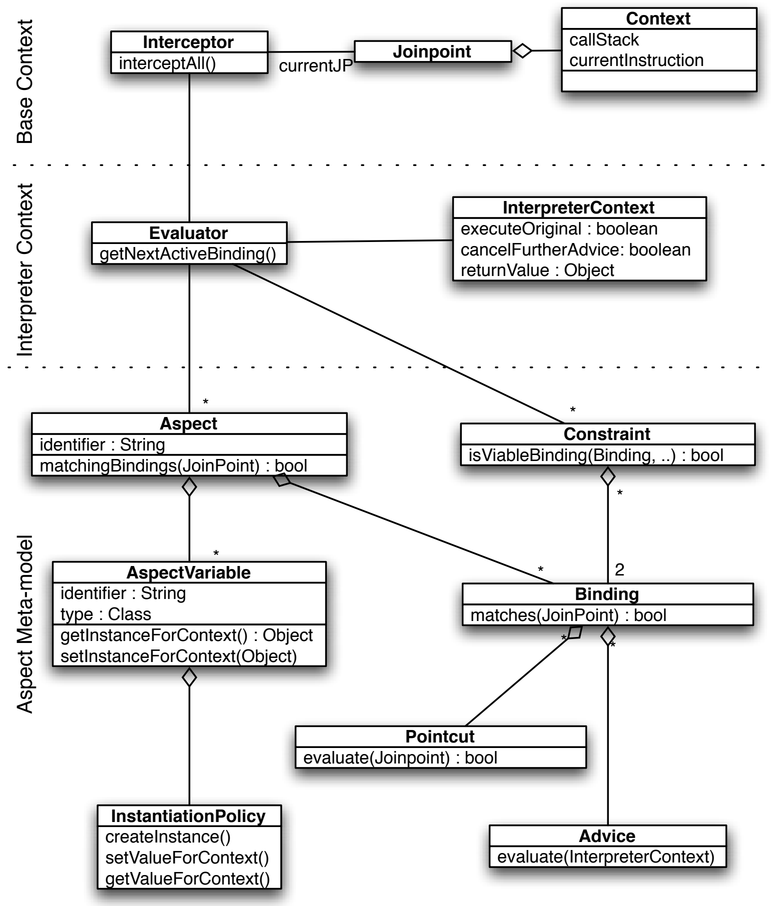


Figure 4.2: Java Aspect Meta-model Interpreter - Implementation overview

based approach it is however trivial to add or remove aspects and constraints at runtime.[3]

Aspects are modeled as consisting of *bindings* between *pointcuts* and *advices*, as well as *aspect variables* that can be used to share state between several advices, or between multiple executions of the same advice.

Advices can internally share state by using *aspect variables*, available through

[3]This is typically called “dynamic deployment” of aspects.

the interpreter context. Which instance of an aspect variable is used is decided by the *instantiation policy* (see figure 4.2) associated to a specific aspect variable. Instantiation policies use information available through the join point context to find the correct instance; for example, aspect variables may be instantiated per target object as indicated by the current join point context.

4.3.2 The high-level control flow within JAMI

Figure 4.3 shows a detailed specification of the control flow within the aspect evaluator.

```
 1: function $aspectEvaluator(aspects, constraints, currentJP, intContext)$
 2: begin
 3: $allBindings \leftarrow \emptyset$
 4: for all $a \in aspects$ do
 5:     $allBindings \leftarrow allBindings \cup a.bindings$
 6: end for
 7: $matching \leftarrow \{b \in allBindings \mid b.pointcut.evaluate(currentJP) = true\}$
 8: $remaining \leftarrow matching$
 9: $executed \leftarrow \emptyset$
10: while $remaining \neq \emptyset$ do
11:     $viableBindings \leftarrow \{b \in remaining \mid$
        $\#\{c \in constraints \mid c.isViableBinding(b, matching, executed) = false\} = 0\}$
12:     if $viableBindings = \emptyset \vee intContext.cancelOtherAdvices()$ then
13:         return
14:     else if $\#viableBindings > 1$ then
15:         $selectedBinding \leftarrow chooseBinding(viableBindings)$
16:     else
17:         $selectedBinding \leftarrow viableBindings\#0$
18:     end if
19:     $selectedBinding.advice.evaluate(intContext)$
20:     $executed \leftarrow executed \cup \{selectedBinding\}$
21:     $remaining \leftarrow remaining \setminus selectedBinding$
22: end while
23: end
```

Figure 4.3: Control flow specification of the aspect evaluator

We explain the control flow in detail, based on this figure. First, the aspect evaluator gathers all bindings from all active aspects (line 1–4). Then, the pointcuts referenced by these bindings are evaluated, and those that match the current join point are collected (line 5). Lines 8–20 define a loop that runs while there are remaining bindings to be executed. Line 9 describes the core of our constraint mechanism: of the bindings remaining to be executed, only those are considered viable for which none of the constraints indicates that they are not currently viable to be executed. Constraint implementations can make this decision based on the binding that is being considered for execution, as well as other bindings that match at the same join point, and finally a list of the bindings that have already been executed; in section 4.3.3 we show implementations of several constraint types that use this information.

If, after evaluating the constraints, no viable bindings remain, or if an advice that was executed in a previous iteration of the loop has signaled (through the interpreter context) that remaining advice executions should be cancelled, the control flow is returned to the base program (line 10+11).

Otherwise, if there are still multiple bindings viable for execution, an ordering has to be decided using an explicit order resolution algorithm (line 13), rather than a resolution based on only declarative constraints. Thus, JAMI supports two mechanisms to specify advice orderings. Although this is not strictly necessary,[4] different languages may want to use one method over the other. Order resolution algorithms can be supplied by users of the framework. For example, it is possible to write a simple ordering algorithm that prioritizes bindings based on an ordered list of aspect identifiers.

Finally, the advice indicated by the selected binding is evaluated (line 17), potentially affecting the base program and the interpreter context. The selected binding is then added to the set of bindings already executed at this join point (line 18) and removed from the set of bindings remaining to be executed (line 19).

Thus, the interpreter framework defines the high-level structure and control flow of aspects. To provide the flexibility required to model features of particular languages, each concept can be either instantiated in a dedicated configuration of framework elements, or refined (extended) when necessary. JAMI provides many of the common implementations found in different aspect languages as refinements of the classes in figure 4.2, which we list in section 4.3.3. In section 4.4, we show that these implementations of common concepts are defined at an abstraction level that is appropriate when prototyping DSALs, since the concepts are reused in implementations of quite diverse

[4]i.e., we could simply assume that constraints have to be present such that at any point, at most one binding would be viable for execution, and display a warning or error message otherwise

languages.

By enforcing a fixed high-level control flow, our framework provides a common platform that enables composition of aspects written in multiple DSALs, as we will show in section 4.5.

4.3.3 Implementations of common aspect concepts

In this section, we list refinements of many common aspect concepts, implemented within JAMI. Each of the concepts listed here are implemented as extensions of the top-level classes shown in figure 4.2.

First of all, JAMI natively[5] supports several types of join points, listed below:

Behavioral join points

Behavioral join points are join points that occur during the execution of the program. All behavioral join points contain a reference to the context of the (base) program. Below is a list of implemented join point types:

- *MethodCallJoinPoint*: represents calls to methods, including constructor methods.
- *MethodReturnJoinPoint*: represents return instructions, including returns from instructor methods.
- *ConstructorCallJoinPoint*: represents calls to only constructor methods.
- *ConstructorReturnJoinpoint*: represents returns from only constructor methods.
- *AssignmentJoinPoint*: represents assignments to instance variables.

The meta-model itself can be extended to support structural join points that refer to specific elements in the program structure (e.g., classes or method definitions). Such "join points" could then be generated when, for example, a class is loaded or at program startup. These join points would be bound to

[5]That is to say, these join point types are already implemented; the framework can freely be extended, although adding new join point types will typically necessitate updating the interceptor mechanism as well. For other elements of the meta-model this is usually not the case.

"structural advices", that is, advice that adapts the structure of the application – for example, introductions of annotations as discussed in previous chapters. For practical reasons, this is not implemented in the interpreter-based prototype of the meta-model.

Pointcuts

A pointcut or "join point selector" selects (matches) join points. Each pointcut implements a method *evaluate(JoinPoint)*, which returns true or false based on whether or not the pointcut expression should match the supplied join point. Pointcuts can also be composed from sub-pointcuts, e.g., using the common logic operators (and, or, not). Below, we show a list of predefined pointcut classes. Many of these classes can be *parameterized* upon instantiation. In those cases, the parameters are displayed in a different typeface (`like this`).

Selection based on join point type

- *SelectConstructorCalls* matches join points of type ConstructorCallJoinPoint, thus matching calls to constructors.

- *SelectConstructorReturns* matches join points of type ConstructorReturnJoinPoint, thus matching return instructions from constructors.

- *SelectMethodCalls* matches join points of type MethodCallJoinPoint, thus matching calls to regular methods as well as constructors.

- *SelectMethodReturns* matches join points of type MethodReturnJoinPoint, matching returns to regular methods as well as constructors.

Selection based on call properties

- *SelectByMethodName* matches if the most recent call on the stack (according to the join point context) is to a method with the specified `name`.

- *SelectByMethodParameterTypes* matches if the most recent call is to a method with the specified `parameter types`.

- *SelectByMethodSignature* matches if the most recent call is to a method with the specified method signature. This pointcut is a composition of *SelectByMethodName* and *SelectByMethodParameterTypes*.

- *SelectByMethodAnnotation* matches if the most recent call is to a method with the specified `annotation`.
- *SelectByObjectType* matches if the most recent call (obtained through the join point context) has a *target* respectively *sender* object of a specified `Type`. Whether the selector looks for target or sender objects is specified by a parameter, provided when the advice is instantiated.
- *SelectByObjectSuperType* idem, but matches if the specified `Type` is a parent type of the current sender or target.

Selection based on field properties

- *SelectFieldAssignments* matches assignment instructions to fields (instance variables).
- *SelectByFieldTypeAs* matches assignment instructions to fields of a specific `Type`.
- *SelectByFieldName* matches assignment instructions to fields with a specific `Name`.
- *SelectByFieldContainingType* matches assignment instructions to fields contained by the specified `Type`.

Composed selection criteria

- *TrueSelector* matches any join point.
- *AndSelector* matches only if two supplied sub-pointcuts both match the current join point.
- *OrSelector* matches only if either of two supplied sub-pointcuts match the current join point.
- *NotSelector* matches only if a specified selector does *not* match the current join point.

Advice actions

The framework provides a set of actions commonly used as part of advices, such that they can be reused by several language implementations. These actions can be composed into composite advice blocks.

Each advice action implements a method *evaluate(InterpreterContext)*, which executes the advice, and returns a boolean value indicating whether or not the evaluation was successful. An advice can indicate failure, for example, if it attempts to forward a call to a method implementation with an incompatible signature. Through the interpreter context, advices can access join point information (and hence, the base context). In addition, advices can affect the state of the aspect interpreter through the interpreter context. The return values of advices, indicating success or failure, can be used in constraint specifications, as explained in section 4.3.3.

Below, we list predefined primitive advice actions:

- *MethodCallAction* executes a call to a method with a given signature. The target object to which this method call is sent can be set independently (see: *SetTargetObjectAction* and its subclasses), as can the parameter values of the call, if any (see: *SetCallParametersFromJPAction* and *SetCallParametersToMetaAction*).

- *SetTargetObjectAction*, or specifically its subclasses described below, can be used to set (within the interpreter context) the target object of a method call to be invoked by another advice, such as a *MethodCallAction*. *SetTargetObjectAction* has the following subclasses: *SetStaticTargetObjectAction*, which sets the target object to a static object instance, i.e., as supplied when the advice action is instantiated; *SetTargetObjectFromJPAction*, which sets the target object to be the same as the target object found at the top of the call stack within the join point context; and finally, *SetTargetObjectFromAspectVariableAction*, which sets the target object to the value of an aspect variable.

- *SetCallParametersFromJPAction* sets the method parameters of a call that may later be executed by advice actions such as *MethodCallAction*. The parameter values are copied from the parameter values of the method found on top of the call stack, as found through the join point context. This action allows parameters to be rearranged and/or left out to accommodate the forwarding of calls to methods with a different signature.

- *SkipOriginalCallAction* is an advice action that may be evaluated at "method call" or "field assignment" join points. It instructs the interpreter to "skip" the original instruction, such that, if no other advices are present, the program continues executing after the skipped instruction.

  Note that if several pointcuts match the same join point, other advices may be planned for execution. It is the responsibility of the framework user to define the desired semantics in such a case, e.g. when skipping a call you may (or may not) want to cancel the scheduling of other advices at the same join point (see: *CancelOtherAdvicesAction*).

  In addition, when skipping a call that is expected to return a value, the framework user has to make sure that a return value is set by the advice code instead. For field assignment join points, the return value set by an advice is used as the assigned value instead of the skipped assignment statement. Advices can set the "base context" return value in several ways, for example by setting a fixed return value (see *SetFixedResultAction*) or by invoking an alternative ("redirected") method implementation (see *MethodCallAction*).

- *CancelOtherAdvicesAction* instructs the aspect evaluator (through the interpreter context) that it should not evaluate any further advices at this same join point instance (if any).

- *SetAspectVariableValueAction* sets the value of an aspect variable to the current return value found in the interpreter context. Such values may have been put in the interpreter context by prior method calls or advice actions.

- *SetFixedResultAction* sets the result variable within the interpreter context to a fixed value, specified upon instantiation of this advice action.

- *ComposedAdviceAction* executes a specified list of advice actions in sequence.

Aspect Variable Instantiation Policies

Aspect variables can be instantiated based on properties found in the join point context. For example, advices may share the same globally defined instance of an aspect variable, or share an instance for each target object to which a particular advice is applied. Section 4.4 provides several examples demonstrating the use of these instantiation policies. Here, we list several predefined instantiation policies:

- *Singleton* returns the same instance for the entire application
- *PerObject* returns the same instance per target object of the most recent call found through the join point context
- *PerObjectMethod* returns the same instance per combination of target object and method of the most recent call found through the join point context
- *PerClass* returns the same instance per class of the target object of the most recent call found through the join point context
- *PerClassMethod* idem, but per combination of target class and most recently invoked method
- *PerThread* returns the same instance per thread of execution

Note that instantiation policies are defined *per variable*, rather than e.g., per aspect, as is the case in AspectJ. In AspectJ, aspect variables are either defined static ("global" for that aspect), or according to a single instantiation policy that applies to all variables in an aspect. Our approach allows greater expressiveness in the binding of aspect state (i.e., variables) to advice execution contexts. For example, a single aspect may keep an object representing a database connection *per thread*, while keeping a cache of previously obtained query results *per method*. To express a similar scenario in AspectJ, programmers would have to manually implement a solution that, e.g., employs hash tables to map values per thread or per method.

Constraints

When multiple bindings are active at the same join point (i.e., when the pointcut part of several bindings matches at that same point), constraints may be used to ensure a correct execution order. For example, some advices may depend on (the execution of) other advices. In other cases, several advices may *conflict* when applied at the same join point.

In JAMI, constraints are always specified between two bindings, which may be part of different aspects. Constraints involving more than two bindings can be expressed as multiple binary constraints. Each constraint implements a method *isViableBinding*, which has three arguments: a binding to be tested for viability of execution, a set of all bindings matching the current join point, and a set of bindings already evaluated at the current join point.

This declarative constraint model is based on an existing study of aspect composition at shared join points [105, 106], but is here applied to a generic framework that can also be used to compose aspects written in multiple aspect languages, as we will demonstrate in section 4.5.

- *PreConstraint(X,Y)* can be used to specify a partial ordering between advices. It prioritizes execution of binding X over the execution of binding Y, if both are present at the same join point. To be precise, its implementation of *isViableBinding* returns `false` if and only if the binding tested for viability equals Y, and in addition the pointcuts of X and Y both match the current join point, and X has not yet been executed at this join point.

- *ConditionalConstraint(X,Y)* can be used when the execution of one advice depends on the (successful) execution of another advice. It conditionally executes Y, i.e., only if execution of X was successful. To be precise, its implementation of *isViableBinding* returns `false` if and only if the binding tested for viability equals Y, and either (1) X has not yet been executed at the current join point, or (2) the pointcut specified by X does not match the current join point, or (3) the execution of advice specified by X failed.

- *SkipConstraint(X,Y)* can be used to prevent the execution of conflicting advices at a shared join point. It skips the execution of Y, if X is successfully executed at the same join point. To be exact, its implementation of *isViableBinding* returns `false` if and only if the binding tested for viability equals Y, and in addition, X was either already successfully executed at the same join point, or X is not executed yet but its pointcut does match the current join point.

The use of constraints will be exemplified extensively, in section 4.5.

4.3.4 Prototyping DSALs: general workflow

We now briefly describe the steps involved in prototyping a DSAL using JAMI.

First of all, we define a grammar that can conveniently express the domain concepts of a particular DSAL, as well as the relations between those concepts. Next, a parser is needed - using a parser generator is typically the most convenient way to implement this (we use Antlr, but any Java-based parser generator could be used). Subsequently, we convert the abstract syntax trees (ASTs) obtained from the parser to an object-based version, such that the domain concepts are semantically represented by objects. This conversion can be implemented using handwritten code, or by using generated “tree walkers”. The final step is to convert the object-based AST representation of domain concepts to JAMI elements. That is, framework elements are instantiated such that they together represent an aspect program. We currently implement this conversion using handwritten code.

Once an aspect written in a DSAL is (automatically) converted to JAMI elements in the way described above, the JAMI interpreter framework can run the aspect as part of a normal Java application. Typically, we write an explicit instruction to load and deploy the aspect at application startup. Once the aspect is deployed, JAMI ensures that the aspect behavior is called at the appropriate times, as described in the previous section.

In the following section, we introduce several examples to demonstrate the framework in detail. We focus on expressing each example using JAMI elements, showing the object structure of the JAMI representations of each aspect[6]. The full examples, including parsers and code that executes the mapping steps as described above, can be downloaded from the JAMI website [8].

4.4 Features of JAMI, demonstrated by example

In this section, we show 3 aspect languages, each optimized for a specific task, implemented using the Java Aspect Metamodel Interpreter. We first introduce a running example that we use to demonstrate each language.

Figure 4.4 shows the UML class-diagram of a simple word processor application. Within this application, class *Document* defines several methods to modify a document (`addLine` and `setContent`), a method to obtain the doc-

[6]In fact, the object diagrams in this chapter are based on observing (using the Eclipse debugger) the actual runtime JAMI ASTs, which where automatically converted from the DSAL notations

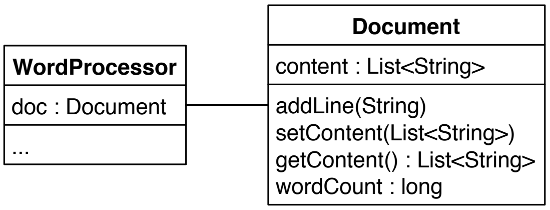


Figure 4.4: Example application, used throughout this chapter

ument content (`getContent`), as well as a method that counts the current number of words in the document (`wordCount`).

In the following subsections, we extend this example using aspects written in several domain-specific aspect languages. These extensions will allow us to: (1) create an auto-save mechanism using a modularized version of the decorator pattern, (2) synchronize access to documents, such that multiple threads can access its content concurrently (for example, to run a background spellchecker), and (3) cache the results of expensive method calls, as long as variables on which the method depends are unchanged.

4.4.1 A domain-specific language for the decorator pattern

Suppose we want to add autosave behavior to our word processing application. We can implement this using the decorator pattern [57] by defining a class *AutoSaveDocument*. This class implements the same methods as class *Document*, but adds the behavior to save any changes made to the document (e.g., to a file), before forwarding method calls to the original document object - see figure 4.5.

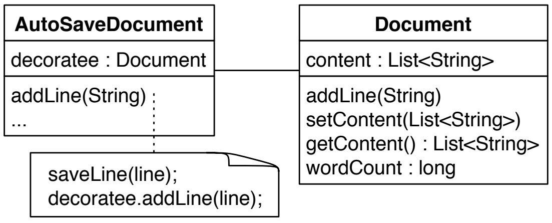


Figure 4.5: Decorator pattern example

Listing 4.1 shows how we can use this decorator class:

```
 1 public class WordProcessor {
 2   Document doc;
 3   AutoSaveDocument autoSaveDoc;
 4 
 5   public void testAutoSave() {
 6     doc = new Document();
 7     autoSaveDoc = new AutoSaveDocument(doc);
 8 
 9     autoSaveDoc.addLine("AutoSaved"); // ok
10     doc.addLine("Not AutoSaved"); // bad!
11   }
12 }
```

Listing 4.1: Example of decorator pattern usage

There are two issues with this code. (1) When writing this in plain Java, we can still make calls to the object that is being decorated (also called the *decoratee*) - see line 10. This is almost certainly unintended, as the behavior of the decorator is not invoked this way. (2) Part of the code dealing with the decorator pattern is the responsibility of the client (class *WordProcessor* in this example) – it is not fully modularized. We will show simple domain-specific extensions to Java to solve these issues.

Enforcing the decorator pattern

We start with the issue of enforcing the decorator pattern. Once a *decorator* is associated with a *decoratee* (listing 4.1, line 7), all subsequent calls should be made to the *decorator*. We define a small domain specific aspect language (DSAL) to enforce this - by automatically forwarding calls to a *decoratee* object to the *decorator*. In the first version of our language, a program in this DSAL defines which classes may act in the *decorator* and *decoratee* roles, respectively, see listing 4.2:

```
decorate: Document -> AutoSaveDocument
```

Listing 4.2: A statement in our decorator DSAL

We take the specification in listing 4.2 to mean the following: objects of type *Document* may be decorated by objects of type *AutoSaveDocument*. However, we do not want to simply decorate every object of type *Document*. Doing so would defeat the purpose of the decorator design pattern, which is used to decide *dynamically* which objects should be decorated. Therefore, our first implementation will automatically infer the *decorator-decoratee* relationship between objects from the occurrence of constructor calls such as shown in listing 4.1, line 7. That is, an association is established upon calling a constructor of a class that is indicated to be a *decorator* in our DSAL specification (listing 4.2), of which the first argument is of the corresponding specified *decoratee* class.

Mapping to JAMI

We first express the simple DSAL described above in terms of the aspect language concepts discussed in section 4.2. Subsequently, we express its implementation in terms of JAMI elements (and refinements thereof, when necessary) as defined in section 4.3.

The main task of aspects written in the DSAL proposed above is to intercept calls to *decoratee* objects, and forward them to the associated *decorator* object. In aspect terminology, the interception specification can be seen as a *pointcut*, whereas the forwarding part is an *advice* specification.

For the above pointcut/advice definition work as desired, the aspect program needs to know which objects are *associated* in the roles of *decoratee* and *decorator*, i.e. we need to establish and store this association as part of the aspect.

Therefore, to create the association, we use another pointcut that intercepts calls to the constructor of the type acting as *decorator*. The corresponding *advice* creates an association between the object being instantiated (the *decorator* object), and the first argument of the constructor call (which we assume to be the *decoratee* object, as discussed above). This association is stored in an aspect variable, so that it can be shared between advices.

We now explain the mapping to JAMI in detail, by showing object diagrams that represent the aspect program given in listing 4.2. The object structures shown here consist of elements (classes) predefined by the JAMI framework, unless indicated otherwise.

Figure 4.6 shows an aspect definition expressed using JAMI elements. The figure shows one aspect, containing two "selector-advice-bindings", which are described in subviews shown in figures 4.7 and 4.8. These will be discussed shortly. In addition, the aspect defines two aspect variables, *decorator* and *decoratee*, used to share aspect state between related advices, or between multiple executions of the same advice. Each aspect variable has an instantiation policy, which defines what instance of the variable should be used in each particular join point context. For example, a "singleton" policy means that there is one

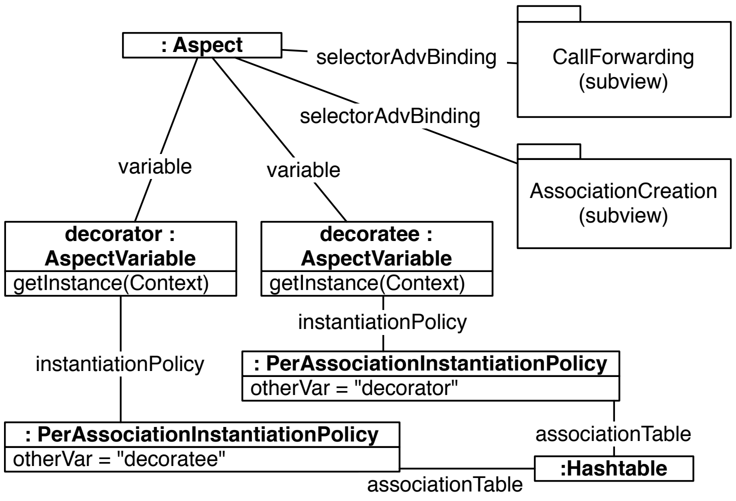


Figure 4.6: Bindings between the parts of a decorator aspect

instance of the variable for the entire program, a "per object" policy means there is one instance of the aspect variable for each target object (where the current target object depends on the join point context), etc. In JAMI, each variable can have its own instantiation policy, i.e. even variables within the same aspect module can have different instantiation policies.

By default, instantiation is implicit: new instances are automatically created when needed (using the default constructor of the specified variable type), i.e. on first use of the aspect variable in a particular join point context. However, explicit instantiations are possible as well, as we show in this example; the aspect variables *decorator* and *decoratee* defined in this example both have a "per association" instantiation policy. This policy means that the two variables are associated, such that when the value of one of them is known, the other can be looked up. In our implementation, this is done through the hash table shared by the two instantiation policy objects (see figure 4.6). The associations themselves have to be explicitly instantiated, as we will show in the "AssociationCreation" subview (figure 4.7). The implementation of "association variables" is similar to (and inspired by) the concept of "Association aspects" as proposed in [116].

Figure 4.7 shows how associations between *decoratee* and *decorator* objects are created. The top element, a SelectorAdviceBinding connects a point-

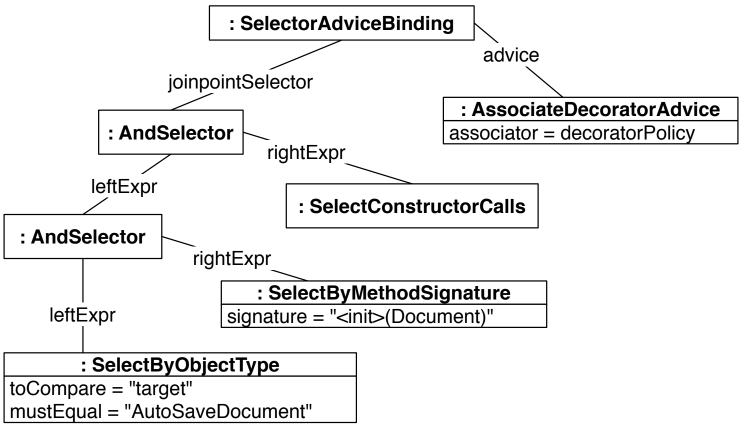


Figure 4.7: Association by intercepting constructor calls

cut definition (on the left) to an advice definition (on the right). The pointcut definition (called “join point selector” in JAMI) in this case consists of several “primitive” selection criteria, which are combined using the standard logic operators (i.e. *and*, *or*, *not*). The pointcut in this figure selects (a) constructor calls, (b) for which the type of the created object is `AutoSaveDocument`, and (c) the constructor being called has 1 argument of type `Document`. This pointcut is connected to a custom advice class – extending the framework – which explicitly creates the association between *decorator* (the constructed object) and *decoratee* (the value of the first argument). The class `Associate-DecoratorAdvice` is a DSAL-specific refinement (consisting of 20 lines of Java code), extending the default `AdviceAction` defined by JAMI, and is used to establish the association between *decorator* and *decoratee* objects.

To finish the example, figure 4.8 shows the definition of the call forwarding part of the aspect. The pointcut in this figure (on the left) selects method calls for which the target object occurs as a *decoratee* value in the association table, *except* those for which the caller object is a *decorator*. The latter is necessary to ensure that the *decoratee* object is reachable at all. Otherwise, the first call to the *decorator* object will lead to infinite recursion, as the *decorator* will at some point call the *decoratee*, see figure 4.5, and this call would then be intercepted again.

The advice attached to this pointcut is composed of several JAMI elements. The first, *BindObjectToVariableAction*, binds the value of the current

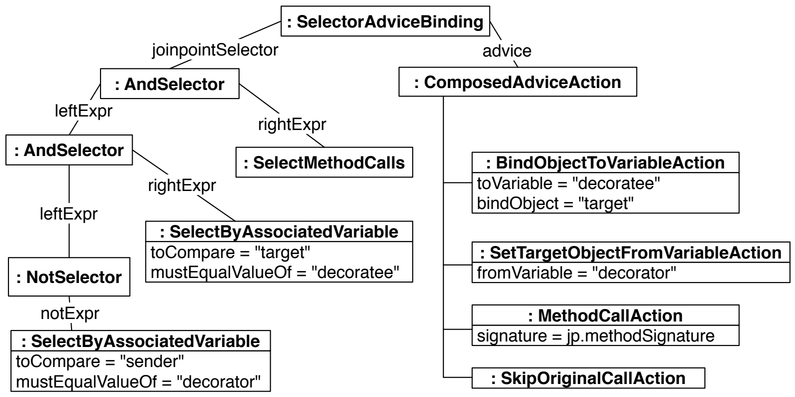


Figure 4.8: Forwarding calls from decoratee to decorator

target value to the variable *decoratee*. Next, the instruction *SetTargetObject-FromVariableAction* modifies the target object of the call to the value of the *decorator* variable, which can now be looked up through the corresponding aspect variable. In addition, we instruct the interpreter to execute the method call indicated by the join point context on the target set by the previous instruction. As we did not modify the signature of the method to be invoked, effectively a method with the same signature (as referred to by the current join point context) is called, except on a different object (i.e. the *decorator* object instead of the *decoratee*). Finally, the instruction *SkipOriginalCallAction* instructs the interpreter not to execute the original call. Except for the first action, which was implemented for use with the association instantiation policy, all these primitive advice actions are predefined as part of the JAMI framework.

When we initialize the aspect interpreter using the aspect described in the figures above, together with the "word processor" base program presented earlier, we can now write code that attempts to directly call the *decoratee* (as in listing 4.1, line 10), and still get the correct behavior: the call to the *decoratee* object (Document) is automatically forwarded to the *decorator* object (AutoSaveDocument).

Modularizing the decorator pattern

The simple aspect language defined above does not fully modularize the decorator pattern: associations are created in the base program using explicit constructor calls to a *decorator*. If we want to fully separate the decorator from the base code, we have to find a way to specify *when* a *decorator* has to be created, and to *which decoratee* object it should apply. This is non-trivial, as the decorator pattern is (usually) to be applied selectively, i.e. not simply to all objects of a particular class.

In this section, we look at one possible way to specify decorator associations using aspects. There are many possible ways to specify this, each having their own language design trade-offs. A benefit of using JAMI is that we can quickly prototype such proposals, allowing us to experiment with the resulting language using real programs – without needing a huge initial time investment.

Here, we propose a solution that lets programmers specify which particular instance variable (indicated by name) should be decorated, and by which decorator class. We specify this as shown in listing 4.3, line 1. The decorator is created and associated whenever a new value is assigned to the *decoratee* instance variable – variable "doc", in this case. In listing 4.3, this happens on line 9. The forwarding behavior stays the same as before, i.e. on line 11, the call will be forwarded to the auto-saving decorator.

```
 1 // DSAL code, specifying the decorator:
 2 decorate: Document WordProcessor.doc -> AutoSaveDocument;
 3 
 4 // Java code, containing no reference to the decorator
 5 public class WordProcessor {
 6  Document doc;
 7 
 8  public void testAutoSave() {
 9    doc = new Document();
10 
11    doc.addLine("AutoSaved!");
12  }
13 }
```

Listing 4.3: Decorator example: modularized version

To implement this, we only have to replace the "association creation" part as it was shown in figure 4.7. Instead, we create the structure as shown in figure 4.9. The pointcut combines 3 criteria using the logical AND operator: the join point must be of the type *field assignment*, must be contained by class *WordProcessor*, and have the name *doc*. The advice is a refined advice class (extending the JAMI model) that associates the value assigned to the field (i.e. the *decoratee*) to a newly created and initialized *decorator* object.

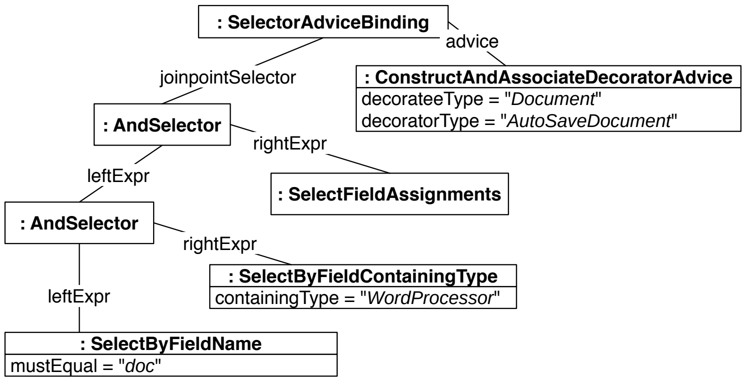


Figure 4.9: Modularized creation of a decorator object

As shown above, we could implement our proposal by writing a minimal amount of (new) source code: a method constructing the (partial) aspect AST such as displayed in figure 4.9, as well as the custom advice class *ConstructAndAssociateDecoratorAdvice*. Everything else is already handled by the (existing) framework and interpreter. It took us 4 days to design and implement the entire language, enabling both the enforcement as well as modularization of the decorator pattern.

4.4.2 Using the D/COOL domain-specific aspect language for synchronization

To show that JAMI can be used to conveniently accommodate more complex domain-specific languages as well, we implement the coordination aspect language “COOL”, which is part of the D language framework. The language is documented extensively in the dissertation describing this framework [98]. This chapter shows an example that uses only a (representative) subset of COOL, so that the example does not become overly complex; the entire COOL language proposal was however implemented using JAMI without any major issues [34].

Suppose we want to add a spellchecker to our word processor, which runs concurrently with the user interface by using a separate thread. To ensure correct behavior when multiple threads may access the document concurrently, we use a synchronization specification written in COOL, as shown in listing 4.4. By using COOL, we do not have to put any synchronization-related code in the Java source code itself.[7]

```
1 coordinator Document {
2   selfex addLine, setContent;
3   mutex {addLine, setContent};
4 
5   mutex {addLine, getContent};
6   mutex {addLine, wordCount};
7   mutex {setContent, getContent};
8   mutex {setContent, wordCount};
9 }
```

Listing 4.4: Using COOL to synchronize reader/writer access

Listing 4.4 specifies that we want to coordinate instances of class *Document*. Line 2 specifies that the methods *addLine* and *setContent* are self-exclusive; i.e. only 1 thread at a time is allowed to run those methods. Line 3 specifies that these methods are mutually exclusive in addition; i.e. only one thread may be active in either *addLine* or *setContent* at a given time. Note that self-exclusion does not imply mutual exclusion: without mutual exclusion, it could still occur that one thread is running *addLine*, while another is running *setContent*. Vice versa, mutual exclusion does not imply self-exclusion either: although only one of the methods in a mutual exclusion specification is allowed to run at the same time, multiple threads may be executing that one method, given the absence of selfex specifications that would prevent this.

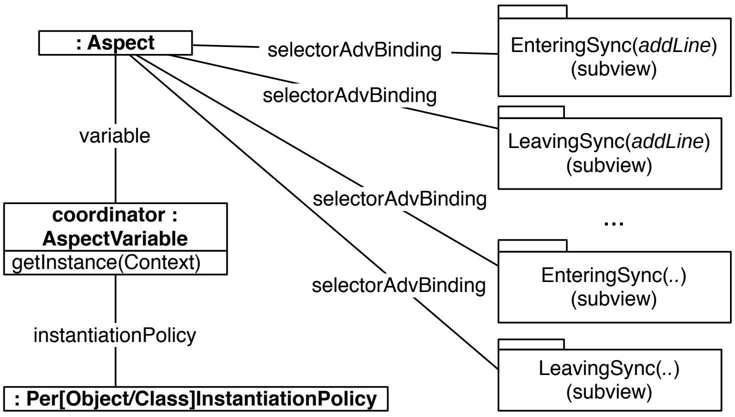
Lines 5-8 also specify pairs of methods that are not allowed to run at the same time - *addLine* and *setContent* are writer methods, and should not run at the same time as reader methods *getContent* or *wordCount*.

By default, COOL synchronizes method access *per object*, i.e. in the above example, several threads can still run method *addLine* at the same time, as long as they do so within different object contexts. COOL also allows to specify a *per class* modifier, which makes the synchronization "global" for the specified class.

[7]This is why COOL is called "aspect-oriented"; the Java program is oblivious to (i.e., does not contain any references to) the synchronization code, whereas COOL specifications contain quantified synchronization specifications, e.g., over multiple objects and methods.

Mapping to JAMI

We now describe a mapping of the subset of COOL discussed above to JAMI. First, for each method involved in a synchronization (i.e. selfex/mutex) specification, we calculate the set of methods that may not be entered while another thread is active within that method. For method *addLine*, this "exclusion set" contains *addLine* itself (because of the selfex specification on line 2), as well as methods *setContent*, *getContent* and *wordCount* (because of the mutex specifications on line 3, 5 and 6). How these exclusion sets are determined and why this ensures the "correct" results is precisely documented in [98]. To give an intuition of the definition, for each $selfex m$ expression, the indicated method *m* is added to its own exclusion set. For a $mutex(m_1, m_2, ..m_n)$ expression, for each method m_x in that expression (i.e., where $x = 1..n$), all other methods m_{notx} specified in the same expression are added to the exclusion set of m_x.



: Aspect
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
variable
coordinator : AspectVariable
getInstance(Context)
instantiationPolicy
: Per[Object/Class]InstantiationPolicy
EnteringSync(addLine) (subview)
LeavingSync(addLine) (subview)
...
EnteringSync(..) (subview)
LeavingSync(..) (subview)

Figure 4.10: Expressing COOL coordinators using JAMI concepts

A coordinator specification, such as found in listing 4.4, is modeled in terms of JAMI elements (see figure 4.10) as an aspect that defines one *AspectVariable* named *coordinator*. This variable has a "per object" or "per class" instantiation policy, depending on the specified granularity of the coordinator. Thus, the coordinator variable is shared between advices belonging to this coordinator, and can be used to regulate the synchronization. The instantiation policies in JAMI automatically give us the desired granularity as prescribed by the synchronization specification. Two selector-advice-bindings are defined for each method involved in a synchronization specification; one will be executed upon entering the method, one upon leaving.

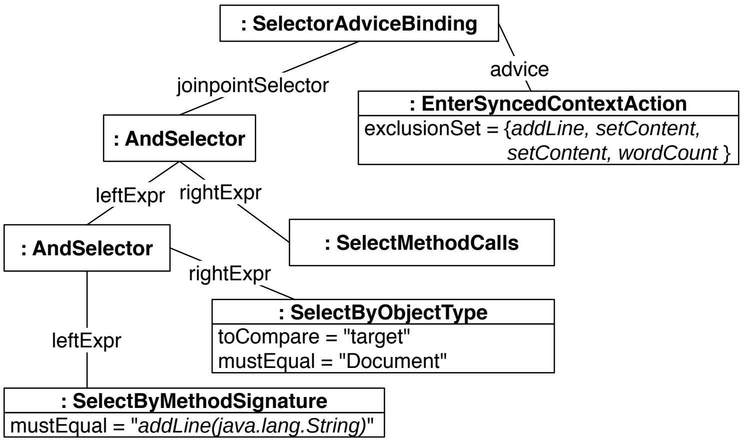


Figure 4.11: Entering a synchronization context: pointcut and advice

Figure 4.11 shows the object diagram for the selector-advice-binding executed upon entering method *addLine*. It matches only join points of type *MethodCall*, of which the target object is of type *Document*, and of which the signature of the called method is *addLine*. Before the call is executed, we execute the advice *EnterSyncedContextAction*, an advice class specific to this language.

We show the source code of this advice implementation in listing 4.5. Each instance of this class is *parameterized* by a method name and an "exclusion set" pertaining to that method. Instances of this advice implementation class are created while converting the AST of a COOL program to JAMI elements. At the same time, the exclusion sets are calculated in the way indicated earlier. When the advice is *executed*, i.e., invoked by the aspect evaluator that is part of JAMI, the advice retrieves (line 12-14) the *coordinator* aspect variable instance belonging to this specific context (i.e. object or class, depending on the instantiation policy). This *coordinator* object ensures that the synchronization "bookkeeping" itself is properly synchronized. While the advice holds a lock on this object (line 16-31), it can safely inspect the *MethodState* objects for this coordinator. For each method (involved in synchronization), such a *Meth-*

odState object tracks which threads are currently running that method. While other threads are active in any method in the exclusion set of the currently invoked method (line 21,22), the advice waits (releasing the lock on the *coordinator* while waiting) until this is no longer the case (line 24-26). When the loop is left, it means the method is free to run - after the advice registers the current thread with the corresponding *MethodState* object (line 30) and releases the lock on the coordinator object.

```
 1 public class EnterSyncedContextAction extends AdviceAction {
 2   MethodReference thisMethod;
 3   Set<MethodReference> exclusionSet;
 4
 5   public EnterSyncedContextAction(MethodReference thisMethod,
 6                           Set<MethodReference> exclusionSet) {
 7     this.myMethod = myMethod;
 8     this.exclusionSet = exclusionSet;
 9   }
10
11   public boolean evaluate(InterpreterContext metaContext) {
12     CoordinatorImplementation coord =
13       (CoordinatorImplementation) metaContext.getAspect().
14       getDataFieldValue(metaContext, "coordinator");
15
16     synchronized(coord) {
17       boolean shouldWait;
18       do {
19         shouldWait = false;
20         // Wait while any other thread is active in any method in our
              exclusionset
21         for (String excludedMethod : exclusionSet)
22           shouldWait |= coord.getMethodState(excludedMethod).
                isActiveInOtherThread();
23
24         if (shouldWait) {
25           try { coord.wait(); }
26           catch(InterruptedException e) { }
27         }
28       } while (shouldWait);
29       // This method is now allowed to run, register it
30       coord.getMethodState(thisMethod).enteringMethod();
31     }
32     return true;
33   }
34 }
```

Listing 4.5: Advice executed when entering a synchronized method

Similarly, another pointcut is created to intercept join points that occur upon *leaving* any of the methods involved in the synchronization specification. The object diagram is analogous to figure 4.11, except the pointcut now matches only join points of type *MethodReturn*, and executes an advice of type *LeaveSyncedContextAction*. We show the source of this advice in listing 4.6. The advice waits until it obtains a lock on the coordinator object within the given context (object or class, as in the previous advice), allowing it to update the synchronization "bookkeeping". Once the lock is obtained, it deregisters the current thread from the *MethodState* object for this method (line 15). It then notifies all waiting threads (if any), such that they can re-evaluate their waiting conditions (line 17).

```
 1 public class LeaveSyncedContextAction extends AdviceAction {
 2   MethodReference thisMethod;
 3
 4   public LeaveSyncedContextAction(MethodReference thisMethod) {
 5     this.thisMethod = thisMethod;
 6   }
 7
 8   public boolean evaluate(InterpreterContext metaContext) {
 9     CoordinatorImplementation coord =
10       (CoordinatorImplementation) metaContext.getAspect().
11       getDataFieldValue(metaContext, "coordinator");
12
13     synchronized(coord) {
14       // deregister this thread from running this method
15       coord.getMethodState(thisMethod).leavingMethod();
16       // Notify all threads, as potentially several may be allowed to
              continue
17       coord.notifyAll();
18     }
19     return true;
20   }
21 }
```

Listing 4.6: Advice executed when leaving a synchronized method

This concludes our implementation of (a subset of) COOL. The above essentially describes the same implementation mechanisms as defined in [98], except using an interpreter-based implementation instead of source-code weaving. We believe that this exercise demonstrates the usefulness of JAMI in several ways. First, we successfully mapped an existing, fairly complex language proposal to JAMI. In addition, it took minimal effort to build a functional prototype, which can be used on real base programs. It took 4 days to design and implement the prototype, and it consists of only 500 lines of code. The advice code as shown in listings 4.5 and 4.6 comprises the majority of the actual implementation mechanism; in addition we created a parser for the subset of COOL used in this example (ca. 100 lines of code), an object-based representation of this AST (ca. 100 lines of code), as well as code to map such object-based COOL ASTs to “aspect-AST” structures such as shown in figure 4.10 and 4.11 (ca. 200 lines of code). Thus, JAMI proves useful as a “testbed” to prototype DSALs.

4.4.3 An experimental DSAL to implement caching

As a final example, we show the implementation of an experimental language that introduces a modular way to specify caching of method return values (also called *memoization*).

Methods (or functions) to which memoization is applied, traditionally have to conform to the following conditions: (1) the method depends on its (input) parameters only; (2) given the same input parameter values, it should return the same result every time; (3) the method should have no side effects. Our implementation maintains the latter two requirements. However, the first requirement is often violated in object-oriented programming, since results of a method call often depend on values of instance variables (within the same object) or specific method calls (on the same object). Therefore, our implementation extends the notion of memoization as defined above, by allowing cached results to be invalidated when the value of particular fields change, or when particular methods are called.

In our example application from figure 4.4, the method *wordCount* is a good candidate for memoization, as repeatedly calculating the number of words – even when the document has not changed – can become quite time consuming on large documents. The method has no side effects, but depends on the value of instance variable *content*. This variable is written by method *setContent*, which contains the statement “this.content = newContent;”. The method *addLine*, containing the statement “content.add(line);” does not overwrite the instance variable itself; it does however modify its contained object structure. Therefore, calls to method *addLine* should also invalidate the return value of *wordCount*.

We specify the above using a domain specific aspect language as shown in listing 4.7.

```
cache Document object {
 memoize wordCount,
 invalidated by assigning content
            or calling addLine(java.lang.String);
}
```

Listing 4.7: Example specification of a memoization aspect

This specification means the following: apply a caching aspect on each *Document* object (line 1). This caching aspect will memoize the return value of method *wordCount* (line 2). The cache will be invalidated when a new value is assigned to instance variable *content* within the corresponding *Document* object (line 3), or when the method *addline(..)* is called on the *Document* object (line 4).

Mapping to JAMI

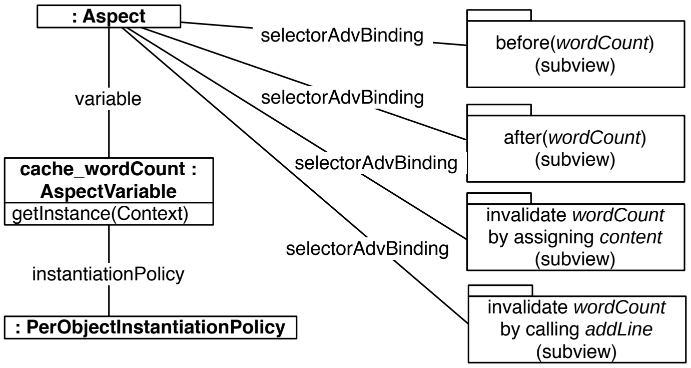

: Aspect
variable
cache_wordCount : AspectVariable
getInstance(Context)
instantiationPolicy
: PerObjectInstantiationPolicy
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
selectorAdvBinding
before(wordCount) (subview)
after(wordCount) (subview)
invalidate wordCount by assigning content (subview)
invalidate wordCount by calling addLine (subview)

Figure 4.12: Mapping a caching aspect to JAMI concepts

We now show how to map the specification shown in listing 4.7 to JAMI. As figure 4.12 shows, we create an aspect variable of type *Cache* for each *memoize* declaration. Its instantiation policy can again be specified as per object or per class - in the example above, we want to cache the return value of method *wordCount* for each object of type *Document*. The class *Cache* models a simple wrapper object that can store and retrieve an object, as well as clear its currently stored value, to be called when the cache should be invalidated.

For each memoized method, we need a pointcut that intercepts calls to that method, coupled to an advice that returns the cached value (if one is stored). Another pointcut intercepts *returns* from the memoized method, coupled to an advice that stores the return value in the cache. Finally, a pointcut is needed for each cache invalidation specification, coupled with an advice that clears the cache. In this example there are two such pointcuts, corresponding to the invalidation specifications in line 3 and 4 of listing 4.7).

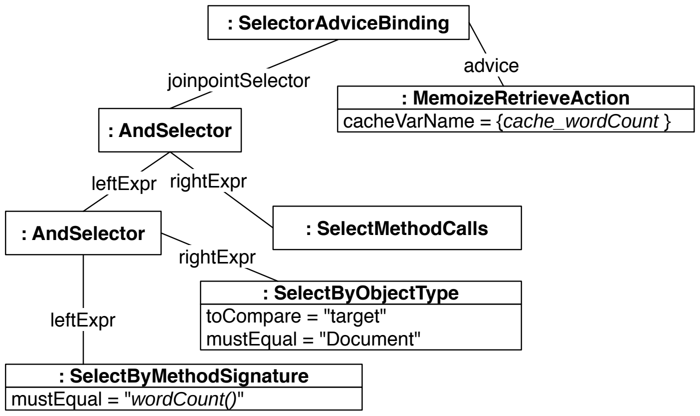


Figure 4.13: Selector-advice binding for retrieving cached values

Figure 4.13 shows the pointcut specification to intercept calls to a method of which the results should be cached. The advice that is executed is shown in listing 4.8. First, the advice retrieves the aspect variable corresponding to this *memoize* declaration (line 9,10). Note that the name of this aspect variable is specified as a parameter of the advice constructor, and is initialized when the specification of this *memoize* declaration is converted to JAMI elements. If the cache currently contains a value (which means it must have been set after a previous call), we instruct the interpreter not to execute the original call after it finishes executing this advice (line 14), and instead to set the return value to the value found in the cache (line 15).

```
1  public class MemoizeRetrieveAction extends AdviceAction {
2    private String cacheVarName;
3
4    public MemoizeRetrieveAction(String cacheVarName) {
5      this.cacheVarName = cacheVarName;
6    }
7
8    public boolean evaluate(InterpreterContext metaContext) {
9      Cache cache = (Cache)metaContext.getAspect()
10       .getDataFieldValue(metaContext, cacheVarName);
11
12     if (cache.hasValue())
13     { // Use cached value!
14       metaContext.setExecuteOriginalCall(false);
15       metaContext.setReturnValue(cache.getValue());
16     }
17     return true;
```

```
 }
}
```

Listing 4.8: Advice: retrieving a cached value

After the method returns, the advice in listing 4.9 is called, which stores the return value of the method.

```
 1 public class MemoizeStoreAction extends AdviceAction {
 2  private String cacheVarName;
 3 
 4  public MemoizeStoreAction(String cacheVarName) {
 5   this.cacheVarName = cacheVarName;
 6  }
 7 
 8  public boolean evaluate(InterpreterContext metaContext)
 9  {
10   Cache cache = (Cache)metaContext.getAspect()
11     .getDataFieldValue(metaContext, cacheVarName);
12 
13   cache.setValue(metaContext.getReturnValue());
14   return true;
15  }
16 }
```

Listing 4.9: Advice: storing a cached value

First, the advice retrieves the cache variable (line 10,11). Next, it stores the return value of the called method, which can be obtained through the interpreter context (line 13). Note that we do not take method parameters into account in this implementation (fortunately, the method *wordCount* does not have any). This is done to avoid cluttering the example; adding this behavior would be straightforward.

To finalize our example, we show one of the pointcut-advice-bindings used to invalidate the cache. Figure 4.14 shows a pointcut that will match field assignments, but only to the field named *content*, and when the assignment takes place within an object of type *Document*. The advice calls the method *clearValue* on the aspect variable *cache_wordCount*. The cache can also be invalidated by particular method calls; this binding reuses the same advice, but has a pointcut selecting the specified method, in a way equivalent to many of the examples shown above, e.g. in figure 4.13.

It took us 3 days to design and implement this language, including a parser for specifications as shown in 4.7 and an automated mapping of the parsed structure to the JAMI object diagrams as shown in this section.

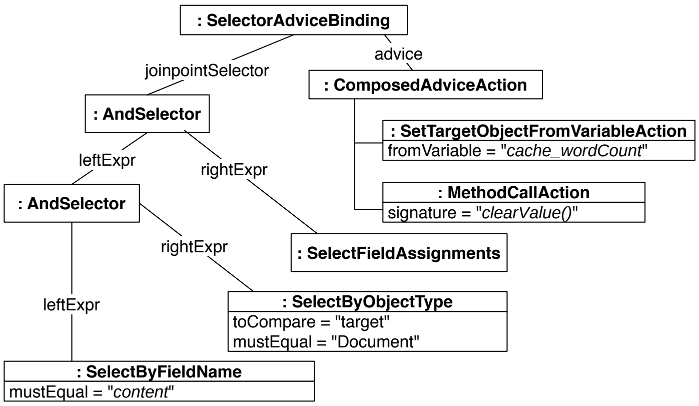


Figure 4.14: Selector-advice binding for invalidating cached values

4.5 Composition of multiple DSALs

As each DSAL is designed to address concerns within a particular problem domain, and many applications will deal with concerns from several such domains, it would make sense to combine the use of several domain-specific aspect languages within a single application[8]. Implementing this is not straightforward however, as partial programs expressed in several languages have to be composed into a single combined, working application. Even if this is technically feasible (which is not necessarily the case), running the combined application may reveal unexpected and/or undesired results.

In this section, we discuss how several aspects written in different DSALs (all implemented using JAMI) can be composed and used within the same application. We discuss several difficulties that may occur in this case, and explain how JAMI can help to address these issues.

4.5.1 DSAL composition in JAMI

In general, the composition of multiple aspect languages is far from trivial. As an example, consider the common implementation of aspect languages by transformation of the source code or byte code representation of the base pro-

[8]Note that the entire discussion about the composition of DSALs technically also holds for the composition of general purpose aspect languages, or a mixture of these. However, we believe composition of DSALs is much more realistic to expect, hence we focus on this.

gram (where each of these aspect language implementations may, or may not, share a common infrastructure). This requires the sequential execution of aspect language implementations over the incrementally transformed base code. Typically, such byte code transformations are not commutative, meaning that the behavior of the resulting program could vary, according to the –normally undefined– order in which the aspects are applied to the base program.

In section 4.4, we have shown how aspects written in several DSALs are mapped to JAMI elements. Such aspects, expressed in terms of JAMI elements, or refinements of JAMI elements, can be combined within a single application – even if they originate from different aspect languages. This is enabled by the common runtime platform provided by JAMI.

This platform defines common abstractions and a common data structure for the representation of aspects (e.g. in terms of pointcut expressions, pointcut-advice bindings, ordering constraints, etc.). Further, the framework imposes a unified high-level control flow for the execution of aspects, as described in figure 4.3. At the same time, while adopting these predefined abstractions and high-level control flow, for each language there is a large freedom to define in varying ways how e.g. pointcuts can be defined and matched.

Thus, using JAMI, it is possible to execute aspects written in different DSALs within a single application. This does not require any tailoring or design decisions that are specific to the other DSALs that are combined. However, having a common platform to run several aspect languages does not guarantee that resulting applications will exhibit the “correct” or “desired” behavior. As is the case with aspects written in a single language, interactions or interference may also occur between aspects written in different DSALs, for reasons inherent to these DSALs.

This phenomenon has also been observed before: e.g. in [100], two categories of aspect interactions are distinguished[9]:

- *co-advising*: the composition of advice expressed in multiple aspect languages at a shared join point.
- *foreign advising*: this corresponds to the notion of “aspects on aspects”, where advice expressed in one aspect language may apply to a join point associated with the execution of advice expressed in another aspect language.

[9][100] defines these terms using a description based on weaving semantics, we reformulated these in terms of aspect execution

In the remainder of this section, we first explain the design rationale of the composition mechanisms supported by JAMI, in subsection 4.5.2. We then discuss the issue of co-advising, in subsection 4.5.3, and finally foreign advising, in subsection 4.5.4. These problems are all illustrated by combining the aspects shown in section 4.4 within the same application.

4.5.2 Design rationale of JAMI's advice composition mechanisms

As discussed in section 4.3, JAMI offers two complementary advice composition mechanisms. First, it implements a generic ordering constraint mechanism as proposed in [105, 106]. At shared join points, constraints may limit which advices are applicable at any given moment during the evaluation of aspects. Such constraints may be conditional, and may for example depend on which advices where already executed (at the same join point). Even so, the application of constraints may still leave several advices eligible for execution. Second, JAMI therefore supports a "scheduling" interface to determine the further selection of advice execution. Different strategies can be implemented to disambiguate the selection of advice. Our default implementation picks an arbitrary element from the set of applicable bindings, and in addition prints a warning that the program is potentially ambiguous.

In addition, the aspect evaluator can decide to cancel further advice executions at a given join point, if requested to do so by particular advice actions[10], see e.g. the *SkipOriginalCallAction* and *CancelOtherAdvicesAction* advice specifications explained in section 4.3.3.

Constraints are specified over pointcut-advice-bindings, as these are the primary elements over which we want to express ordering criteria — as opposed to ordering specified per advice, pointcut, or aspect. The reason is that advices (and pointcuts) can be reused in several bindings; the desired ordering may differ per binding. In addition, pointcut-advice-bindings are the most "low-level" construct within JAMI to which an ordering can be applied. Ordering between aspects can be expressed in terms of (several) constraints between selector-advice-bindings. These are examples of *program-level* constraints. *Language-level* constraints are also expressed in terms of (several) constraints between individual selector-advice-bindings. The framework could include "convenience methods" to allow to directly express constraints between all bindings within particular aspects, or between all bindings

[10]This corresponds to the run-time detection and resolution of aspect interactions in [120].

of all aspects written in a particular language. We believe that the constraint mechanism adopted by the framework can be used as the basis of any such higher-level ordering mechanism.

Constraints are decoupled from the "aspect modules" (as shown in e.g. figure 4.6), and are instead kept as a separate set of entities within the aspect evaluation framework. This enables the specification of constraints between pointcut-advice-bindings that are part of several aspects, or that even originate from several aspect languages. It would be possible to define a "constraint language" that can express constraints over aspects from several different DSALs, although this requires that DSALs make it possible to identify (e.g. by name) the entities to which an ordering may need to be applied.

4.5.3 Co-advising

When multiple pointcuts match at the same join point, the order in which advices bound to these pointcuts are executed may lead to different behavior, if there are dependencies between the aspects [106, 46]. Reversely, in the absence of any ordering specification at shared join points, the application behavior may be non-predictable and undesirable.

The above is also true if the shared join points originate from programs written in different aspect languages. For individual languages, many mechanisms exist to deal with this — for example, [49, chapter 4] discusses behavioral conflicts among aspects, and [81] describes a number of existing approaches.

However, when pointcuts originate from different languages, there are two additional issues:

- We need improved or additional mechanisms to compose advices from different aspect languages. The reason is that we (want to) assume DSALs to be developed independently, so that aspects written in a particular DSAL are likely (and preferably) unaware of those written in another DSAL. JAMI supports a uniform constraint model (see section 4.3.3) that facilitates ordering constraints *within* as well as *between* languages. We demonstrate this below.

- There is a distinction between *language-level* and *program-level* composition [100]. In particular for DSALs, composition constraints may be specific to a combination of DSALs, and should apply to all aspects written in those DSALs (i.e. language-level constraints). However, it may

–in addition– be possible that some constraints are program-specific (i.e. program level).

Example: composing the synchronization and caching aspects

When we deploy the aspects for synchronization (shown in listing 4.4) and caching (listing 4.7) within our original application (see figure 4.4), we observe that several shared join points occur, as most calls to methods within class *Document* are advised by both aspects. Therefore, we need to determine in what order these advices should be executed.

As an example, we consider the join point that occurs when returning from method *wordCount*.

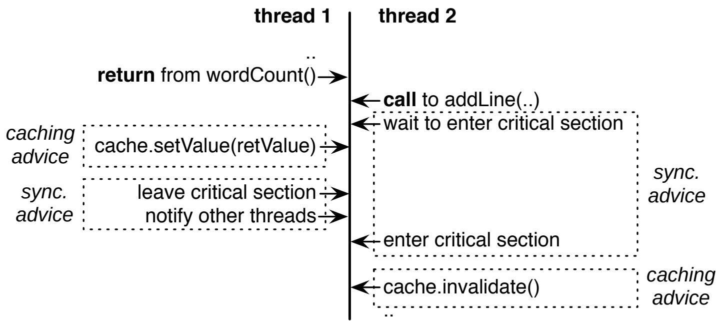


Figure 4.15: Concurrent execution; correct advice ordering

At this join point, a caching advice will store the value that was returned by the method. The synchronization advice leaves the critical section that was entered before the method was executed, as shown in listing 4.6. In this case, the caching advice –at the end of a method– should be executed before the synchronization advice. This is illustrated in figure 4.15, whereas figure 4.16 illustrates a specific scenario of two threads where –in both cases– the synchronization advice precedes the caching advice. In the latter case, a different thread executing a *writer* method may invalidate the cache as soon as the critical section is left, while subsequently the caching aspect stores an (already invalidated!) value in the cache. In that case, the next call to *wordCount* would return a cached value that is incorrect.

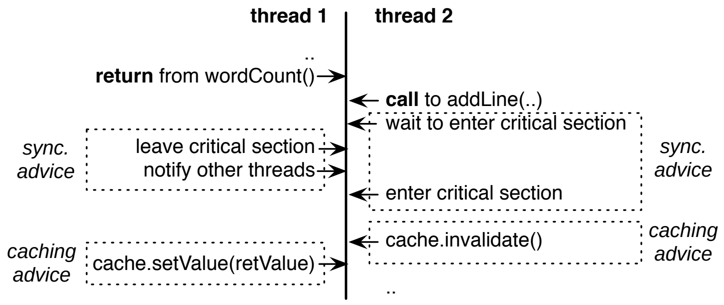


Figure 4.16: Concurrent execution; incorrect advice ordering

To generalize the example, we observe that any caching advice should occur within the critical sections as imposed by the synchronization advice. Specifically, for advices executed at a shared *MethodCalljoinpoint*, the synchronization advice should have precedence, while at a shared *MethodReturnJoinpoint*, the caching advice should have precedence. This is an example of a language-level composition constraint.

Example: composing further with the decorator aspect

As a test case, we illustrate the composition of three DSALs in JAMI, including a join point where advices from all three languages must be applied. In addition to the synchronization and caching aspects discussed above, we add the decorator aspect as shown in listing 4.2 to our application. This means that for objects of type *Document* that are decorated, all calls to methods within *Document* will be forwarded to *AutoSaveDocument*. Thus, shared join points may occur where all three aspects (each originating from a different language) want to execute an advice. In this case, the desired behavior is more complex than simply ordering the advices.

When a call is redirected by a forwarding advice (as defined by a decorator aspect), the original call does no longer lead to the execution of a method –as specified by the *SkipOriginalCallAction* in figure 4.8. Therefore, after the execution of the advice of the decorator aspect (in this case), there is no method execution join point active. Effectively, this means that no other advices should be executed at this join point. This implies that any advice from the decorator aspect should be executed before advices specified by both other languages. After all, it would be illogical to cache or synchronize the execution of a method that will not actually be executed at all.

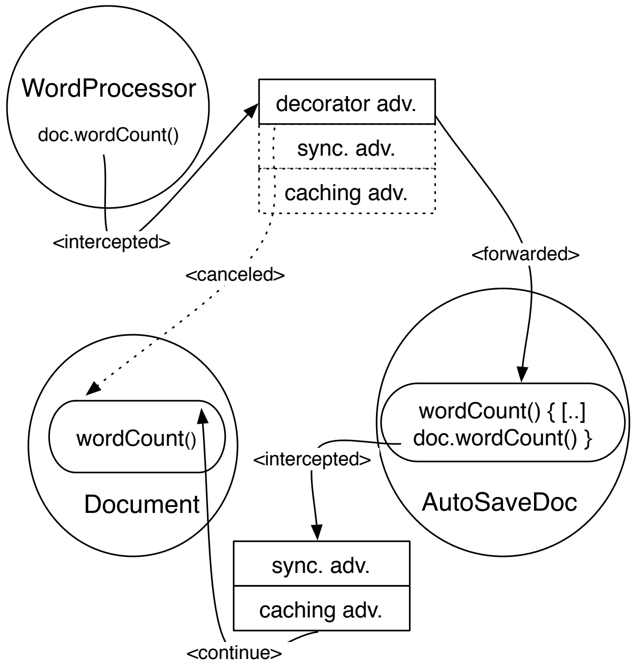


Figure 4.17: Control flow combining aspects from 3 DSALs

In figure 4.17, we illustrate that the intended behavior is obtained by ordering the advices per language as described above. When method *Document.wordCount* is called, all three aspects match this join point. The decorator advice will be executed first, and forwards the call to *AutoSaveDoc.wordCount*, which results in the *cancelation* of the original call. In addition, the advice has to ensures that no other advices are executed at this join point. Subsequently, the implementation of *AutoSaveDoc.wordCount* calls *Document.addLine* again. The decorator aspect does not match this (new!) join point, as internal calls from decorator to decoratee are not intercepted by the decorator DSAL (as defined by the pointcut expression in figure 4.8). The other two aspects both match this join point however, and are executed in an order such that caching takes place within the critical section of the synchronization advice.

Implementation in JAMI

As discussed above, for our example we want to specify language-level composition based on the originating language of each pointcut-advice-binding. We do not need any program-level constraints, in this case. Therefore, we simply (programatically) create constraints between all pointcut-advice-bindings, such that caching advices occur within the critical section created by synchronization aspects (if the advices apply at the same join point), and decorator advices get even higher precedence.

Finally, we extend the "forwarding" advice of the decorator aspect (as shown in figure 4.8) with an advice action `CancelOtherAdvicesAction` (see section 4.3.3), which instructs the scheduler to cancel any other advices at the current join point. This action should be part of the decorator language, as its "forwarding" advice effectively negates the occurrence of the join point at which it is executed. Therefore, no other advices should be executed at that join point.

A working implementation (in JAMI) that composes aspects written in all three DSALs discussed in this chapter – including the constraints as discussed in this section, is downloadable as part of the example discussed throughout this chapter [8].

This example illustrates that with JAMI, composition of more than two aspect languages is supported, even in the presence of delicate interdependencies; JAMI supports the expression of the necessary composition constraints such that the intended effect of each aspect is preserved.

4.5.4 Foreign advising

Another way to compose aspects, which we did not discuss so far, is the application of aspects on aspects. In particular, the application of advice written in one DSAL on another advice written in a different DSAL, is also called "foreign advising" [100].

Within JAMI, advices expressed in any DSAL are eventually executed in terms of the same kind of instructions used by the base program (e.g. using objects and method calls). Such advice instructions can again be advised, like any normal base language construct. A weaver-based approach must make sure to weave aspects (written in several DSALs) in a particular order, to ensure that the effects of one DSAL can be advised by another –or to ensure that they are *not* advised by another DSAL. As no weaving takes place within JAMI, the execution of advice is simply a runtime occurrence that can be intercepted like any other join point, if needed.

To test the application of “foreign advice” within JAMI, we apply a decorator aspect to the caching aspect discussed in listing 4.7. Our example decorator class logs the actions of the caching aspect, which can be useful when e.g. debugging the caching aspect. We use the following decorator specification:

```
decorate: MemoizeRetrieveAction -> MemoizeLogDecorator
```

This specification intercepts all executions of the memoization advice, and redirects them to an instance of class *MemoizeLogDecorator*, which logs the activity of the caching aspect and then forwards the call back to the original advice method. Thus, it applies an advice to the caching advice.

We remark that the advising of advice can quite easily lead to infinite interception loops. Therefore, by default, we exclude all framework classes (and extensions thereof, such as class *MemoizeRetrieveAction*) from interception by the framework. However, in cases where interception is desired, an annotation can be used to indicate that a particular class *should* be considered for interception. On the other hand, method executions within the base language that are triggered by advice executions are by default intercepted.

4.6 Discussion

In this section, we discuss some of the important design decisions and implementation characteristics of JAMI.

4.6.1 Efficiency vs. flexibility

JAMI is designed to provide maximum flexibility while designing or testing new aspect languages or language features. As a result, in cases where we had to choose between flexibility and efficiency, we chose the former. For example, the interception of all joinpoints within an application is not very attractive from a performance point of view. However, using this mechanism makes it much simpler to experiment with pointcuts that depend on complex combinations of runtime state (including historic information). While designing a language, the developer can ignore such details as e.g. deciding which part of a pointcut can be evaluated statically, and which remaining dynamic checks have to be woven at such statically determined “shadow join points”. In addition, when composing several DSALs, the effects of one DSAL may influence the static evaluation of pointcuts in other DSALs. Using JAMI, such issues do not occur, as the entire pointcut evaluation takes place at runtime.

However, when creating an efficient language implementation is a primary design goal rather than the prototyping of (new) language features, the use of aspect-aware virtual machines or weaving-based approaches, such as *abc* [18], may be more suitable.

A research framework that aims specifically at modeling efficient implementation techniques of AO mechanisms is the Aspect Language Implementation Architecture (ALIA) [24, 26]. This framework is structured similarly to JAMI, but contains more detailed information necessary to enable efficient implementations, such as discussed in [25]. Because of the structural resemblance, it should be possible to (manually) map JAMI models to ALIA models. In this way, one could use the JAMI framework to experiment with the language design, and subsequently use the ALIA framework to implement the language to be as efficient as possible.

4.6.2 Modeling different types of advice

JAMI does not make a distinction between different types of advice, such as *around*, *before* or *after* advice. Join points *before* and *after* method calls are simply considered different (kinds of) join points altogether (i.e. *MethodCall* and *MethodReturn* join points). The two defining features of *around* advice (as it is known in e.g. AspectJ) are the ability (1) to share state between the *before* and *after* advice parts around a method call (using local variables within the advice definition), and (2) to decide whether or not to execute the code at the location of the current join point (determined by whether or not the advice "executes" a *proceed* construct). Both can be accomplished using JAMI as well; the first by using aspect variables with an appropriate instantiation policy, the second by using meta-advice actions that instruct the interpreter whether or not to execute the code at the current join point. For an example using both features, see section 4.4.3. We believe that our advice model allows more freedom over the specification of e.g. advice ordering and the semantics of particular advice actions, while it is also able to accommodate existing advice models, as indicated above.

4.6.3 Modeling advice implementations

For maximum flexibility, our framework supports several ways to implement advice. In this subsection, we discuss the tradeoffs involved when choosing between these.

One way to implement advice is by using the primitive advice "building blocks" provided as part of the framework. An example is shown in figure 4.8, which uses several predefined JAMI elements to construct a compound join point selector that selects calls to *decoratee* objects. The benefit is that this results in an advice representation that is relatively easy to analyze (using automated tools), as the actions that the advice may execute (e.g. skipping a method call) are directly visible in the advice representation. In addition, these primitive advice actions can be reused by several language implementations, as we have demonstrated through the examples in section 4.4.

In addition, advice can be implemented by creating implementation classes that are custom-built for a specific DSAL implementation. An example can be found in listing 4.8, which shows the implementation of a cache retrieval advice action (`MemoizeRetrieveAction`). This example contains a control statement (an *if*-statement, listing 4.8, line 12), for which no predefined JAMI advice component exists. The reason is that adding such components would eventually lead to duplication of most of the base language, which is clearly not the purpose of this framework. It will be harder to analyze custom-defined advice classes - however, we foresee that DSAL implementers could provide semantic specifications (e.g., using annotations) as part of their custom advice implementation to help facilitate this.

4.6.4 Future work: static analysis

When advices are expressed using the "advice actions" provided by JAMI, it would be possible to reason about potential interaction/interference in an automated way. For example, analyzing the advice structure may enable the classification of advices into different categories, such as for example those suggested by Rinard [115]. Some combinations of advice actions are likely to be unintended when they occur at shared join points (without explicit ordering constraints). For example, when one advice cancels the execution of others, or when one advice cancels the execution of a method, the ordering of advices is very likely to lead to different results.

To analyze the effects of advice in an automated way (e.g. to detect conflicts), it will help to give a (formal) semantic specification of the effect of each advice action. We suggest that an operation-resource model as proposed in [49] may provide a convenient abstraction level; using this model, each advice action has to indicate which resources (e.g. the join point context, target object, variables, etc.) it accesses and what kind of operations (e.g. changing, reading, writing, locking, ..) it executes on those resources. Subsequently, several kinds of formal analysis methods may be applied to the resulting model. The particular types of operations and resources used, as well as the analysis methods applied to them could even be tailored to particular domains. This can be an additional strength of using DSALs rather then general-purpose aspect languages: as they are limited to a specific problem domain, it may be easier to define semantic analysis rules over the advices they apply.

4.7 Related work

In [103], Masuhara and Kiczales propose the Aspect Sand Box, an interpreter framework to model aspect mechanisms. Using this framework, the effects of aspects are defined in terms of weaving semantics. The weaving process is modeled by extending or modifying the interpreter of a base language that models a single-inheritance object-oriented language (which can be seen as a core subset of Java). In comparison, JAMI defines a common runtime environment for aspects, which allows us to express explicit ordering constraints between advices, even if they are written in different DSALs, and enables the deployment of multiple aspect languages within a single application. As discussed in section 4.5, it would be harder to define a single weaver that models the composition of multiple languages. Essentially, JAMI can be seen as an implementation of a single, parameterized (by using different predefined framework classes) aspect composition process as indicated in the future work section of [103]. In this sense, JAMI can be seen as an elaboration on the ideas proposed in the Aspect Sand Box project, thus enabling the implementation of additional features such as mentioned above.

The AspectBench Compiler (*abc*) [18] is a workbench that – like JAMI – facilitates experimentation with new (aspect) language features. Unlike JAMI however, it focuses mainly on extensions to AspectJ, and strives to provide an industrial-strength compiler architecture that facilitates efficient implementations of extensions to the AspectJ language. In contrast, while designing JAMI we specifically tried to avoid design decisions that would limit the flexibility of our framework, as discussed in section 4.6.1. In addition, *abc* is not designed to handle composition between multiple languages.

In [19], Bagge and Kalleberg propose to implement DSALs by creating a library that implements a program transformation system (cf. weaver), in addition to a notation that “configures” the behavior of this library. The paper does not discuss the composition of multiple DSALs, or the ordering of advices at shared joinpoints.

In [30], Bräuer and Lochmann describe how to integrate multiple DSLs based on a common semantic metamodel, using an MDA-based transformation approach. Like JAMI, this provides a model to integrate modularized specifications written in several DSLs. However, the paper aims to propose a common semantic model for the composition of multiple DSLs in general – as opposed to expressing *crosscutting* functionality (i.e. aspects) in particular, as is the focus of JAMI. Likely as a result of this, the paper does not discuss the interference issues that may result from composing multiple crosscutting specifications.

The work from Kojarski and Lorenz [100, 93, 92] is strongly related to ours; in particular, they also investigate the issues around the composition of multiple aspect languages.

In [93], seven interaction patterns among features of composed aspect languages are described. Some of these, such as *emergent advice ordering*, are also discussed in this chapter. However, because (1) JAMI introduces its own set of *abstract features*, such as selector-advice bindings, and (2) in our interpreter-based approach, individual aspect languages are not translated into base language terminology, hence, there is never accidental interaction, not all interaction patterns are applicable. However, the proposed analysis approach could also be applied in the context of our work.

In [100, 92], the AWESOME framework is described; instead of an interpreter-based approach, this adopts a weaver-based approach, that also addresses foreign advising, and language-level, but currently –according to [100] – not program level co-advising (which we presented in section 4.5.1).

The *Reflex* AOP kernel [121, 120] is also closely related work; it is a reflection-based kernel for AOP languages, with a specific focus on the composition of aspect programs. To this extent, it provides an (extensible) set of composition operators, which can be used when translating an aspect specification to a representation in terms of the kernel-level abstractions. Although there are many similarities with JAMI, a key difference of the current implementation is that it is weaving-based, rather than interpreter-based. Mostly due to this, the support for foreign advising is limited (as e.g. exemplified in [93]).

The XAspects project [117] implements a system to map DSALs to AspectJ source code. The approach addresses the need to compose aspects written in multiple DSALs, but does not provide explicit mechanisms to deal with interactions between aspects, other than suggesting the use of the AspectJ *declare precedence* construct. Compared to this, JAMI offers more elaborate ways to specify the composition of aspects.

In [32], Brichau et.al. propose the definition and composition of DSALs ("Aspect-Specific Languages") using Logic Metaprogramming. Although their approach is not based on a typical OO framework, it does allow the reuse and refinement of aspect languages. It is based (in [32]) on static source code weaving (by method-level wrapping). The composition of aspect languages (program level composition is not supported) is achieved by explicit composition of languages into new, combined languages. In our opinion, this is less flexible, as it requires explicit composition for each configuration of aspect DSALs that occur in an application, and the late addition of a new aspect language in a system may not be possible without restructuring the composition hierarchy.

Dynamic AspectJ [16] attempts to bring dynamic resolution of advice ordering at shared join points to the world of (typically statically woven) AspectJ. To this aim, the authors introduce a thin interpretation layer in the statically woven code introduced by the AspectJ compiler. This layer enables the evaluation of constraints when multiple advices are "scheduled" to be executed at the same join point. However, the constraints are specified in an imperative way, as opposed to the declarative constraints used in this chapter, which are based on a constraint model first introduced in [106].

4.8 Conclusion

In this chapter, we introduced an aspect interpreter framework aimed at the prototyping of (domain-specific) aspect languages. We demonstrated the framework by implementing three domain-specific aspect languages. Using our framework, it took only 3-4 days (per DSAL) to create functional prototypes of these languages. Aspects written in these DSALs can be composed with regular Java programs at runtime, in an interpreted style.

We have used JAMI in a programming language course to teach the common aspect language concepts and various implementations thereof. As part of this course, students successfully developed small DSALs within limited allotted time. This supports our claim that JAMI can be used to prototype DSALs while requiring relatively little effort, even including the learning curve of the framework itself.

We contribute the effectiveness of JAMI as a framework for prototyping DSALs in large part to its flexibility and expressiveness. For example, as aspects are completely dynamically evaluated, it is easy to experiment with pointcuts that express complex selection criteria over the runtime state. In addition, our support for "aspect state" using variables that each may have different instantiation policies provides a flexible way to implement aspect language features, while requiring relatively little effort.

We have shown that our framework supports applications composed of aspects written in several DSALs. In addition, we have discussed interactions that may occur when combining multiple DSALs, and demonstrated mechanisms implemented as part of JAMI to specify aspect composition – also of aspects written in different languages.

The complete framework as well as the examples shown in this chapter can be downloaded from the JAMI website [8].

5

Constructing Composable Composition Mechanisms

5.1 Goals and motivation

The history of programming languages shows a continuous search for new — presumably better — composition techniques. The typical aim of such techniques is to find better ways for structuring increasingly complex software systems into modules that can be developed and reused independently.

Composition operators are language mechanisms that let programmers compose behavior and/or data, defined as separate entities, by means of a *composition specification*. An example of a composition operator is *function application* (viz., calling a function or method). This operator allows the invocation of functionality that is defined separately (as a function definition), by means of a call statement (fulfilling the role of *composition specification*). Other examples of composition operators include inheritance (in many different styles), delegation, pointcut-advice mechanisms, composition filters, mixins, traits, etc.

Most languages adopt a fixed set of composition operators, typically with explicit notation and predefined semantics. In case a language does not provide a composition operator with the desired compositional behavior, programmers may need to write workarounds in their applications, or may attempt to introduce new operators as macros, libraries, frameworks or language extensions. However, in many cases, the newly introduced operator may not be fully integrated with the language, and thereby the composition semantics may not fully benefit from the expressive power of the language.

For example, in object-oriented languages that do not support explicit *delegation* [97], similar functionality can be simulated by *forwarding* calls from one object to another. However, when forwarding a call in this way, the "self" or "this" object context changes to the object to which the call is delegated. In contrast, in languages that support explicit delegation, the “self” context does not change when delegating an operation. This difference leads to the so-called “self problem” in languages that do not support explicit delegation [97]. That is, “self”-calls within the object to which a call is delegated, will always be handled by that object, rather than the original “interface” object to which the call was originally sent (before being delegated). Although it is possible to work around this by passing the original interface object as an extra call parameter, this is an unsatisfactory solution. For example, this workaround necessitates changes to the interface of affected methods, which may be undesirable[1]. In addition, one of the motivations for using object-based languages is their built-in support for object-based abstractions, such as the “self” object reference. If the use of such language primitives is restricted, this limits the benefits of using a language that supports such primitives in the first place. Thus, by using such “second class” implementations of composition operators, the expressiveness of the language can no longer be used to its full potential, at least in areas that are affected by the use of such operators [122].

The availability of only a limited set of composition operators causes additional issues, which have previously been recognized in the context of Component-oriented programming. The aim of Component-oriented programming is that ideally, it should be possible to compose (complex) software from a set of smaller components, which are easier to comprehend and maintain than the system in its entirety, and which can be evolved or reused independently of other components. In reality however, most existing languages have a bias towards one kind of decomposition of software systems, which also imposes constraints on the viability of particular evolution scenario’s, or in other words, the *extensibility* of software [41].

For example, in object-oriented languages, it is easy to modularly define new *variants* of existing classes, by means of subclassing. It is however much harder to define new *operations* in a modular way, if an operation has to be supported by several classes. In that case, the new operation has to be implemented by every variant of a class that needs to support the operation, thus scattering the implementation of the operation over existing components. To alleviate this problem, design patterns such as the *Visitor* pattern can be used [58]. When using the Visitor pattern, it is however still necessary to add

[1]e.g., if this changes the interface of a class that is part of an existing library. In addition, consider the result if delegation is applied to a subclass that overrides methods in its superclass. If a method signature is changed in a subclass, it no longer overrides the same method in its superclass.

callback methods to each visited class.

As another example, in languages that support an inheritance mechanism similar to Smalltalk [60] or Java, subclasses can override methods defined in their superclass, and decide whether the original behavior (in the superclass) is invoked. For this reason, it is impossible to restrict subclasses from completely redefining existing behavior as implemented in the superclass. This makes it very easy to define subclasses that (accidentally) break properties that where guaranteed by their superclass[2]. Users of these classes will however assume such properties to hold, potentially leading to incorrect behavior[3].

A different style of inheritance mechanism is supported by the programming language Beta [101]. In this language, superclasses have control over the execution of methods that have been refined by subclasses. Thus, a superclass can more easily guarantee certain properties (e.g., invariants), as it has control over the invocation of any sub-behavior. On the other hand, this approach makes it impossible to completely replace ("override") existing behavior by means of subclassing, thus severely limiting the potential directions in which a class can evolve. In addition, the desired extension points must be predicted correctly.

Thus, each composition operator (and hence, language) has a bias that makes some types of evolution scenario's easier to accommodate, or less error-prone, than others. Such trade-offs are inherent to the choice of particular composition operators – there exists no single composition operator that is able to address all kinds of evolution scenario's equally well, while still providing meaningful higher-level abstractions.

To work towards addressing the issues identified above, we present a composition infrastructure that (a) supports the definition of a range of composition mechanisms, (b) allows composition mechanisms to be expressed in terms of first-class entities, enabling the construction of new composition mechanisms from existing ones, (c) supports the use of multiple composition mechanisms within the same program, while (d) supporting a variety of aspect- as

[2]A common occurrence in Java programs is when classes override the method *equals(..)* as defined in *java.lang.Object*. Due to the contracts of the methods *equals(..)* and *hashCode()*, redefining one almost always necessitates redefining the other. This is often neglected.

[3]In the above example, instances of classes that override *equals(..)* but not *hashCode()* or vice versa, may lead to incorrect behavior when such instances are added to collections, such as sets or hash tables [7].

well as object-based composition mechanisms.

To this aim, we generalize the (aspect-specific) model of software composition offered by the Java Aspect Metamodel Interpreter (JAMI) as presented in chapter 4, to a model that supports the expression of both object- and aspect-oriented composition mechanisms. The model is based on the underlying primitive mechanism of implicit invocation, as already supported by JAMI.

Our approach has been implemented and presented in terms of a small language, called “*Co-op*”. In this language, composition operators can be constructed using several “primitive” elements, such as selectors, bindings, actions and constraints, which can be used to define composition operators based on implicit invocation. These primitive elements are expressed in terms of first-class elements (objects), so that they can be freely composed. We use these primitive elements to express several composition mechanisms, including different styles of inheritance, e.g. as found in Smalltalk [60] or Beta [101], as well as aspects.

This way, programmers (or language designers) can experiment with new language operators or tailor the existing ones without going through the tedious task of modifying the parser, compiler and interpreter.

To this aim, *Co-op* is designed to be a simple language to explain and implement; it uses a small set of mechanisms that facilitate the creation of new composition operators, as well as the expression of meaningful examples that use these operators.

The next section discusses the design of *Co-op*, and demonstrates its use by showing implementations of diverse inheritance mechanisms, an example using a general-purpose aspect-oriented composition mechanism, as well as a domain-specific aspect-based mechanism that implements the observer-pattern as a first-class abstraction. This is followed by a discussion of several properties of the language, related work, and an evaluation/conclusion. A comprehensive description of the denotational semantics of *Co-op* is included in appendix A.

5.2 The Co-op Language

In this section we introduce the language Co-op, which we will use to construct several composition mechanisms.

5.2.1 Language definition

As our main focus is to define and experiment with composition mechanisms based on a common set of lower-level implementation concepts, we prefer to start with a language that does not include a fixed set of composition mechanisms of its own.

Also, we want this language to be based on objects, i.e., supporting data abstraction and encapsulation "out of the box".

Listing 5.1 shows a simple example that demonstrates the structure of the language.

```
 1 module Point {
 2   var @x, @y;
 3
 4   init(x, y) { @x = x; @y = y; }
 5   setX(x) { @x = x; }
 6   getX() { return @x; }
 7   ...
 8 }
 9
10 module Main {
11   main() {
12     var p;
13     p = [Point new: "10", "20"];
14     [p setX: "30"];
15   }
16 }
```

Listing 5.1: Example *Co-op* program

The example defines a module `Point`, with instance variables `x` and `y`. Instance variables are prepended by the symbol `@`, to avoid confusion with parameters or local variables. On line 4-6 in listing 5.1, three operations that can be invoked on a `Point` object are defined. Operation implementations consist of (optional) local variable declarations and zero or more statements. Co-op supports three types of statements: variable assignments, return statements, and "event generation" statements. Line 4 and 5 each show assignment statements, line 6 a return statement, while line 13 and 14 each show an "event generation" statement. Line 14 specifies the creation of an event with intended target object `p`, message selector `setX` and a parameter with value `30`. At this stage, we intentionally speak about "generating an event" as opposed to "calling a method" or "sending a message", since it is not yet known which part(s) of the program may take an interest in this event, or in other words, which operation implementation(s), if any, will be eventually invoked (executed) as a result of this event.

The language includes a default *dispatch mechanism* that connects an event to an operation invocation. For each event that occurs, this mechanism tries to execute the operation implementation corresponding to the message selector specified by the event, in the above case, `setX`. This invocation only succeeds if the *lookup type* of the event, which is initially set to the type of the target object, actually defines such an operation.

Note that this mechanism is expressed in terms of implicit invocation: *for each* generated event, *invoke* the operation with the selector specified in the event, within the indicated target class. Before the operation is invoked, a pseudo-variable "this" is in addition bound to the target object, so that the operation implementation can refer to a "this" variable. We will later define other composition mechanisms, which may use constraints to obtain precedence over this default dispatch mechanism. Thus, the language does not have a *fixed* dispatch mechanism (it can be changed), but does include a *default* mechanism.

In the above example, the target object indicated by the event is `p`, an instance of module type `Point`. Thus, the event has the same effect as a direct method call (i.e., a call that does not involve polymorphism or other complex dispatch mechanisms) would have in a regular object-based language.

Finally, line 13 contains an assignment statement of which the right-hand side is also an "event generation" expression. The selector `new` is implemented as a primitive operation, which indicates that a new object should be constructed, of the type specified as the target of the event, in this case `Point`. When such an event is handled by the default mechanism, a new object is created and returned as the value of this expression. The module type of the new object is also specified (as the target of the `new` operator), so that the dispatch mechanism can later connect events pertaining to this object to the correct module implementation. In addition, if defined, the operation `init` of the specified type is invoked with the newly created object as the target object.

When an instance variable is referenced in an operation implementation, the value is looked up based on the current "this" object and the name of the variable. Instance variables may only be referenced within the module in which they are defined, to ensure encapsulation; from outside that module, they can of course be accessed through accessor methods. However, even though instance variables cannot be (directly) referenced by other modules, it is still possible to create composed objects that "contain" instance variables from multiple modules: since instance variable lookup is based on the value of "this", and composition mechanisms can influence that value, it is possible to invoke operations in different modules with the same “this”-context, if a composition mechanism so desires.

The above may appear contradictory: if instance variables are encapsulated, but a “this”-context can still be shared between modules, does this not break the encapsulation? Consider the following analogy: given two (Java) classes; class X, defining a private instance variable x, and class Y, which extends X, and defines a private instance variable y. Now, for each instance of class Y, the instance variables x and y are contained within the same “this”-context (object). Nonetheless, class Y cannot obtain the value of x other than through invoking an accessor defined in class X.

Since *Co-op* allows only the defining module to refer to its own instance variables, all instance variables are effectively private. Thus, the situation is analogous to the above example. We will later show examples that have to deal with the sharing of instance variables between modules, e.g., in the context of multiple inheritance.

As a main goal of our language is experimentation with composition mechanisms, we restricted ourselves to including the minimal features that make the language usable for that purpose. The current version therefore supports only two “basic” object types: strings and booleans. To be able to write meaningful examples while keeping the language definition itself relatively small, we also added support for closures, which can be used to define control structures in very much the same way as provided by Smalltalk. In addition, built-in module definitions are provided for lists and dictionaries (hash tables). Using the above, we can implement common control structures as shown in listing 5.2.

```
1 [aCondition ifTrue: { [Console writeln: "condition is true"]; } ];
2 [{aCondition} whileTrue: { ...statements to be repeated... } ];
3 [aList forEach: {|item| [Console writeln: "The list contains item: ",
      item]; } ];
```

Listing 5.2: Using control structures

Line 1 triggers an event with target object `aCondition`, message selector `ifTrue` and a closure (the part between curly braces) as a parameter. The target object should be a boolean, and the implementation of the `ifTrue` operation will execute the closure depending on whether `aCondition` is the object representing `true` or `false`. Similarly, line 2 creates an event with message selector `whileTrue`, sending it to a closure object, which will repeatedly evaluate the closure and as long as it returns `true`, executes the second closure, specified as an argument of the event. Line 3, finally, creates a `forEach` event and sends it to a list object. The closure, again specified as a parameter of the event, takes an input parameter `item` which can be used within the closure; thus allowing the program to iterate over collections. When executed, closures set their return value to the result of the last evaluated statement within the closure. A closure can be executed by sending an `execute` event to it.

The above approach allows us to define a language with a small syntax, while still having full expressive power. Below, we show the abstract syntax of the language. Note that in this abstract syntax, the composition primitives (such as bindings and selectors) are modeled explicitly. In the concrete grammar, these primitives are modeled as regular objects (e.g., closures), and can thus be freely assigned to (instance) variables, passed as parameters, etc.

$$\begin{array}{rcl}
Program & \triangleq & Module* \\
Module & \triangleq & id : String; body : ModuleBody \\
ModuleBody & \triangleq & instanceVars : Variable\_list; \\
 & & operations : BehaviorDecl\_list \\
BehaviorDecl\_list & \triangleq & BehaviorDecl* \\
BehaviorDecl & \triangleq & id : String; \\
 & & formalParameters : Variable\_list; \\
 & & body : BehaviorBody \\
BehaviorBody & \triangleq & localVars : Variable\_list; \\
 & & statements : Statement\_list \\
Statement\_list & \triangleq & Statement* \\
Statement & \triangleq & ReturnExpression|Assignment|EventExpression \\
ReturnExpression & \triangleq & returnValue : Expression \\
Assignment & \triangleq & target : Variable; value : Expression \\
Expression & \triangleq & EventExpression|Variable|Literal|ClosureDecl \\
EventExpression & \triangleq & intendedTarget : Expression; \\
 & & selector : OperationSelector; \\
 & & actual\_arguments : Expression\_list \\
OperationSelector & \triangleq & name : String; annotation : String \\
Expression\_list & \triangleq & Expression*; \\
Variable\_List & \triangleq & Variable*
\end{array}$$

$$
\begin{array}{rcl}
Variable & \triangleq & id : String \\
Literal & \triangleq & String|Boolean \\
Type & \triangleq & String \\
ClosureDecl & \triangleq & formalParameters : Variable\_list; \\
 & & body : BehaviorBody \\
NewStatement & \triangleq & var : Variable; \\
 & & type : Type; \\
 & & init\_params : Expression\_list \\
EventSelector & \triangleq & eventToBool : ClosureDeclaration \\
ActionSelector & \triangleq & eventToActionRef : ClosureDeclaration \\
Binding & \triangleq & es : EventSelector; as : ActionSelector \\
Constraint & \triangleq & PreConstraint|CondConstraint|SkipConstraint \\
PreConstraint & \triangleq & b1 : Binding; b2 : Binding \\
CondConstraint & \triangleq & b1 : Binding; b2 : Binding \\
SkipConstraint & \triangleq & b1 : Binding; b2 : Binding \\
Event & \triangleq & sender : Object; local\_type : Type; \\
 & & target : Object; lookup\_type : Type; \\
 & & msg\_sel : String; actual\_args : Object\_list \\
Object & \triangleq & identity : N; type : Type; \\
Object\_list & \triangleq & Object*
\end{array}
$$

Appendix A contains a detailed definition of the denotational semantics of the above grammar. In this chapter, we informally describe the semantics based on multiple examples. It is almost unavoidable that such informal descriptions are not always exhaustive; in such cases, please refer to the appendix for a detailed description.

5.2.2 Base composition mechanism

The core composition mechanism of *Co-op* is based on the following elements:

- *Events*, which may be published (generated) during the execution of an operation,

- *Event Selectors*, queries that match published events based on their properties, as well as (other) reflective information about the program that can be reached through the context of an event,
- *Action Selectors*, which select an operation to be invoked, as well as its intended target,
- *Bindings*, which bind event selectors to action selectors, and in addition specify the binding of values between an event (the "calling side") and the invoked behavior (i.e., sharing of values between contexts),
- *Constraints*, which can be used to impose constraints on the execution in case multiple bindings match the same event. Three predefined constraint-types are provided: skip-constraints, conditional-constraints and pre-constraints. These fulfill a role that is analogous to the constraints defined in the context of JAMI in chapter 4: if multiple composition mechanisms are interested in the same event, constraints can be imposed between these composition mechanisms.

These elements form the core of our approach, allowing the expression of various object- and aspect-oriented composition mechanisms, as well as the composition of composition mechanisms. The next section demonstrates this in detail.

We briefly discuss how composition mechanisms can be constructed from the above elements.

First, events have the following properties: a sender object (the "this"-context at the moment the event is generated), an intended target object (specified as the first argument of the event), a message selector (cf. method name), a lookup type (the module to which the message may be dispatched, initially set to the type of the target object), local type (the module in which the event originated, set to the type of the "this"-context at the moment the event is generated), and parameters (cf. call parameters).

Whenever an event is generated within the program (by means of an `EventExpression`), it is matched against all event selectors registered with the system.

Event selectors may match particular events based on their properties, as well as other reflective information available through the event (e.g., values of instance variables of the target object). Action selectors return an action to be taken; they return a reference containing several – potentially modified –

event properties, in particular the lookup type and message selector. Bindings specify which event selector is connected to which action selector, and may optionally define additional context variables, such as “this”, which by default is set to the target of the current event and may thus be omitted.

The language includes one “default”-binding, with the following behavior: it matches any event, while its action part invokes (executes) the operation implementation indicated by the *lookup type* and *message selector* specified by the event. In addition, the binding defines a context variable “this”, of which the value is the *target* of the event. Such “context variables” are available while evaluating the selectors, as well as within the invoked operation. This way, operations can create events targeting the ‘this”-object. Note that even this “default”-binding can be effectively overridden by constraints, such that the mechanism is not fixed (i.e., the default behavior can be overridden).

In the next sections, we will use the concepts defined below to define several composition mechanisms.

5.3 Constructing Composition Operators Using Co-op

In this section, we show implementations of several object-based composition mechanisms, such as single inheritance, multiple inheritance, and `Beta`-style inheritance [101]. In addition, we construct a (domain-specific) aspect-based composition mechanism, realizing the observer pattern.

5.3.1 Building a composition mechanism: single inheritance

Co-op allows the definition of composition mechanisms using the concepts discussed in the previous section. These concepts are represented as first-class entities within the program, i.e., they can be used as parameters, returned as values of operations, assigned to (instance) variables etc., like any other object. Thus, composition operators are not built into the language, but can be expressed and composed using primitives supported by the language.

Listing 5.3 shows the implementation of a basic single-inheritance mechanism. We define the intended behavior of inheritance as follows: given an inheritance relation where module `X` extends module `Y`, events of which the lookup type is `X` should in principle lead to execution of the indicated operation within module `X`. This corresponds to the behavior implemented by the default call binding, which does exactly that. However, if (and only if) module `X` does not define an operation with the message selector as indicated by the event, the event is instead re-evaluated while its *lookup type* is changed to `Y`, thus “forwarding” the message to the “parent” type.

To implement this in terms of the concepts defined in the previous section, when the *lookup type* of an event matches the `child` type of an inheritance relation, we also invoke the same operation indicated by the event on its `parent` class. This means that two different “bindings” match the event: the default binding, which matches all events, as well as the just described “inheritance binding”. Since this would lead to the execution of two operations when both the `parent` and `child` type implement an operation with the specified message selector, we add a constraint between the default binding and the inheritance binding. The constraint specifies that if the default binding (i.e., the `child` type) can successfully invoke the operation (which is the case only if the `child` type defines an operation with the name specified as the message selector of the event), we want to skip the invocation of the binding to the `parent` type. This way, operation definitions in the `child` type effectively override those in the `parent` type, while operations not defined in the `child` type are forwarded to the `parent` type, as intended by the inheritance mechanism.

```
module Extends {
 var @inheritanceBinding;

 init(child, parent) {
   var overrideConstraint;
   var callToChild, sendToParent;

   callToChild = [Selector new:
     { [[event lookupType] isEqual: child]; }
   ];
   sendToParent = [Selector new:
     { [OperationRef new: parent, [event selector]]; }
   ];
   @inheritanceBinding =
     [Binding new: callToChild, sendToParent]];

   overrideConstraint = [SkipConstraint new: defaultCallBinding,
       @inheritanceBinding];

   [overrideConstraint activate];
   [@inheritanceBinding activate];
 }

 getInheritanceBinding()
 {
   return @inheritanceBinding;
 }
```

```
27 }
```

Listing 5.3: Single inheritance implementation

Listing 5.3 shows the implementation of the inheritance mechanism as described above. For each inheritance relation between two classes an instance of this module is created, specifying a `parent` and `child` type. Within listing 5.3, lines 8–10 define an event selector, defined as a closure. When evaluated, it will match any events of which the *lookup type* is the same as the specified `child` type. The event itself is available as an (implicit) context variable of every selector. Lines 11–13 define an action selector that specifies the operation characterized by module type `parent`, and operation name equal to the original event message selector. When this action selector is invoked, the event will be re-evaluated with properties *lookup type* and *message selector* as specified by this operation reference. The binding defined on lines 14+15 specifies that each "call" to the `child` type should thus also be sent to the `parent` type.

Note that "event-dispatch" is thus potentially a multi-stage process: if the parent module is itself involved in another `Extends` relation, the re-evaluated message may get redirected again, until it is at some point dispatched by the default binding, or discarded if no "up-stream" composition mechanisms are interested in it. We designed the mechanism this way to allow the layered application of the same composition mechanism (e.g., multiple levels of inheritance), or potentially even of different composition mechanisms, without the need to specify the complete message dispatch mechanism as a single (and thus not very composable) function.

Without any explicit constraints, the default binding (which matches any event) and the inheritance binding specified here, will both match events of which the lookup type equals the specified `child` module, leading to the execution of two operations. Since it is the intention of this inheritance mechanism to only "forward" calls not supported by the `child` type to the `parent`, line 17 specifies a constraint that skips the action defined by the inheritance binding if and only if the default call binding matches the same event, and manages to successfully invoke its specified action. A reference to the default binding is for convenience made globally available as variable `defaultCallBinding`, such that constraints can also be applied to this binding.

Finally, lines 19 and 20 register the inheritance binding and constraint with the implementation of the system; once the "activate" operation is invoked on a selector or binding, it is internally registered to the list of active composition elements, which means all events occurring from that moment onwards will be matched against that binding. It is also possible to deactivate existing bindings or constraints, effectively removing them from the system. This way, it is possible to dynamically deploy and even change existing composition relations, although this is not further explored in this chapter.

Listing 5.4 introduces an example that uses the inheritance mechanism defined above.

```
module Widget {
 var @visible;

 display() { [[this isVisible] ifTrue: { [this draw]; }]; }
 show() { @visible = true; }
 hide() { @visible = false; }
 isVisible() { return @visible; }
 getDimensions() { /* ..return on-screen dimensions.. */ }
}

module Window {
 var @windowTitle;

 setTitle(title) {@windowTitle = title; }
 draw() { /* Draw window */ }
}

module DialogWindow {
 var @windowText;

 setText(text) { @windowText = text; }
 draw() { /* Draw dialog window */ }
}

module Main {
 main() {
  var window;

  [Extends new: "DialogWindow", "Window"];
  [Extends new: "Window", "Widget"];

  window = [DialogWindow new];
  [window setTitle: "Confirmation requested"];
  [window setText: "Are you sure?"];
  [window display];
 }
}
```

Listing 5.4: An example using single inheritance

In this listing, we define three modules representing user interface elements: `Widget`, `Window` and `DialogWindow`. A fourth module defines inheritance relationships between them, and shows how the resulting composition can be used. Specifically, line 29 creates an instance of the `Extends` module defined in listing 5.3, and specifies that `DialogWindow` inherits from `Window`. Similarly, line 30 specifies that `Window` inherits from `Widget`. As the constructor (init operation) of the module `Extends` immediately activates the bindings and constraints it defines, from this point on, calls to objects of type `DialogWindow` will be forwarded to `Window` in the case that `DialogWindow` does not implement an operation itself, and similarly, events that cannot be handled by `Window` itself will be forwarded to `Widget`. So, at this point, there are 3 active bindings in the program: the default binding, the inheritance binding between `Widget` and `Window`, as well as the inheritance binding between `Window` and `DialogWindow`.

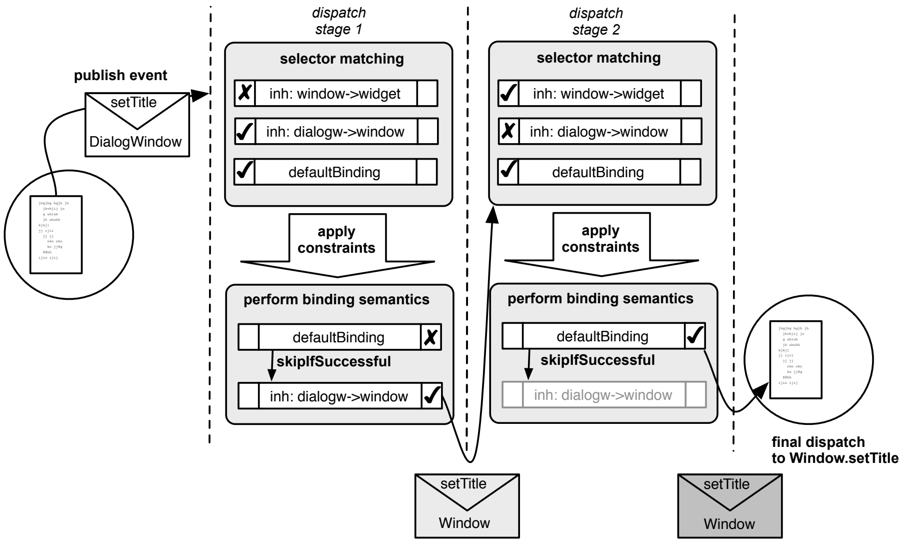


Figure 5.1: Example of message dispatch in Co-op; single inheritance

Using figure 5.1, we explain the control flow for each of the three events generated in lines 33–35.

Line 33 generates an event with target object `window`, message selector `setTitle` and a string literal parameter. Figure 5.1 depicts the message flow initiated by this event. First, when the event is "published", it is matched against all active selectors. In this case, there are 3 active selectors: the default binding (which is always active), as well as the two inheritance bindings, one for each inheritance relation defined in this example. Two of the three bindings match the published event: the default binding, which matches all events, and the inheritance binding between `DialogWindow` (child) and `Window` (parent). The latter binding matches, since its event selector specifies the following matching criteria: `[[event lookupType] isEqual: child]`. The *lookup type* of each event is initially set to the type of the target object, which in this case was instantiated as a `DialogWindow` on line 32. In principle, the actions specified by both matching bindings would be invoked. However, the `Extends` relation has also defined a `Skip`-constraint (line 17 in listing 5.3), specifying that the action indicated by the inheritance binding will only be executed if either the default binding does not match at all, or if it fails to execute its operation. In this case, the default binding *does* fail to execute its operation, as module `DialogWindow` does not define an operation named `setTitle`. Thus, the second binding is evaluated, which modifies the *lookup type* to the parent specified by the binding: in this case, module `Window`. The message is then re-evaluated (this corresponds to the right-hand side of the figure). Again, it matches two bindings: this time the *lookup type* matches the binding between `Window` (child) and `Widget` (parent). The default binding also matches. This time, the default binding, which looks for operation implementations in the type referred to by the *lookup type*, can successfully execute its action part, since module `Window` does define an operation `setTitle`. For this reason, the execution of the other binding is skipped.

Line 34 shows a similar example. In this case, the default binding can immediately execute the operation `setText`, as it is implemented by module `DialogWindow`, which is the initial *lookup type* of the event. Thus, since the default binding successfully executed, the inheritance binding to module `Window` is skipped.

When executing line 35, the same sequence of events happens as on line 33; except this time, the module `Window` does not contain an operation `display` either. Therefore, the action indicated by the inheritance binding between `Window` and `Widget` message gets executed, which re-evaluates the message with the *lookup type* set to Widget. Now, the default binding matches the message, as `Widget` does contain an operation `display`. Thus, the handling is essentially the same as in figure 5.1, but with an extra “dispatch stage” in the middle.

The implementation of `Widget.display` contains a call to object `this`. The variable `this` still points to the same object, i.e., the value of variable `window` (since `this` is by default set to the target object of the event). Since this object is of type DialogWindow, any calls to this will correctly be dispatched through the same inheritance structure.

Finally, since we are constructing composition mechanisms within the language itself, there has to be a "lowest level" binding, defined as the "default binding" in the previous section, that deals with the actual call dispatch. This binding is implemented as a primitive operation, i.e. its internal events are not seen by any custom-defined selectors. The implementation of each composition mechanism, which may itself generate events, must eventually reduce to this "lowest"-level binding. Thus, the definition of a composition mechanism should not generate events that will be matched by the implementation of that mechanism itself, or the implementation of a mechanism on which it depends. Otherwise, the process of event matching itself generates new events, which recursively get processed by event matching, ad infinitum.

In the example above, calls to the "event"-object, as well as creation of the built-in module types Selector, Binding, OperationRef and SkipConstraint only use the low-level default-binding, i.e. these modules only use the low-level object-based (without inheritance) composition mechanism.

5.3.2 Extending a composition mechanism: multiple inheritance

The representation of composition mechanisms as first-class entities enables us to reuse definitions of existing mechanisms. In this section, we extend the inheritance mechanism defined in the previous section to a multiple-inheritance mechanism. Listing 5.5 shows the complete definition of this mechanism. This particular implementation allows each child module to have several parent modules; these parents are specified as an ordered list, and our intention is that if the child module does not specify an implementation of a particular operation itself, the lookup mechanism searches for operation implementations within those parent modules *in the specified order*.

```
module MultiExtends {
 init(child, listOfParents) {
  var prevBinding;

  [listOfParents foreach:
  { |parent|
    var binding;
    binding = [[Extends new: child, parent] getInheritanceBinding];

    [[prevBinding isDefined] ifTrue: {
      [[SkipConstraint new: prevBinding, binding] activate];
     }];
13      prevBinding = binding;
14    }];
15  }
16 }
```

Listing 5.5: Multiple inheritance as an extension of single inheritance

Listing 5.5 implements the following behavior: for each `parent` in the list of parents (line 5), an `Extends` is created between the specified `child` and `parent` module (line 8). Thus, in case the `child` module does not define an particular operation implementation itself, each event targeted at an object of type `child` is sent to *all* its parent types. Since this “redirection” of events to a parent type is exactly the behavior implemented by module `Extends`, we reuse this existing composition mechanism. This results in an implementation of multiple inheritance that is based on a *composition* of several instantiations of the existing `Extends` mechanism. Using just the expressions discussed above (i.e., corresponding to lines 5–8 of listing 5.5), the implementation of multiple inheritance is already almost correct, *except* in cases where multiple parent modules define an operation with the same name. In that case, rather than invoking all those operations, we invoke only the operation defined by the parent with highest priority. For this example, we made the design decision to define this priority as “the left-most module in the list of parents that supports the requested operation”. To implement this restriction, we create an additional constraint between each inheritance binding and the one “to the left” of it (except for the first parent), specifying that if the binding on the left matches, the one on the right should be skipped. Note that it is possible to create such additional constraints between existing inheritance bindings, since the definition of module `Extends` allows us to access the inheritance binding it defines; alert readers may have noticed the accessor implementation in listing 5.3 (lines 23–26).

Lines 10–13 define the above behavior: during the first iteration, the value of `prevBinding` is still undefined (line 10), but on all following iterations, a constraint is created between the current and the previous binding, specifying that the binding on the right should be skipped if the one on the left matches the same event and can invoke its operation successfully (line 11).

Listing 5.6 shows an example using the multiple inheritance mechanism as defined above. Lines 1–4 define an additional module, which adds printing functionality. Assume we want to add this functionality to module `Dialog-Window` by using inheritance. As before, `Window` extends `Widget` (line 10), while `DialogWindow` now extends both modules `Window` and `Printable` (line

11).

```
 1 module Printable {
 2  print() { /* print the contents of the window */ }
 3  getDimensions() { /* ..return on-paper dimensions.. */ }
 4 }
 5
 6 module Main {
 7  main() {
 8    var window;
 9
10    [Extends new: "Window", "Widget"];
11    [MultiExtends new: "DialogWindow", [List new: "Window", "
          Printable"]];
12
13    window = [DialogWindow new];
14    [window display];
15    [window print];
16    [window getDimensions];
17  }
18 }
```

Listing 5.6: Example using multiple inheritance

Note that this implementation of multiple inheritance can also deal with so-called “diamond-shaped” inheritance structures. Suppose, for example, that in addition to the structure defined above, class `Printable` extends class `Widget`. Thus, we have the following structure: class `DialogWindow` extends both the classes `Window` and `Printable`, while both of these classes inherit from class `Widget`.

In that case, the above implementation of multiple inheritance will always try to dispatch calls through the leftmost inherited module (and its superclasses) first, and only if this fails, dispatch through the next modules, as specified during the construction of the inheritance relation; see listing 5.6, line 11. In the above example, operations will always first be dispatched through module `Window`, and only if this fails, through module `Printable`.

Since state is associated based on the value of the “this” object, there is no problem with regard to state duplication, as might occur in C++ (where, unless the programmer takes care to explicitly specify “virtual inheritance”, 2 duplicates of each instance variable declared in class `Widget` would exist in class `DialogWindow`). Rather, since the value of “this” is independent of the module or operation implementation to which an event is dispatched, the value of any instance variables will be the same, regardless of the followed inheritance path (e.g., through class `Window` or class `Printable`).

5.3.3 Calls to "super": dispatch deviation

Most object-oriented languages provide special keywords to refer to related modules in the program structure. For example, in Java the keyword `super` can be used by subclasses to call method implementations defined in an ancestor class, even if the subclass overrides the original method implementation (defined in one of the ancestor classes). Such keywords are necessary, as regular "this"-calls would be sent to the method implementation in the subclass itself, leading to an infinite sequence of recursive calls.

However, the keyword `super` does not refer to an object in the same sense as `this` can be considered a normal object. Although the value of `this` depends on the current execution context (unlike normal variables, hence the often used designation "pseudo-variable"), to a programmer it appears to behave like any other variable. Keywords such as `super`, on the other hand, are rather an indication to the message dispatch mechanism of a language that it should deviate from its normal procedure: instead, the lookup procedure followed to dispatch the message should be started at the of the class in which the keyword occurs.

In other words, the goal of a call to `super` is to replace the regular message invocation to a "this"-objects with a different type of invocation. Rather than introducing a keyword (such as `super`), we model the desired behavior using "event annotations". Effectively, we model a call to "super" as a normal "this"-call, while adding an annotation to the event indicating that the composition mechanisms should handle it differently from normal calls. Listing 5.7 shows a code fragment, revisiting the implementation of module `DialogWindow`:

```
1 module DialogWindow {
2   ...
3   draw() {
4    [this draw@super]; /* draw the window itself */
5    /* draw additional elements of the dialog window */
6   }
7 }
```

Listing 5.7: Example of a "super"-call

In the context of this example, it makes sense that the implementation of `DialogWindow.draw` would invoke the original implementation in `Window.draw`, since it only adds additional graphical elements. Line 4 above specifies such an event, i.e. the target object is `this`, the message selector is `draw`, and the event is annotated with the designation `super`. Listing 5.8 shows how a composition mechanism in Co-op can use such annotations to implement alternative message dispatch strategies.

```
 1 module SingleInheritance
 2 {
 3   var @inheritanceTable;
 4
 5   init()
 6   {
 7     @inheritanceTable = [Dictionary new];
 8   }
 9
10   extends(child, parent) {
11     var inheritanceBinding, superBinding;
12     var callToSuper, sendToSuper;
13
14     [@inheritanceTable put: child, parent];
15
16     inheritanceBinding = [[Extends new: child, parent]
           getInheritanceBinding];
17
18     callToSuper = [Selector new:
19       { [[[event lookupType] isEqual: child] and: [[event
             callAnnotation] isEqual: "super"]]; }
20     ];
21
22     sendToSuper = [Selector new:
23       { [OperationRef new: [@inheritanceTable get: [event localType
             ]], [event selector]]; }
24     ];
25
26     superBinding = [Binding new: callsToChildSuper, sendToSuper];
27
28     [[SkipConstraint new: superBinding, defaultCallBinding] activate
           ];
29   }
30 }
```

Listing 5.8: Dealing with “super” calls

First of all, to resolve “super”-calls, the composition mechanism specified in listing 5.8 has to keep track of inheritance relations between modules. Unlike previous versions, this mechanism therefore stores each inheritance relation as it is added (line 14). Then, it reuses the existing module `Extends` to instantiate the inheritance behavior. In addition, line 18 defines an event selector that matches if an event is directed at the specified child type and the event has the annotation `super`.

The action selector defined on line 22–24 is different from what we have seen so far: rather than returning a reference to a fixed “parent” module, it looks up the parent (in the inheritance table) of what is currently the “local type”. This event property is set to the module in which the event originated.

Note that this variable can very well have a different value than the *lookup type*, which is initially set to the type of the target. In this case, both happen to have the same value (`DialogWindow`), but if a "super"-call were initiated within module `Window`, the *local* type would be `Window` whereas the *lookup* type (i.e., the type of the `this` object) would still be `DialogWindow`. Hence, the distinct event properties are necessary; without a reference to the module context in which the program is currently executing, it would be impossible to reference related program entities relative to this module.

Finally, line 28 specifies that if the binding to "super" matches and can be invoked successfully, the default binding should be skipped.

5.3.4 Beta inheritance

To demonstrate the flexibility in defining composition mechanisms provided by Co-op, we show the implementation of a rather different style of inheritance mechanism, as found in the language Beta [101]. Beta turns the inheritance structure "upside down" as compared to Java or Smalltalk: calls are always sent to the root class of an object, and only if the root class does not define an operation implementation, the dispatch mechanism navigates down the inheritance tree towards the "actual" type of the object. However, this would in principle prevent the refinement (by child classes) of existing operations (in parent classes). Therefore, Beta provides the keyword `inner`, which allows parent classes to explicitly invoke refinements of an operation defined in a child-class.

Listing 5.9 shows revisited fragments of the previously defined Widget modules. In this example, all calls to an object of type `DialogWindow` are initially sent to module `Widget`, as it is the root class in this inheritance structure. If module `Widget` does not implement the requested operation, it is redirected to module `Window`, then `DialogWindow` respectively. Analogous to the implementation of "super" discussed previously, this is again handled by annotating the invocation, in this case with the annotation `@inner`. Line 6 shows an example of a call to `inner`: after executing its own behavior (e.g., clearing a buffer for drawing), the `draw`-implementation of module `Widget` invokes the operation `draw` on the next module down in the inheritance structure towards the actual type of the target object, if it exists (in this case, module `Window`). Similarly, the `draw`-implementation of module `Window` again invokes its `inner`-implementation, if one exists (in this case it does, within `Dialogwindow`).

A benefit of this type of inheritance is that existing behavior is guaranteed to be present in (future) extensions; it is possible to extend modules only in a way that is controlled by the parent class. In contrast, when using a Java- or Smalltalk-like inheritance mechanism, it is impossible to prevent child-classes from completely overriding existing behavior. Depending on the situation, either mechanism may be more suitable, and Co-op allows the use of both, even within the same program[4].

```
module Widget {
  ...
  display() { [[this isVisible] ifTrue: { [this draw]; }]; }
  draw() {
   /* Clear drawing buffer, then call: */
   [this draw@inner];
  }
}

module Window {
 ...
  draw() {
  /* Draw basic window elements, then call: */
  [this draw@inner];
  }
}

module DialogWindow {
  ...
  draw() { /* Draw dialog window specific elements */ }
}

module Main {
  main() {
    var composition, window;

    composition = [BetaInheritance new];
    [composition extends: "DialogWindow, "Window"];
    [composition extends: "Window", "Widget"];

    window = [DialogWindow new];
    ...
    [window display];
  }
}
```

Listing 5.9: Example using Beta inheritance

[4]However, a single module must not be made part of both types of inheritance hierarchies, as the respective composition mechanisms resolve message dispatch in inherently incompatible ways.

Listing 5.10 shows our implementation of the inheritance mechanism discussed above. Similar to the inheritance mechanism supporting "super"-calls, the mechanism defines two bindings: one to implement inheritance (lines 17–24), the other to implement calls to `inner` (lines 26–45). The definition of the inheritance binding is very similar to the ones previously discussed; the most important difference is that the "skip"-constraint defined on line 24 specifies the opposite order as in the definitions for normal inheritance (compare listing 5.3, line 17). Because of this, events will be redirected all the way to the root module of a particular object, and are only executed by modules "downwards" in the hierarchy if none of the parent modules defines the requested operation.

Line 26 matches events that have the annotation `inner` attached, and which originate within a module that is registered as a `parent` in an inheritance relation. The action selector (line 29–44) determines where to redirect the message. It figures out the intended target module by starting at the type of the target object (line 33), then repeatedly navigating upwards in the inheritance structure (line 36–40) until it encounters the `parent` module. The intended target module is then the one found just before navigating to the parent itself[5].

Finally, the action selector returns the necessary event properties, modified according to the above description (line 42), annotating this call with the annotation `innerimpl` to prevent the normal inheritance binding from matching and thus sending the message back up the inheritance tree, as it would do with regular calls.

```
 1 module BetaInheritance {
 2   var @inheritanceTable;
 3 
 4   init()
 5   {
 6     @inheritanceTable = [Dictionary new];
 7   }
 8 
 9   extends(child, parent) {
10     var overrideConstraint;
11     var inheritanceBinding, innerBinding;
12     var callToChild, sendToParent;
13     var callToInner, sendToInner;
14 
```

[5]The lookup mechanism has to be implemented like this, as every parent class can obviously have multiple extensions (subclasses). Without navigating upwards through the module hierarchy, starting at the type of the target object, it would not be possible to determine to which of several child modules the mechanism should navigate.

```
  [@inheritanceTable put: child, parent];

  callToChild = [Selector new:
   { [[[event lookupType] isEqual: child] and: [[event callAnnotation] isEqual: ""]];}
  ];
  sendToParent = [Selector new:
   { [OperationRef new: parent, [event selector], [event callAnnotation]]; }
  ];
  inheritanceBinding = [Binding new: callsToChild, sendToParent];
  [[SkipConstraint new: inheritanceBinding, defaultCallBinding] activate];

  callToInner = [Selector new:
   { [[[event localType] isEqual: parent] and: [[event callAnnotation] isEqual: "inner"]]; }
  ];
  sendToInner = [Selector new:
   {
    var childType;
    var prevType;
    childType = [[event target] type];
    prevType = childType;

    [ { return [[childType isEqual: parent] not]; } whileTrue:
    {
     prevType = childType;
     childType = [@inheritanceTable get: childType];
    }];

    [OperationRef new: prevType, [event selector], "innerimpl"];
   }
  ];
  innerBinding = [Binding new: callsToInnerChild, sendToInnerChild];

  [[SkipConstraint new: innerBinding, inheritanceBinding] activate];
  [[SkipConstraint new: innerBinding, defaultCallBinding] activate];

  [inheritanceBinding activate];
  [innerBinding activate];
 }
}
```

Listing 5.10: Definition of Beta inheritance

5.3.5 Aspect-based composition

To demonstrate that Co-op can accommodate general-purpose aspect-based compositions, this section revisits the example from the introductory chapter (chapter 1), implementing an electronic hotel booking system with access control.

First, listing 5.11 shows the definition of a general-purpose pointcut-advice mechanism, that supports *before* and *after* advice (similar to AspectJ). An aspect is initialized by specifying a pointcut, which can match events based on the properties available through the `event` context-variable. In addition, the initializer specifies an advice operation to be invoked, and the desired "advice type" (*before* or *after* advice, indicated by a string value). The implementation of module `Aspect` binds the pointcut to the advice (lines 6,7), and defines a constraint that orders the execution of the aspect relative to the default binding, executing the aspect either before or after the operation invoked by the default binding (lines 9–18).

```
 1 module Aspect {
 2   init(pointcut, advice, advType) {
 3     var binding;
 4     var constraint;
 5
 6     binding =
 7      [Binding new: [Selector new: pointcut], [Selector new: advice],
           { return [[Dictionary new] put: "this", [event target]]; }
          ];
 8
 9     [[advType isEqual: "before"] ifTrue:
10     {
11       constraint = [PreConstraint new: binding, defaultCallBinding];
12       [constraint activate];
13     } ];
14     [[advType isEqual: "after"] ifTrue:
15     {
16       constraint = [PreConstraint new: defaultCallBinding, binding];
17       [constraint activate];
18     } ];
19
20     [binding activate];
21   }
22 }
```

Listing 5.11: A general-purpose pointcut-advice mechanism for Co-op

Using this general-purpose pointcut-advice mechanism, we can modularly express the crosscutting access control concern that was defined as part of the electronic hotel booking system. In listing 5.12, first a module `Booking` is defined (lines 1–11). This module only implements functionality related to (adding or canceling) bookings. Then, lines 13–31 define a module named `AccessControl`. This module implements aspect-based behavior based on the pointcut-advice mechanism defined in listing 5.11: it defines a pointcut that matches all calls to module `Booking`, and instructs the aspect to invoke operation `checkAccess` in module `AccessControl` *before* executing the actual operation invoked on module `Booking`. The operation `checkAccess` first checks whether the user has access, and prints a warning message if this is not the case[6].

```
 1 module Booking {
 2   addBooking() {
 3     [Console writeln: "Adding a booking"];
 4     // insert bussiness logic here..
 5   }
 6
 7   cancelBooking() {
 8     [Console writeln: "Canceling a booking"];
 9     // insert bussiness logic here..
10   }
11 }
12
13 module AccessControl {
14   var @user;
15
16   init(currentUser) {
17   @user = currentUser;
18     [Aspect new:
19       { [[event lookupType] isEqual: "Booking"]; },
20       { [OperationRef new: "AccessControl", "checkAccess"]; },
21       "before"
22     ];
23   }
24
25   checkAccess() {
26   [[user hasAccess: "Booking"] ifFalse: {
27       [Console writeln: "Access control: Action not permitted to this
            user"];
28       // "throw exception"; this is not implemented.
29     } ];
30   }
31 }
32
33 module Main {
34   main() {
35     var bookings;
```

[6]Since Co-op does not currently support exceptions, we can not create an implementation that is completely analogous to the example in chapter 1; this implementation does not actually throw an exception.

```
    var ok;

    bookings = [Booking new];
    [AccessControl new];

    ok = [bookings addBooking];
  }
}
```

Listing 5.12: An access control aspect expressed in Co-op

5.3.6 Domain-specific composition: the Observer pattern

To show that Co-op is also suitable for the implementation domain-specific composition operators, we demonstrate a composition operator that provides a generic, reusable implementation of the “Observer” pattern.

Listing 5.13 shows an example of its usage. Line 3 creates an instance of an `Observe` module, which takes 3 parameters: the object to be observed, a list of operations in which the observer takes an interest, and the behavior (specified as a closure) to be invoked after any of the listed operations is invoked on the observed object. In this example, the observer prints a message after each invocation of method `show` or `hide` on object `window`, stating whether the window is currently visible or not. (Since the behavior is defined as a closure, it can access all variables visible within the scope where the closure is declared. Therefore, the implementation of the `Observer` module does not need to explicitly refer to the specific context of each observer instance. Thus, this implementation of the observer pattern is a generic and reusable solution.)

```
1 window = [DialogWindow new];
2 ...
3 [Observe new: window, [List new: "show", "hide"], {
4  [Console writeln: "The window is now ",
5    [[window isVisible] ifTrueElse: { "visible" }, { "invisible" }]];
6 }];
7
8 [window show]; // observer prints: "The window is now visible"
9 [window hide]; // observer prints: "The window is now invisible"
```

Listing 5.13: Observer pattern example

Listing 5.14 shows the implementation of the observer composition mechanism itself. Line 10 defines the event selector: the target of the event must equal the intended subject of observation, while the event message selector must be in the list of operations in which the observer is interested. Then, the operation `notify` is invoked within module `Observe` (line 11), which in turn will execute the closure specified as an argument of this observer instance.

Finally, line 16 shows a new type of constraint: a `Pre`-constraint indicates that if both bindings match the same event, the one on the left hand side should always be evaluated before the one on the right. Thus, such `pre`-constraints define a partial ordering between bindings. Note that unlike what happens when using a `skip`-constraint, both bindings are still executed. In this example, we specify that the action invoked by the default binding (i.e., the call to the observed object) should be executed before the action specified by the observer binding (i.e., the action invoking operation `notify`). By simply exchanging the order, notification would be executed *before* rather than *after* the observed call. In AspectJ terminology, one could say that the notification "advice" would then behave as a `before`-advice rather than an `after`-advice.

```
 1  module Observe {
 2   var @observerBehavior;
 3
 4   init(subject, listOfOperations, observerBehavior) {
 5     var callToSubject, sendToObserver;
 6     var binding, constraint;
 7
 8     @observerBehavior = observerBehavior;
 9
10     callsToSubject = [Selector new: { [[[event target] isEqual:
           subject] and: [listOfOperations contains: [event selector]];
            } ];
11     sendToObserver = [Selector new: { [OperationRef new: "Observe",
           "notify"]; } ];
12
13     binding = [Binding new: callsToSubject, sendToObserver, { [[
           Dictionary new] put: "observer", this]; } ];
14
15    [[PreConstraint new: defaultCallBinding, binding] activate];
16     [binding activate];
17   }
18
19   notify() {
20     [[observer getObserverBehavior] execute];
21   }
22
23   getObserverBehavior()
24   {
25     return @observerBehavior;
26   }
27  }
```

Listing 5.14: Generic implementation of the observer pattern

5.4 Implementation of Co-op

The Co-op language as well as the examples discussed in this chapter, are implemented and available for download [9]. The current prototype is implemented on top of the Java Aspect Metamodel Interpreter (JAMI), which was presented in chapter 4. The language is implemented using the same workflow as described in detail for several domain specific aspect languages, see section4.3.4.

First, an Antlr-based parser converts Co-op programs to an object-based AST, which is then converted into JAMI elements. Since JAMI already supports the notion of event selectors, bindings to (advice) actions and constraints between bindings, the main implementation work is to implement the different kinds of "actions" or statements supported by Co-op. For each one (assignments, return statements, and event generation), a language-specific JAMI advice class is implemented, dealing with that type of action. Operation implementations (i.e., methods) are then constructed out of these basic actions. In addition, the system adds some internal actions to the operation implementations, for example, to keep track of the current execution context and variable scopes (e.g., creating and removing stack frames). Event generation statements can lead to the invocation of other operations (advice actions) through the selector-advice binding system already present in JAMI.

Closures are implemented by creating a shallow copy of the execution context (stack) at the point where a closure is created. Thus, closures can refer to all variables available in their "parent" scope, even after that scope itself has been closed. Several of the examples in this chapter use this functionality.

The operation implementations of some built-in modules are implemented natively (as part of the system), for example those representing selectors, bindings and constraints. This is necessary, as these modules need to interface with the system's implementation in JAMI.

5.5 Discussion

5.5.1 Composition and mixed use of composition operators

Co-op enables "composition" at different levels, which we distinguish here for the sake of clarity:

- Composition operators can be defined within a program, allowing the implementation of domain-specific composition mechanisms. Our implementation of the *Observer* pattern (section 5.3.6) demonstrates this.

- Since the concepts used to define composition operators are modeled as first-class entities (objects) within the program, composition operators can also themselves be composed of (or, can reuse parts of) other composition operators. Our implementation of multiple inheritance mechanism as an extension of a previously defined single inheritance mechanism demonstrates this.

- In section 5.2.2, we defined elements that support the construction of composition mechanisms based on implicit invocation, queries on generated events, binding of state and behavior, as well as constraints between parts of (potentially different) composition mechanisms. This allows the expression of various object-oriented as well as aspect-oriented composition mechanisms.

- Multiple kinds of composition operators can be used (mixed) in the same program. For example, the *observer* mechanism was demonstrated in an example that also involves an *inheritance* mechanism. We note that our approach cannot *automatically* guarantee the correctness of such compositions.

It is the responsibility of the programmer that designs and implements a new composition mechanism to ensure that it works correctly if combined with other composition mechanisms — if such use is intended.

Co-op does however *facilitate* the implementation of constraints between multiple composition mechanisms, by supporting declarative constraint specifications. For example, the definition of an aspect oriented composition mechanism in section 5.3.5 specifies a *partial ordering constraint* between the execution of aspect-related behavior and the default binding. In other words, a `PreConstraint` ensures that the action specified by one binding must be executed *at some point* before the action specified by the other, but not necessarily *immediately* before. If other composition mechanisms (e.g., inheritance) match the same event, in addition to the aspect-oriented mechanism, it may be the case that both composition mechanisms specify constraints in relation to the default binding. However, unless explicit constraints are added directly between these two composition mechanisms, the "precedence" between these composition mechanism is undefined, if they both match the same event.

When allowing the combined use of multiple composition mechanisms, the necessary constraints between composition mechanisms depend on domain knowledge about the defined composition specifications. For this reason, such constraints cannot in general be derived automatically. As an example, should inheritance be resolved before evaluating aspects, or vice versa? This needs to be decided by the designers of the respective composition mechanisms.

In some cases, composition mechanisms implement inherently conflicting notions of composition, and can therefore never be composed in a meaningful way. For example, when adding a single module to both a Beta-inheritance hierarchy as well as a Smalltalk-like inheritance hierarchy, the results can never be correct, since both mechanisms have an inherently incompatible notion of inheritance (unless adopting the approach in [61]). Still, as long as each module occurs in at most one of the hierarchies, even these composition mechanisms can both be used in the same program.

5.5.2 Reasoning about correctness, performance and optimizations

Since our approach is currently based on an interpreter and dynamic typing, it is hard to implement any kind of static reasoning on the resulting compositions. Since it was our intention to first allow the definition of composition mechanisms with maximum flexibility, we have designed the language in a way that does not impose many constraints. For example, currently the only fixed part of an event selector, is the fact that they take an event as input, and return a boolean result as output, i.e. they behave as any function. Since functions (operations, closures) are already supported by the language, we did not have to add any special language syntax to describe the selectors and other parts of the base composition mechanism.

It would however be possible to define a more restricted selector-language, to which static reasoning or partial evaluation could be applied more easily. In addition, partial evaluation of selectors could reduce the amount of event generation statements that can potentially match each selector, thus improving both the possibilities for optimization and reasoning about the program.

Performance has deliberately been excluded as a consideration in the design of the language and composition mechanism. The prototype implementation will exhibit slow performance for larger examples. One of the main reasons is that every event is to be considered at least once by all selectors, which all need to be evaluated for each event. We do believe that the performance for such a language can be substantially improved, by applying many known techniques for optimization and efficient implementation of dynamic languages [38], as well as efficient models for implementing aspect-based mechanisms [67]. Suggestions already mentioned above are partial evaluation, as well as adding restrictions to the freedom that the current interpreted version allows (e.g., by defining a more structured selector language).

5.6 Related work

The work in this chapter is related to a large body of research on defining new languages that support novel composition techniques, especially in the domain of object-based and aspect languages. Many papers also present a (small) set of composition techniques that aim at unifying existing composition techniques. However, *most* of such related research proposes a *fixed set of composition operators*, presented as part of a language, extension of a language, or an application framework. In contrast, our work focuses on a language that has no—or alternatively a single—built-in composition operators, but rather is a platform for constructing a wide range of user-defined composition operators.

To the best of our knowledge, there are no other languages that offer *dedicated* support for user-defined composition operators (that can be reused and combined), at least not within the domain of object-oriented and aspect-oriented languages. Please note that this excludes languages that offer generic extension mechanisms – such as macro's in Lisp – or allow for the extension and modification of the program through metaprogramming – as we will discuss below –.

Of the research that aims at unifying composition techniques, we first discuss a few that relate particularly to the example composition operators we have shown in this chapter:

- In [29], mixin inheritance is presented as a generalization of both regular ('Smalltalk-style') inheritance and Beta-style inheritance, as well as CLOS-style mixins. But the mixin mechanism itself is a fixed composition operator, and cannot be used to define new composition operators. The definition of a Co-op composition operator that implements mixin inheritance is part of our future work; it is our belief that this will not pose substantial technical problems.

- We have demonstrated in this chapter that we can define and use multiple composition operators, including the use of respectively Smalltalk/-Java style *super* and Beta-style *inner*, in parallel. In [61], it has been shown how a *specific* method dispatching technique, implemented in the language *MzScheme*, allows the usage of both inheritance styles simultaneously.

- Compositional Modularity [20] is an inheritance model that supports a wide range of compositions on modules (which correspond to self-referential namespaces in this model). This is achieved by modeling the compositions as a set of operations on the modules. Compared to our approach, there are limitations to its expressiveness, due to the fixed set of primitive operations. The compositional modularity model has been applied to a variety of 'base' artifacts.

- *Classpects*[113] unify aspect- and object-oriented programming. The language *Eos-U* implements the *classpect* construct, which can be considered as a combination of aspect-behavior and object behavior in a single abstraction mechanism. Eos-U offers the concept of bindings, which have roughly the same structure as the bindings in Co-op: binding advice to join points. However, there is no mechanism for expressing ordering constraints beyond declaration order. Regardless of these similarities, Eos-U is distinct from *Co-op* by offering only a fixed set of composition operators and abstractions.

- Composition filters (or interface predicates) in the Sina language [13, 21] define a single language mechanism that can be used –among other things– to express various data abstraction mechanisms such as different forms of inheritance (single, multiple, conditional), delegation, and aspects. *filter modules* are abstractions of several filter expressions, but not an independent composition operator. The introduction of new filter types can be used to add additional composition behavior to a system, but all within the same framework of composition filters, not as new, independent composition operators.

There are other approaches that allow for the construction of user-defined composition operators. In particular, our work relates to metaprogramming [39] and especially meta-object protocols[85]. Depending on the programming language/environment, metaprogramming offers the programmer the full power to modify the behavior of programs. This includes the ability to write custom compositions. As explained e.g. in [84], the power of metaprogramming comes with more complexity and responsibility. In particular, it may be extremely hard to define multiple application-specific compositions in such a way that work together without interference (i.e. such that they are composable).

Meta-object protocols (MOPs) aim at addressing this by providing a framework –albeit at the metalevel– with more structure and constraints, so that e.g. composition operators can be defined within a well-defined structure. This means that the difficulty of language design –except for the concrete syntax– is now on the MOP designer. Indeed, our work might just as well have been presented as a MOP, but for practical reasons we chose to use a concrete language, *Co-op*. We are not aware of any MOPs (or languages, or frameworks) that offer similar generic abstractions and structure as we presented in this chapter. In particular, we do not know any MOPs that provide abstractions for defining new composition operators with similar variety, expressiveness and composability. For example, *Co-op* explicitly supports a variety of object-oriented as well as aspect-oriented composition mechanisms. In *Co-op*, composition mechanisms are constructed using first-class, composable elements, which can be reused to define or compose new composition mechanisms. In addition, the resulting composition operators are also first-class entities, which means they can be reused and extended as well.

It may well be possible to implement our approach as a metaobject protocol on top of, e.g., CLOS. However, the core contribution of our approach is not the Co-op language itself, but rather the model of composition that is enabled by the elements presented in section 5.2.2, e.g., selectors, bindings, constraints. These elements provide explicit support for the expression of composition mechanisms that are based on the notion of *implicit invocation*, such as aspects. In addition, our approach supports the expression of explicit, declarative constraint specifications to address dependencies between composition mechanisms. In a CLOS-based implementation, we would still have to provide all the novel abstractions and infrastructure that we have presented in this chapter.

Of the research that aims at providing frameworks higher-level languages through or meta object protocols, we briefly discuss the following:

- AspectS [76] is framework for supporting aspect-programming in Smalltalk. It extends the Smalltalk MOP with features that enable aspect programming. As such, it does not extend the language itself. For instance, it uses Smalltalk itself as the pointcut language, similar to our use of the 'base' language for defining selectors.

- MetaClassTalk [28] aims at 'unified aspect-oriented programming'. It exploits a combination of mixin-based inheritance and reflection to achieve this. Its aspects consist of (a) a set of mixins, (b) a pre-weaving script, (c) a post-weaving script. In this approach, every programmer is a meta-programmer, with a lot of control—and responsibility to write correct meta-programs. MetaClassTalk also involves 'weave-time' code; which is a disadvantage from the point of view of abstraction, but does have potential benefits with respect to performance optimization and static reasoning.

Finally, we mention several frameworks that aim at offering a generic platform for OO and AOP language implementations (e.g. [67], [121], and [50]). For such platforms, the designers have also made efforts to find a small set of generic constructs that typically serve as a target 'language' for a compiler/-code transformation. An important distinction with our work is that these platforms do not aim at, and hence do not support, the ability of creating user-defined composition operators.

5.7 Evaluation and Conclusion

In this chapter, we presented the Co-op language. The contribution of Co-op is that it enables the creation –and usage– of first-class composition operators for expressing a wide variety of composition techniques. Examples that we have demonstrated in this chapter are single inheritance, multiple inheritance, beta-style inheritance, super- and inner-calls, pointcut-advice composition, and a domain specific composition for observations. In addition, we have defined a detailed denotational semantics of the *Co-op* language, which is included in appendix A

An important feature is the composability of the composition operators: since composition operators are represented as first-class entities, their definitions can be reused to compose new composition operators. This is demonstrated in section 5.3.2, where a multiple inheritance mechanism is defined as an extension of a previously defined single inheritance mechanism. In addition, multiple types of composition can be used in the same program, as long as they do not specify inherently conflicting ways to handle composition. Further, we have shown how language keywords such as `super` (in Java) and `inner` (in Beta) can be represented as event annotations, i.e., as a property indicating that the dispatch mechanism should handle the call differently from regular method invocations.

We believe the notion of user-defined, first-class composition operators, brings us closer to the following goal, as expressed by Guy Steele in his "growing a language" talk [64]:

> "a language design can no longer be a thing. It must be a pattern – a pattern for growth - a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal."

In this case the pattern is a means to grow (by composition) user-defined composition operators, which express particular patterns of interaction among modules.

5.7.1 Design Considerations and Lessons learned

To achieve a better understanding how Co-op achieves the features we just presented, we will now discuss the key elements in the design of the Co-op model, and discuss how they contribute to the capabilities of Co-op. Although these design elements cannot be seen in isolation, we believe that the discussion below is partially generalizable to other composition techniques. In fact, many observations have been derived from the experiences in the design of object- and aspect-oriented languages.

- *implicit invocations*: to be able to offer a generalized mechanism for both object-based and aspect-based compositions, the core design of the language needed to fully decouple message sends (i.e. event generation) from an eventual method execution. For this reason we adopt the notion of implicit invocations [109]; there exists a dynamic, one-to-many relationship between a message send and the possible operation executions or metalevel actions.

- use of *queries* (the 'selectors' in our language) for selecting events and actions: the advantage of specifying a query instead of a fixed, 'hard-coded' identifier to refer to program elements, is that a query allows for much more conceptual/semantic relations (see e.g. [107]) , rather than accidental and inflexible identifier-based connections. A consequence of using a query is that it may yield zero, exactly one, or many results, and this has to be taken into consideration.

- use of *reflective information* (event reflection, program introspection and state information—through object interfaces) within queries: this is also one of the lessons from aspect languages; the use of context information—through a generic interface— is a powerful means to pass on context information between modules without making the modules dependent on each other (i.e. improving the composability). This context information may be available only at run-time.

- concept of *bindings* for (a) associating queries with actions, and (b) associating data variables in different contexts: the first is a common technique in AOP languages (see e.g. Eos-U, as discussed in section 5.6).

- use of *constraints* for composing bindings: this turned out to be a crucial issue in achieving composability; the ability to express constraints, including dynamic constraints, at a fine-grained, per-binding level. A topic of future work is whether it is important to be able to apply constraints to particular groups of bindings, which may even be selected by queries

- using *metalevel actions* to manipulate events: composing systems through meta-level manipulation of messages between the system components has been proven before to be a successful technique [22, 53]. It has two attractive properties: it allows to reason about the system without breaking the encapsulation of the individual components, and it is relatively easy to offer a generic, shared abstraction of messages, which avoids application-specific dependencies.

- composition operators, bindings, queries (selectors) and constraints as *first-class* citizens: this is a crucial property for composition operators; it allows the definition and reuse of composition operators, a key contribution of Co-op. For bindings, queries and constraints, this is —at least currently— less vital, but at times very convenient for the language designer, as it avoids the necessity of dedicated language constructs, e.g. for passing around references.

- *multi-stage dispatch*: this is an important feature, because it helps to realize transitive composition relations, such as exemplified by inheritance in section 5.3.1, where the method lookup for a single message send may involve multiple *extends* composition operators. Multi-stage dispatch also promotes the composability of composition operators, as it allows for the application of multiple composition operators for a single event.

- *dispatch to multiple operations*: this means that a single event may yield multiple actions and even multiple operation executions. This allows for expressing before/after advices in an AOP style, and also less common examples of multiple dispatch, such as multi-cast semantics. Note that both multi-stage and multiple dispatch require proper ordering constraints.

5.7.2 Future Work

There are still many possible issues to explore, we list a few of them:

- The language itself can be refined, taking e.g. ease-of-use into account.

- We would like to investigate the possibilities to make Co-op a statically typed language, and understand the consequences of such a change.

- Currently, static analysis of Co-op programs is hard; to investigate this, we may introduce some slight adoptions to the language; by making the creation of selectors, bindings and constraints, as well as their activation, more explicit, it is easier to analyze the actual structure. In addition, we may investigate replacing the current, full Turing-complete selector language with another selector language that is easier to analyze statically.

- Once we can do better static analysis, also a lot of potential for performance optimizations will appear.

- One issue that needs to be further investigated, is to what extent it is necessary to specify constraints among (bindings of) different composition operators, as this bears the theoretical risk that for every new composition operator, all possible interactions with existing composition operators needs to be investigated and specified. However, to date, we have not experienced such a need!

- There are still many composition techniques that we would like to experiment with, and demonstrate that they can be expressed using Co-op. Examples include: delegation [97], mixins [29], composition filters [21], and so forth. Along these lines, we are also interested to adopt a framework as proposed in [110], which outlines a number of core operations from which a wide range of OO compositions can be constructed: it would be interesting to be able to express the core operations as separate composition operators, and use those to compose new, higher-level,composition operators.

Conclusion

In this chapter, we highlight the contributions of the research presented in this thesis. In addition, we evaluate the results of the work presented in each chapter, by discussing the engineering trade-offs that are involved.

6.1 Resolving introductions and detecting ambiguous compositions

Chapter 2 studies issues caused by aspect-oriented composition mechanisms that allow aspects to change the structure of a program, called *introductions*. In particular, we focus on the use of *annotations* in combination with aspect-oriented programming. We present several types of scenario's where annotations can reduce the fragility of pointcuts in evolving applications, and improve the reusability of aspects. The use of annotations by aspects also entails the *introduction* of annotations based on pointcuts. This is discussed in section 2.2.

We discuss the introduction of annotations in the context of Compose*, a language- and compiler platform for aspect oriented language research. Compose* supports an expressive pointcut language, which we use to specify where annotations should be superimposed. However, by supporting pointcuts that express potentially complex predicates over the structure of a program, the introduction of annotations based on such pointcuts can itself be influenced by the introduction of other annotations (e.g., as specified by other aspects). In other words, dependencies among such introductions superimposition specifications may occur.

To properly apply introductions (e.g., of annotations) in the presence of such dependencies, these dependencies must be resolved. In some cases however, the dependencies cannot be resolved in an unambiguous manner, implying that the program as specified can be interpreted (e.g., by a compiler) in multiple ways, leading to different compiled programs with different behavior. This is typically highly undesirable. We demonstrate, in section 2.5, that these issues occur in several aspect-oriented languages, including AspectJ (the most mainstream AOP language).

We present the following solution to this problem: we propose an algorithm (in section 2.6) that resolves the introduction of annotations, if this can be resolved unambiguously. If the specified introductions cannot be unambiguously resolved, the algorithm statically detects this, identifying the introductions that cause the ambiguity.

To validate our approach, we implemented and tested the introduction annotations within the Compose* platform [69]. In addition, the algorithm that resolves dependencies between introductions and detects ambiguous combinations of introductions, is successfully implemented within Compose*.

Thus, the results of this work are integrated with the Compose* implementation, which is available for download [10]. The research presented in this chapter was published as a paper at the 5th International Conference on Aspect-oriented Software Development, AOSD 2006 [71].

6.2 A Graph-based Approach to Modeling and Detecting Composition Conflicts

The main contribution of chapter 3 is that it proposes a general approach to the precise analysis of aspect composition conflicts, specifically those related to the introduction of program elements based on pointcuts.

We employ a graph-based model to represent aspect-oriented programs, and represent introductions as graph transformations on these program representations. This enables the expression of conflict detection rules in terms of graph matching patterns. These graph-based representations have been introduced to deliver a precise explanation why and when specific compositions cause a conflict. In addition, this precise modeling allows us to perform conflict detection in a fully automated way, leveraging existing tools for the analysis of graph-based models.

We present a classification of composition conflicts related to introductions as caused by either: violation of language rules, unintended effects, or ambiguous aspect specifications (section 3.2), and discuss the options that aspect language developers have to address the conflicts in each category (section 3.5).

We validate our approach by showing that the graph-based approach, described above, is suitable for the automatic detection of composition conflicts. We have implemented a prototype that performs automatic conflict detection for each of the three conflict categories mentioned above. The prototype can handle Java/AspectJ source code, automatically converting relevant parts of the program structure to a graph-based representation. In addition, it automatically converts introductions (called "inter-type declarations" in AspectJ) to specifications that represent graph transformations. We use the Groove ("Graphs for Object-Oriented Verification") tool set to automatically detect and reveal conflicts in the graph-based representation of a program. Groove is an existing tool set, developed for the purpose of model-based verification of object- and aspect-oriented programs. We describe the graph-based representations in section 3.3. The automated detection of composition conflicts, based on those representations, is discussed in section 3.4.

The research presented in this chapter was published as a paper at the 6th International Conference on Aspect-oriented Software Development, AOSD 2007 [73].

6.3 Prototyping and Composing Aspect Languages

Chapter 4 introduces a meta-model of aspect-oriented programming languages. We use this meta-model to express diverse domain-specific aspect languages, thus illustrating that the model can accommodate diverse aspect-oriented programming languages with different sets of features, based on a common set of aspect language abstractions.

The meta-model is implemented as an interpreter framework, called JAMI, the Java Aspect Metamodel Interpreter. This framework can be used to prototype and test aspect oriented languages with actual (Java) programs in an interpreted style.

The meta-model framework enforces two important design concerns, while otherwise allowing a large degree of freedom and flexibility to model various aspect languages: the high-level structure and high-level control flow of aspects are imposed by the framework.

As a result of the common high-level structure, aspects expressed in concrete aspect oriented languages can be mapped to this common structure in a straightforward manner. The framework includes several refinements of each structural element, modeling common aspect-oriented language constructs. It requires a very reasonable amount of effort to prototype aspect oriented languages based on our framework. We demonstrate this by prototyping and testing several aspect oriented languages in relatively few lines of code and in a reasonable amount of time, in section 4.4.

In addition, the enforcement of a common high-level control flow allows the composition of aspects written in several aspect oriented languages, in one program. The framework explicitly addresses composition at shared join points, by means of declarative constraint specifications.

The main contribution of chapter 4 is a novel meta-model interpreter framework, described in detail in section 4.3, which supports:

- Prototyping of (domain-specific) aspect oriented languages. Apart from research- and educational purposes, these prototypes can be used to test and validate the behavior of (experimental) aspect languages using concrete programs. The definition of several language prototypes can be found in section 4.4.

- Composition of programs that use aspects expressed in several (domain-specific) aspect oriented languages, based on a constraint model that can express complex constraints between aspects, even if these aspects have been written in several aspect languages. An example demonstrating a complex composition of multiple aspects, written in several aspect languages, is given in section 4.5.

The research presented in this chapter was published as a paper at the 22nd European Conference on Object-Oriented Programming, ECOOP 2008 [72].

6.4 Constructing Composable Composition Mechanisms

Chapter 5 presents a composition infrastructure that (a) supports the definition of a variety of composition mechanisms, (b) allows composition mechanisms to be expressed in terms of first-class entities, enabling the construction of new composition mechanisms from existing ones, (c) supports the use of multiple composition mechanisms within the same program, while (d) supporting a variety of aspect- as well as object-based composition mechanisms.

We generalize the (aspect-specific) model of software composition offered by the Java Aspect Metamodel Interpreter (JAMI) as presented in chapter 4, to a model that supports the expression of both object- and aspect-oriented composition mechanisms. The model is based on the underlying composition mechanism of implicit invocation, as already supported by JAMI.

Our approach has been implemented and presented in terms of a language, called "*Co-op*", which we use to express the elements necessary to define composition mechanisms based on implicit invocation. We use these elements to express several composition mechanisms, including different styles of inheritance, e.g. as found in Smalltalk [60] or Beta [101], as well as aspects, and even *domain-specific* compositions. The design of these composition mechanisms is discussed in detail, in section 5.2.

An important contribution is the composability of the composition operators: since composition operators are represented as first-class entities, their definitions can be reused to compose new composition operators. This is demonstrated in section 5.3.2, where a multiple inheritance mechanism is defined as an extension of a previously defined single inheritance mechanism. In addition, multiple types of composition can be used in the same program, as long as they do not specify inherently conflicting ways to handle composition. Further, we have shown how language keywords such as `super` (in Java) and `inner` (in Beta) can be represented as "event annotations", i.e., as a property indicating that the dispatch mechanism should handle the call differently from regular method invocations.

Since Co-op is implemented on top of JAMI, programs and composition operators expressed in terms of Co-op can be tested in practice. The interpreter, as well as the examples described in this chapter, are available for download [9].

In addition to the implementation, we have also formally defined the denotational semantics of *Co-op*, a description of which is included in appendix A.

6.5 Evaluation of the work presented in this thesis

The design of programming languages, frameworks, or composition mechanisms involves making trade-offs between specific software engineering qualities. Within this thesis, many such trade-offs have been mentioned, e.g., to motivate the design or specific uses of various composition mechanisms.

In this section, we explicitly list a number of these trade-offs, and evaluate how relevant software engineering qualities may be affected by the work presented in this thesis.

Based on the ISO standard for Software Product Quality [78], the work in this thesis may primarily affect two of the six main quality characteristics defined by that standard: maintainability and efficiency[1]. *Maintainability* is defined as "the capability of the software to be modified". The standard defines several sub-characteristics of maintainability, three of which are most relevant for the discussion in this section: *analyzability* ("the capability of the software product to be analyzed for deficiencies or causes of failure in the software, or for the parts to be modified to be identified"), *changeability* ("the capability of the software product to enable a specified modification to be implemented"), and *stability* ("the capability of the software product to minimize unexpected effects from modifications of the software"). *Efficiency* can be divided into space- and time complexity, but in the context of the work in this thesis also in *compile time* and *run time* performance.

We discuss the influence of the work presented in this thesis on these characteristics from two perspectives. First, the work may affect quality characteristics as seen from the perspective of *software engineers*, i.e., *end users* of the composition mechanisms, languages and analysis tools presented in this thesis. At another level, our work may affect quality characteristics as seen from the perspective of *language engineers*, for whom our work may for example facilitate the prototyping or analysis of (domain-specific) aspect languages and/or composition mechanisms.

Since the quality characteristics introduced above are still rather broad, we define several more specific sub-characteristics, which should be understood in the context of evaluating these quality characteristics from the perspective of software- or language engineers, as indicated above.

In this context, a sub-characteristic of *analyzability* is the *comprehensibility* of the source code: the ability of a programmer to understand the intended functionality of the code. Comprehensibility can again be divided in two sub-characteristics: the comprehensibility of modules in isolation (without extensive knowledge of the entire system), or the comprehensibility of a system in its entirety, without extensive knowledge of all its modules (i.e., whether it is easy to comprehend the relations or connections between parts of the system, and how different parts of the system may affect each others behavior). We will refer to these terms as "local" and "global" comprehensibility,

[1]The other characteristics defined by the standard are: functionality, reliability, usability and portability. We focus specifically on those characteristics that are primarily affected by those engineering characteristics of a system that are the subject of this thesis.

respectively. Related to comprehensibility, *conciseness* is the ability to express intended functionality in a brief but comprehensive way.

More specific characteristics that define *changeability* are *reusability*, the ability to reuse or refine previously defined functionality, and *flexibility*, the ability to create new compositions out of new and/or existing functionality.

The specific characteristics discussed above can also be applied to the different perspectives: from the perspective of a language engineer, our work may for example provide the *flexibility* to define various aspect language mechanisms in a relatively *concise* manner. At the level of a *user* of such mechanisms (i.e., a "regular" software engineer), the creation of domain-specific aspect languages may enable such a user to write more *concise* programs. Tables 6.1 and 6.2 show the meaning of the characteristics discussed above, from these distinct perspectives.

Quality Characteristic	**Software Engineer's perspective**
Global comprehensibility	Ease of understanding a software system in its entirety
Local comprehensibility	Ease of understanding specific modules of a software system
Conciseness	Brevity of programs (while remaining comprehensive)
Reusability	Ability to reuse/refine elements (e.g., modules) of a program
Flexibility	Ability to compose elements (e.g., modules) of a program
Efficiency (compile time)	Space/time complexity of program compilation
Efficiency (run time)	Space/time complexity of program execution

Table 6.1: Software qualities: Software engineer's perspective

Quality Characteristic	**Language Engineer's perspective**
Global comprehensibility	Ease of understanding a language specification in its entirety
Local comprehensibility	Ease of understanding specific language constructs or composition mechanisms
Conciseness	Brevity of language definitions (while remaining comprehensive)
Reusability	Ability to reuse/refine definitions of language constructs or composition mechanisms
Flexibility	Ability to compose languages, language constructs or composition mechanisms
Efficiency (run time)	Space/time complexity of language constructs or composition mechanisms

Table 6.2: Software qualities: Language engineer's perspective

Below, we discuss the characteristics defined above in the context of the work in each chapter.

Chapter 2 discusses the use of annotations and introductions, and the combination of these two is used to exemplify the interactions that may occur between aspects[2]. From the perspective of *software engineers*, the (combined) use of annotations and introductions may improve *reusability* of aspects, since aspects can refer to any program elements (i.e., potentially in several programs) that have a certain annotation attached, rather than referring directly to specific parts or elements of a (single) program. In addition, programs may become more *comprehensible* (locally) as a result, since annotations can be used to convey the *design intention* of a programmer.

The use of crosscutting specifications to introduce annotations can be used to designate the role of classes in a particular application. A class may have different roles in different applications, and by separating this information from the class itself, its *reusability* is increased.

The derivation of annotations based on program properties that may include other annotations, allows for greater *flexibility* in creating structural compositions, as exemplified in section 2.2.4, item 3.

The use of annotations and introductions may have negative effects as well. First and foremost, annotations and introductions may improve reusability and flexibility, but comes at the cost of decreased *conciseness*, caused by the necessary extra specifications (e.g., as compared to writing pointcuts that directly refer to specific program elements). Software engineers should weigh this trade-off, and only use such constructs when the improved reusability and/or flexibility is likely to pay off; this may be especially the case when defining product lines or when foreseen evolution scenarios may later be accommodated more easily because of the increased flexibility.

Another potential negative effect is that, although the *local comprehensibility* of modules may increase, the *global comprehensibility* of a system in its entirety may suffer from the use of introductions, since they add additional *dependencies* between modules in the system, and these dependencies are not explicitly visible from the "client" module on which the dependency is imposed (as would normally be the case when a dependency is "imported" to a module that needs it). This may make it harder to comprehend how all the parts of a system are connected or related.

[2]i.e., the occurrence of such interactions is not limited to just the context of annotations and introductions, as discussed in section 2.8.2.

Our approach, as presented in chapter 2, attempts to mitigate this effect, by ensuring that combinations of introductions behave in a predictable, declarative way. This increases the *analyzability* of the system in two respects. First, in case of ambiguous specifications, our detection mechanism indicates the problem area. In addition, because the specifications of introductions are declarative, the system can be more easily understood, e.g., by tracing the origins of referenced annotations without taking any ordering rules into account (and hence, without the need to analyze the entire system to establish the effects of such ordering rules).

Runtime *efficiency* is not typically affected by the use of introductions or annotations[3], since annotations are pure meta-data that is commonly used at compile-time only. Similarly, introductions are typically resolved at compile-time, as well.

The compile-time *efficiency* of the algorithm used to resolve dependencies between introductions, as well as to detect potential ambiguities, has an exponential worst-case space- and time complexity, based on the size of its input. However, its input consists of "introductions specifications", of which there are expected to be much fewer than lines of code (orders of magnitude less), even in large applications.

In addition, such a "worst case" scenario would occur only when all "introduction specifications" in a given program affect each others results - something that would only result from very convoluted program designs. In reality, we would never expect (and have indeed never seen) more than 2–4 layers of dependencies between introductions, and for such numbers even our exponential algorithm does not cause any issues; hence we did not investigate further improvements within the scope of this work.

Chapter 3 generalizes the results of chapter 2 to the (static) detection of several types of issues caused by incorrect or potentially unintended software compositions, including ambiguous interpretations of composition specifications. From the perspective of software engineers, this improves the *analyzability* of the software. Since patterns that may indicate unintended effects of compositions are also detected, *stability* may be improved as well, since unintended compositions may be noticed at an earlier stage, thus reducing the potential for unexpected behavior of the software. From the point of view

[3]Exception: when annotations are made available at run time, e.g., for reflective purposes, they may have a – typically minor – performance impact; this is however unrelated to any of the work presented in this thesis.

of language engineers, the graph-based approach enables the straightforward definition of domain- or application-specific detection patterns, which we believe is a more *flexible* and *(locally) comprehensible* approach as compared to, e.g., defining specialized algorithms to analyze the structure (AST) of a program for each potential kind of problem pattern, as also discussed in section 3.6.

Whereas chapters 2 and 3 primarily focused on improving the *analyzability* of existing language constructs, chapters 4 and 5 focus on improving *changeability*, at two levels. First, at the level of software engineers, we propose a framework for aspect-based compositions that supports the (combined) use of several aspect languages, including domain-specific aspect languages. The use of such domain-specific languages typically increases *conciseness*, since the language is tailored to a specific domain. In addition, domain-specific specifications are generally more *comprehensible* (at least locally), as they can be expressed directly in terms of concepts from a specific domain, rather than in terms of general-purpose instructions.

Language engineers can use our aspect metamodel interpreter to prototype aspect languages. From their perspective, our framework offers the *flexibility* to create many different types of aspect-based compositions. However, the flexibility provided by such a dynamic framework, allowing for the quick prototyping of languages, comes at the cost of (run time) efficiency, as discussed in detail in section 4.6.1. In addition, there are intentional limits to the flexibility of the framework: it does enforce a high-level structure and control flow of aspects. Since many aspect languages can be mapped to this common structure and control flow, we can detect when multiple aspects (even if written in distinct languages) affect the same points in the program, and generate warning or error messages if desired, thus improving the *analyzability* of the program. In addition, the common model allows us to express constraints over aspects, providing a resolution mechanism for potential composition conflicts, further enhancing the *analyzability*.

In chapter 5, we introduce a language that supports user-defined composition mechanisms. This allows for a much enhanced *flexibility* in the definition of composition mechanisms. From the perspective of a software engineer, this should lead to more *concise* and *(locally) comprehensible* programs – as no (verbose) workarounds are necessary to “emulate” composition techniques that are not directly supported by the language – we gave several examples of such workarounds in the introduction of chapter 5, and discussed why such workarounds often have serious drawbacks. As an example of the support for domain-specific composition mechanisms, section 5.3.6 shows a mechanism that allows to concisely express the observer pattern.

From the perspective of a *language engineer*, *Co-op* supports the definition of many composition mechanisms, and allows to define constraints between them. This increases the *flexibility* and *analyzability* as compared to languages that do not support such custom-defined composition mechanisms. However, the *global comprehensibility* of the entire system may decrease in the presence of (multiple) custom-defined composition mechanisms, since these may interact in undesired ways. We mitigate this partially by supporting the expression of constraints between composition mechanisms, since constraints can be specified to disambiguate the desired (partial) order or dependencies between composition mechanisms. However, such constraints still have to be defined (and maintained) by the designers of the involved composition mechanisms, which is not a straightforward task. In addition, the same concerns about flexibility vs. efficiency apply as discussed in relation to chapter 4.

For this reason, potential future work includes optimizations to the efficiency, as well as attempting to limit or analyze the potential interactions between composition mechanisms, or at least, to facilitate the definition of expressing constraints between them at a more semantic level, as discussed in section 4.6.4.

A The semantics of Co-op

This appendix describes the denotational semantics of the *Co-op* language, by means of defining the abstract syntax in terms of type definitions in the functional programming language Miranda [124], and consequently defining meaning functions for each element in this abstract syntax.

This appendix is structured as follows: first, we show the abstract syntax definition in terms of Miranda type definitions. Miranda also allows the definition of Abstract Data Types, i.e., a set of functions that may operate on a specific type, while hiding the actual implementation details of the type (i.e., its comprising data elements) from the rest of the program. We define such data types for the most-used elements in the abstract syntax, as this facilitates a much clearer expression of the meaning functions. In addition, we define several abstract data types that are part of the *semantic* domain, which are used to exchange information between the meaning functions. For example, we define data types to represent the dynamic program state and call stack, which may be manipulated by the various meaning functions. Subsequently, we define meaning functions for each element in the abstract syntax.

Finally, we show an example program, and briefly show how a Miranda interpreter can be used to "simulate" its semantics based on the previously defined meaning functions.

This approach to modeling the semantics of a programming language was inspired by the paper "Programming Language Semantics using Miranda" [123].

For the purpose of modeling the semantics of *Co-op* as described in this appendix, we made some modifications to the *Co-op* language. Most importantly, we directly modeled the primitive elements used to construct composition mechanisms, since these are the primary elements of which the semantics are of interest. This is as opposed to the actual implementation, where these primitive elements are modeled as "normal" first-class objects, and can thus be passed around the program as parameters, instance variables, etc. In addition, we omitted the modeling of closures and operation parameters, since these would mainly add complexity to the model without further contributing towards a clarification of the core composition primitives.

A.1 Abstract syntax definition

Listing A.1 defines the abstract syntax of *Co-op* in terms of Miranda type definitions. In Miranda, a list of items is denoted by brackets (e.g. $[items]$), and tuples are denoted by parentheses (e.g. (x, y, z)). Note that the type definitions closely follow the abstract syntax of *Co-op* as introduced in section 5.2. Specifically, the abstract syntax specification uses three types of production rules, which are mapped to Miranda as follows: *aggregates* (e.g., s_module) are modeled in Miranda as tuples, *choices* (e.g. $s\_statement$) are modeled as user-defined types, and *lists* (e.g. $s\_program$) are directly modeled as (Miranda) lists.

```
|| A program consists of a list of modules
s_program == [s_module]
|| Module: name, module body
s_module == (string, s_module_body)
|| Module body: instance variables, operation declarations
s_module_body == (s_variable_list, s_operation_list)
s_variable_list == [s_variable]
|| A variable declaration consists of a name only
s_variable == string
s_operation_list == [s_operation]
|| Operation: name, formal params (not used), body
s_operation == (string, s_variable_list, s_operation_body)
|| Operation body: local variable declarations, list of statements
s_operation_body == (s_variable_list, s_statement_list) || local var
    decls, list of statements
s_statement_list == [s_statement]
|| A statement can be either Assignment, Return or Event generation
s_statement ::= Assignment s_variable s_expression |
                ReturnStatement s_expression |
                EventStatement s_expression operation_selector
s_expression ::= Literal s_literal |
                 EventExpression s_expression operation_selector |
                 LocalVariable s_variable |
                 InstanceVariable s_variable |
                 NewExpression mytype
s_literal ::= BoolLiteral bool | StringLiteral string
|| Event selectors modeled as a function of event X state -> boolean
s_event_selector == event->state->bool
|| Action selectors = function of event X state -> operation ref
s_action_selector == event->state->s_operation_ref
```

```
s_binding ::= Binding num s_event_selector s_action_selector
    s_context_binding_list
|| Model the three types of supported constraints
s_constraint ::= PreConstraint s_binding s_binding |
                 CondConstraint s_binding s_binding |
                 SkipConstraint s_binding s_binding
s_context_binding_list == [s_context_binding]
s_context_binding == (s_variable,s_contextvarbinding)
|| Context variable bindings modeled as a (Miranda) function, rather
     than an s_expression
s_contextvarbinding == event->state->object
|| Operation reference: lookuptype, selector, annotation
s_operation_ref ::= Nil | OperationRef mytype string string
```

Listing A.1: *Co-op* abstract syntax definition

Note that the definitions above make use the following common type definitions:

```
string == [char]
mytype == string
obj_id == num

operation_selector == (string, string) || selector, annotation
```

Listing A.2: Common type definitions

A.1.1 Abstract type definitions

We create abstract type definitions for the three most-used elements in the abstract syntax (module, operation, and binding), such that the meaning functions have access to a convenient interface to work with data of these types.

Each abstract data type defines a set of functions over the specified type. We show the type declarations of these functions, as well as their implementations[1].

The abstract data type *s_module* provides access to the instance variables and operations encapsulated by that module.

```
|| Module::interface
abstype s_module
with
 make_module::string->s_variable_list->s_operation_list->s_module
 get_name::s_module->string
 get_instancevars::s_module->s_variable_list
```

[1]For readers who are not familiar with Miranda, such abstract data types can be compared to classes that store their data in private instance variables, and provide convenient accessor methods for the information they encapsulate.

```
  get_operations::s_module->s_operation_list
  get_operation_by_name::s_module->string->s_operation
  has_operation::s_module->string->bool
  has_instancevar::s_module->string->bool

|| Module::implementation
s_module == (string, s_module_body)

make_module name instancevars operations = (name, (instancevars,
    operations))
get_name (name, body) = name
get_instancevars (name, (ivs, ops)) = ivs
get_operations (name, (ivs, ops)) = ops
get_operation_by_name (name, (ivs, ops)) op = hd (filter (
    h_match_operation_id op) ops)
has_operation (name, (ivs, ops)) op = member (map get_id ops) op
has_instancevar (name, (ivs,ops)) iv = member ivs iv

h_match_operation_id::string->s_operation->bool
h_match_operation_id op_id op = (get_id op)=op_id
```

Listing A.3: Abstract data type definition: Module

The abstract data type *s_operation* provides access to the name, parameters and body of its encapsulated operation.

```
|| Operation::interface
abstype s_operation
with
  make_operation::string->s_variable_list->s_operation_body->
      s_operation
  get_id::s_operation->string
  get_formal_params::s_operation->s_variable_list
  get_body::s_operation->s_operation_body

|| Operation::implementation
s_operation == (string, s_variable_list, s_operation_body)

make_operation name formal_params body = (name,formal_params,body)
get_id (name, formal_params, body) = name
get_formal_params (name, formal_params, body) = formal_params
get_body (name, formal_params, body) = body
```

Listing A.4: Abstract data type definition: Operation

Finally, the abstract data type *s_binding* provides access to its encapsulated event selector, action selector, and context variable binding specifications. In addition, the type specifies a function that enables comparison of bindings based on a (unique) identifier[2].

[2]In this specification, the elements of bindings are specified as functions themselves. Since

```
|| Binding::interface
abstype s_binding
with
 make_binding::num->s_event_selector->s_action_selector->s_context_binding_list->s_binding
 binding_equals::s_binding->s_binding->bool
 binding_member::[s_binding]->s_binding->bool
 binding_id::s_binding->num
 matches_id::num->s_binding->bool
 binding_eventselector::s_binding->s_event_selector
 binding_actionselector::s_binding->s_action_selector
 binding_varbindings::s_binding->s_context_binding_list

|| Binding::implementation
s_binding == t_binding
t_binding ::= Binding num s_event_selector s_action_selector s_context_binding_list

make_binding bnd_id event_sel action_sel varbindings = Binding bnd_id event_sel action_sel varbindings
binding_id (Binding bnd_id event_sel action_sel varbindings) = bnd_id
matches_id i (Binding bnd_id event_sel action_sel varbindings) = bnd_id=i
binding_equals a b = binding_id a = binding_id b
binding_member lst bnd = member (map binding_id lst) (binding_id bnd)
binding_actionselector (Binding bnd_id event_sel action_sel varbindings) = action_sel
binding_eventselector (Binding bnd_id event_sel action_sel varbindings) = event_sel
binding_varbindings (Binding bnd_id event_sel action_sel varbindings) = varbindings
```

Listing A.5: Abstract data type definition: Binding

A.2 Denotational semantics

In this section, we define a meaning function for each element in the abstract syntax tree, thus establishing a denotational semantics of the *Co-op* language. Note that such meaning functions may have two distinct kinds of parameters: elements from the *syntactic domain*, which in this specification are identifiable by their prefix, "*s_*", and parameters to exchange information in the *semantic domain*, for example to track the current state of the program.

Miranda does not allow the direct comparison of functions, we gave each binding a unique id, such that constraint specifications can identify each specific binding.

A.2.1 Abstract type definitions

The meaning functions make use of three additional abstract data types, which are used to model the most important elements in the semantic domain: events, program state, and stack. We show the definitions of each of these types:

The *event* abstraction directly corresponds to the definition given in chapter 5:

```
abstype event
with
  generate_event::object->mytype->object->mytype->
      operation_selector->event
  get_sender::event->object
  get_localtype::event->mytype
  get_target::event->object
  get_lookuptype::event->mytype
  set_lookuptype::event->mytype->event
  get_operation_selector::event->operation_selector
  set_operation_selector::event->operation_selector->event

|| Event::implementation
event == (object, mytype, object, mytype, operation_selector) ||
    sender, localtype, target, lookuptype, selector, params

generate_event sender localtype target lookuptype operation_selector
     = (sender,localtype,target,lookuptype,operation_selector)
get_sender (sender,localtype,target,lookuptype,operation_selector) =
     sender
get_localtype (sender,localtype,target,lookuptype,operation_selector
    ) = localtype
get_target (sender,localtype,target,lookuptype,operation_selector) =
     target
get_lookuptype (sender,localtype,target,lookuptype,
    operation_selector) = lookuptype
set_lookuptype (sender,localtype,target,lookuptype,
    operation_selector) new_lookup = (sender,localtype,target,
    new_lookup,operation_selector)
get_operation_selector (sender,localtype,target,lookuptype,
    operation_selector) = operation_selector
set_operation_selector (sender,localtype,target,lookuptype,
    operation_selector) new_opsel = (sender,localtype,target,
    lookuptype,new_opsel)
```

Listing A.6: Abstract type definition: Event

The meaning of a *Co-op* program can be modeled in terms of a state that is modified by the execution of operations and statements. We model this state as consisting of a (call) stack and a set of instance variables. In addition, reflective information about the program is available as part of the state, including bindings, constraints and the modular structure of the program itself. This is necessary to allow the definition of event- and action-selectors that depend on such reflective information.

```
|| State::interface
abstype state
with
 initialstate::state
 get_stack::state->stack
 open_stackframe::state->state
 close_stackframe::state->state
 get_instance_vars::state->instance_vars
 get_instance_var::state->obj_id->s_variable->object
 set_instance_var::state->obj_id->s_variable->object->state
 get_bindings::state->[s_binding]
 get_binding_by_id::state->num->s_binding
 add_binding::s_binding->state->state
 get_constraints::state->[s_constraint]
 add_constraint::s_constraint->state->state
 get_modules::state->[s_module]
 add_module::s_module->state->state
 set_stack::stack->state->state
 has_module::state->string->bool
 get_nextid::state->num
 inc_nextid::state->state
 get_module_by_name::state->string->s_module
 showstate::state->[char] || Used by miranda to show (print)
     contents of a state

|| State::implementation
state == (stack, instance_vars, [s_binding], [s_constraint],
    s_program, num)

|| type 'stack' is defined later, type instance_vars is defined as:
instance_vars == [(obj_id, variable)] || Object to which the var
    belongs,
variable == (string, object) || var name, current value

initialstate = (emptystack, [], [default_binding], [], [], 1)
get_stack (s, iv, bnd, ct, mods, next_id) = s
open_stackframe s = set_stack (open_frame (get_stack s)) s
close_stackframe s = set_stack (close_frame (get_stack s)) s
get_instance_vars (s, iv, bnd, ct, mods, next_id) = iv
get_instance_var state_in this varname
  = snd (hd (filter (match_key varname) thisvars))
    where
     thisvars = map snd (filter (match_key this) (get_instance_vars
         state_in))
set_instance_var (s, ivs, bnd, ct, mods, next_id) this varname
    newvalue = (s, h_set_ivar ivs this varname newvalue, bnd, ct,
    mods, next_id)
get_bindings (s, iv, bnd, ct, mods, next_id) = bnd
get_binding_by_id s bnd_id = hd (filter (matches_id bnd_id) (
    get_bindings s))
add_binding newbind (s, iv, bnd, ct, mods, next_id) = (s, iv,
    newbind:bnd, ct, mods, next_id)
get_constraints (s, iv, bnd, ct, mods, next_id) = ct
add_constraint newct (s, iv, bnd, ct, mods, next_id) = (s, iv, bnd,
    newct:ct, mods, next_id)
get_modules (s, iv, bnd, ct, mods, next_id) = mods
set_stack newstack (oldstack, iv, bnd, ct, mods, next_id) = (
    newstack, iv, bnd, ct, mods, next_id)
add_module modl (s, iv, bnd, ct, mods, next_id) = (s, iv, bnd, ct,
    modl:mods, next_id)
has_module s name = member (map get_name (get_modules s)) name
get_module_by_name s name = hd (filter (h_match_module_name name) (
    get_modules s))
get_nextid (s, iv, bnd, ct, mods, next_id) = next_id
inc_nextid (s, iv, bnd, ct, mods, next_id) = (s, iv, bnd, ct, mods,
    next_id+1)
showstate (s, iv, bnd, ct, mods, next_id) = show ("Stack: " ++ show
    s ++ "; Instancevars: " ++ show iv ++ "; Bindings: " ++ show bnd
    )

h_set_ivar::instance_vars->obj_id->s_variable->object->instance_vars
h_set_ivar [] this varname newvalue = [(this, (varname, newvalue))]
h_set_ivar ((obj, (name, value)):iv) this varname newvalue
 = ((obj, (name, newvalue)):iv), if varname=name & this=obj
 = ((obj, (name, value)):(h_set_ivar iv this varname newvalue)),
      otherwise

h_match_module_name::string->s_module->bool
h_match_module_name name modl = (get_name modl)=name
```

Listing A.7: Abstract type definition: State

The data type `stack`, defined in listing A.8, offers functions to open or close stack frames, and includes convenience functions to obtain or set the value of variables on the stack. In addition, each stack frame can store the return value of an operation invocation corresponding to this stack frame.

```
|| Stack::interface
abstype stack
with
 emptystack::stack
 open_frame::stack->stack
 close_frame::stack->stack
 set_retval::object->stack->stack
 get_retval::stack->object
 get_var::stack->string->object
 has_var::stack->string->bool
 set_var::stack->string->object->stack
 create_var::stack->string->stack
 showstack::stack->[char]

|| Stack::implementation
```

```
stack == [frame]
frame == ([variable], object) || local vars, return value

emptystack = open_frame []
open_frame stack_in = ([], undefined):stack_in
close_frame (x:xs) = xs
set_retval new_retval ((vars, retval):xs) = (vars, new_retval):xs
get_retval ((vars, retval):xs) = retval
has_var ((vars, retval):xs) var = member (map fst vars) var
get_var ((vars, retval):xs) var
 = snd (hd (filter (h_match_key var) vars)), if member (map fst vars
     ) var
 = (StringObject ("var not found "++var)), otherwise
set_var ((vars, retval):xs) var value
 = ((h_replace_var var value vars), retval):xs, if member (map fst
     vars) var
 = ((vars, retval):xs), otherwise
create_var ((vars, retval):xs) newvar = (((newvar,undefined):vars,
     retval):xs)
showstack stk = show (stk)

h_match_key::*->(*,**)->bool
h_match_key comp (key,val) = key=comp

h_replace_var::s_variable->object->[variable]->[variable]
h_replace_var target value [] = []
h_replace_var target value ((var,val):xs)
 = (target,value):xs, if target=var
 = (var,val):(h_replace_var target value xs), otherwise
```

Listing A.8: Abstract type definition: Stack

A.2.2 Definition of meaning functions

The meaning of a *Co-op* program is defined as the meaning of invoking the operation "main" on an internally created variable of type "Main". As input, the function *m_program* takes an initial state. Note that since the state includes the program structure (as this is used for reflective purposes), no explicit *s_program* parameter is necessary. The meaning-function returns a pair, consisting of the return value of operation main(), and the final state after the program has been evaluated.

```
|| Equivalent of a VM that internally executes "var _main; _main = [
     Main new]; [_main main]"
m_program::state->(object,state)
m_program state_init = m_expression (EventExpression (LocalVariable "
     _main") ("main", "")) main_state
                       where
                        main_state = m_statement (Assignment "_main" (
                             NewExpression "Main")) state_obj
```

```
                    state_obj = m_local_var_decl "_main" state_init
```

Listing A.9: Meaning function: m_program

The meaning of an operation is defined in terms of the meaning of its body. The operation body may also set a return value for this operation (otherwise, the return value is $undefined$).

```
m_operation::s_operation->state->(object,state)
m_operation op state_in
 = (get_retval (get_stack newstate), newstate)
   where
     newstate = m_operation_body (get_body op) state_in
```

Listing A.10: Meaning function: m_operation

The meaning of an operation body is defined in terms of the meaning of the list of statements specified as part of the operation body. Any local variable declarations are added to the input state, yielding the state that is used as the input for the meaning function of the list of statements. Consequently, the meaning of a list of statements is defined as the meaning of applying the first statement to the input state, and applying each subsequent statement to the resulting state of the previous statement's application (the Miranda function "foldr" that is used to accomplish this is also known as "reduce" in some languages).

```
m_operation_body::s_operation_body->state->state
m_operation_body (vars,statements) state_in
  = m_statement_list statements state_with_vars
    where
     state_with_vars = m_local_var_decls vars state_in

m_local_var_decls::s_variable_list->state->state
m_local_var_decls vars state_in = foldr m_local_var_decl state_in vars

m_local_var_decl::s_variable->state->state
m_local_var_decl var state_in = set_stack (create_var (get_stack
     state_in) var) state_in

m_statement_list::s_statement_list->state->state
m_statement_list statements state_in = foldr m_statement state_in (
     reverse statements)
```

Listing A.11: Meaning function: m_operation_body

Co-op supports three kinds of statements, the implementation of $m\_statement$, in listing A.12, shows the meaning of each of them. Each statement transforms an input state to an output state. The meaning of an $EventStatement$ with a given target expression and selector is defined in terms of the meaning of an *EventExpression* with the same target expression and selector, but ignores the return value (if any) that may have been set by operation(s) invoked as a result of evaluating the *EventExpression*. The *EventExpression* may still have other side effects, so the potentially modified state is returned.

An *Assignment* statement evaluates the meaning of the target expression, and based on its return value, assigns this value to the indicated target variable, first searching the local stack frame for this variable, and otherwise assuming it is an instance variable.

A *ReturnStatement* sets the return value (included in the stack) of the current operation invocation (which corresponds to the top stack frame). This implementation of the Return statement does not affect the control flow (i.e., immediately exiting the operation, potentially skipping remaining statements); the actual implementation of *Co-op* does affect control flow, however.

```
m_statement::s_statement->state->state

m_statement (EventStatement target_expr selector) state_in
 = snd (m_expression (EventExpression target_expr selector) state_in
     )
m_statement (Assignment target value) state_in
 = set_stack (set_var (get_stack state_new) target exprval) state_new
     , if has_var (get_stack state_new) target
 = set_instance_var state_new (h_get_this_id state_new) target exprval
     , otherwise
   where
   (exprval, state_new) = m_expression value state_in
m_statement (ReturnStatement expr) state_in
 = set_stack (set_retval value (get_stack newstate)) newstate
   where
    (value,newstate) = m_expression expr state_in

h_get_this_id::state->obj_id
h_get_this_id state_in = (object_id (m_local_variable "this" state_in))
```

Listing A.12: Meaning function: m_statement

As defined in the abstract syntax definition, *Co-op* supports several types of expressions. Literal expressions do not affect the state; their meaning is defined by the function *m_literal*, which simply returns the value (represented as a boolean- or string-object) of the literal. Local variables are obtained from the stack, as defined by *m_local_variable*, whereas instance variables are obtained from the state, as defined by *m_instance_variable*. A `NewExpression` returns a new object of the indicated type. Finally, an `EventExpression` "generates" an event, and invokes the helper function "aspect evaluator" to evaluate any bindings that may be interested in this event. Below, we discuss this in more detail.

```
m_expression::s_expression->state->(object,state)

m_expression (Literal l) state_in = (m_literal l, state_in)
m_expression (LocalVariable v) state_in = (m_local_variable v state_in, state_in)
m_expression (InstanceVariable v) state_in = (m_instance_variable v state_in, state_in)
m_expression (NewExpression mytp) state_in = (Object (get_nextid state_in) mytp, (inc_nextid state_in))

m_expression (EventExpression target_expr selector) state_in
 = (result, outstate), if success
 = (undefined, state_in), otherwise
   where
    (success,result,outstate) = h_aspect_evaluator ev state_two
    (ev,state_two) = (generate_event sender (object_type sender) target (object_type target) selector,state_one)
                 where
                  (target, state_one) = m_expression target_expr state_in
                  sender = get_var (get_stack state_in) "this"
m_expression (ObjRef o) state_in = (o, state_in) || Direct object reference, for internal use

m_local_variable::s_variable->state->object
m_local_variable v state_in = get_var (get_stack state_in) v

m_instance_variable::s_variable->state->object
m_instance_variable var state_in = get_instance_var state_in (h_get_this_id state_in) var

m_literal::s_literal->object
m_literal (BoolLiteral b) = BoolObject b
m_literal (StringLiteral s) = StringObject s

object ::= Object obj_id mytype || Object reference
         | StringObject string || Literal string value
         | BoolObject bool || Literal boolean value
         | DictionaryObject [(object,object)] || Not implemented
         | ListObject [object]        || Not implemented

undefined = Object 0 "null" || Define default "null" object
```

Listing A.13: Meaning function: m_expression

The aspect evaluator, defined in listing A.14, functions as follows: it evaluates the *event selectors* defined by all of the active bindings (which are part of the input state), and gathers those that match the current event. Then, it calls a helper function that applies constraints and invokes any viable bindings. This function, *h_apply_constraints_and_invoke*, determines which of the matching bindings are viable to be executed (which is the case if there are no active constraints that prevent it from being executed at a given moment during the execution), executes the operation of one of the viable bindings indicated by its corresponding action selector, and repeats this process until no more bindings are viable to be executed. Bindings of which the corresponding action has already been executed are not considered for execution again, at the same event.

```
h_aspect_evaluator::event->state->(bool,object,state)
h_aspect_evaluator ev state_in
 = h_apply_constraints_and_invoke matching_bindings [] ev state_in undefined
  where
   matching_bindings = map get_bind (filter binding_matches [(bind, ev, state_in) | bind <- get_bindings state_in])

h_get_bind::(*,**,***)->*
h_get_bind (bnd,ev,state) = bnd

h_apply_constraints_and_invoke::[s_binding]->[(s_binding,bool)]->event->state->object->(object,state)
h_apply_constraints_and_invoke [] executed ev state_in curr_result = (False, curr_result,state_in)
h_apply_constraints_and_invoke matching executed ev state_in curr_result
 = h_invoke_next matching executed ev state_in curr_result (h_get_viable_bindings matching executed state_in)

h_get_viable_bindings::[s_binding]->[(s_binding,bool)]->state->[s_binding]
h_get_viable_bindings all_matching executed state_in
 = map fst (filter (snd) [ (thisbinding, and (map (m_constraint all_matching executed thisbinding) (get_constraints state_in))) | thisbinding <- still_to_execute])
  where
   still_to_execute = filter ((~).(binding_member (map fst executed))) all_matching

h_invoke_next::[s_binding]->[(s_binding,bool)]->event->state->object->[s_binding]->(object,state)
h_invoke_next matching executed ev state_in curr_result [] = (or (map snd executed), curr_result, state_in)
h_invoke_next matching executed ev state_in curr_result viable
 = h_apply_constraints_and_invoke matching (((hd viable), this_succeeds):executed) ev newstate (iftrue this_succeeds this_result curr_result)
  where
   (this_succeeds,this_result,newstate) = h_binding_to_invocation (hd viable) ev state_in

iftrue::bool->*->*->*
```

```
iftrue b x y = x, if b
         = y, otherwise

h_binding_to_invocation::s_binding->event->state->(bool,object,state
    )
h_binding_to_invocation bnd ev state_in
 = (False, undefined, state_in), if opref = Nil
 = h_invoke_method (opref_to_selector opref) varbind_state, if (
     binding_id bnd)=0 || default binding: final dispatch
 = h_aspect_evaluator mod_event state_in, otherwise || Next dispatch
     stage with modified event
    where
     (opref,varbind_state) = m_binding bnd ev (open_stackframe
        state_in) || open_stackframe!
     mod_event = set_lookuptype (set_operation_selector ev (new_sel,
        new_annot)) new_lookup
            where
             (OperationRef new_lookup new_sel new_annot) = opref

opref_to_selector::s_operation_ref->operation_selector
opref_to_selector (OperationRef modl selector annot) = (modl,
    selector)

h_invoke_method::operation_selector->state->(bool,object,state)
h_invoke_method (module,operation) state_in
 = (True, retval, close_stackframe newstate), if (has_module
     state_in module & (has_operation (get_module_by_name state_in
     module) operation))
 = (False,undefined,state_in), otherwise || Invocation fails because
     specified operation does not exist (in this module)
   where
    (retval, newstate) = m_operation (get_operation_by_name (
       get_module_by_name state_in module) operation) (
       open_stackframe state_in)

binding_matches::(s_binding,event,state)->bool
binding_matches (bnd, ev, state_in) = fst (m_binding bnd ev state_in)
     ~= Nil
```

Listing A.14: Definition of aspect evaluator

As discussed in chapter 5, *Co-op* defines three kinds of constraints, the formal meaning of each is defined below. Note that these definitions behave the same as those defined as part of the Java Aspect Metamodel Interpreter, in chapter 4.

```
m_constraint::[s_binding]->[(s_binding,bool)]->s_binding->
    s_constraint->bool

|| Preconstraint: returns false only if checked binding is b2, both
    b1 and b2 match at this point, and b1 has not yet been executed
m_constraint all_matching executed checkviability (PreConstraint b1
    b2)
```

```
 = False, if (binding_equals checkviability b2) &
   (binding_member all_matching b1) & (binding_member all_matching b2) & ~(h_executed b1 executed)
 = True, otherwise

|| Conditionalconstraint: returns false only if checked binding is b2 and b1 has not been executed yet, or b1 failed execution.
m_constraint all_matching executed checkviability (CondConstraint b1 b2)
 = False, if (binding_equals checkviability b2) &
    ( ~(h_executed b1 executed) \/
      ((h_executed b1 executed) & ~(h_executed_succesfully b1 executed)))
 = True, otherwise

|| SkipConstraint: returns false only if checked binding is b2 and either b1 was succesfully executed, or if b1 matches but is not executed yet
m_constraint all_matching executed checkviability (SkipConstraint b1 b2)
 = False, if (binding_equals checkviability b2) &
           (((h_executed b1 executed) & (h_executed_succesfully b1 executed)) \/
            (binding_member all_matching b1) & ~(h_executed b1 executed))
  = True, otherwise

h_executed::s_binding->[(s_binding,bool)]->bool
h_executed bnd executed = binding_member (map fst executed) bnd

h_executed_succesfully::s_binding->[(s_binding,bool)]->bool
h_executed_succesfully bnd executed = snd (hd (filter (match_key (binding_id bnd)) [(binding_id b,ex) | (b,ex) <- executed]))
```

Listing A.15: Meaning function: m_constraint

The other composition primitives, i.e., event selectors, action selectors and bindings, are defined in listing A.16. Since, for the purposes of this definition, event- and action-selectors are specified directly in terms of (simple) Miranda functions, their meaning is simply to evaluate that function (below, we will show example selector definitions). The meaning of a binding is defined as resulting in an operation reference as specified by its action selector, but only if its event selector evaluates to true given the specified event. In addition, a binding can bind values to "context variables", which are used to define pseudo-variables such as `self`. These variables are added to the stack (under the assumption that they do not conflict with existing local variable definitions).

```
|| Meaning functions of composition primitives
m_event_selector::s_event_selector->event->state->bool
```

```
m_event_selector select ev state_in = select ev state_in || "evaluate"
     the selector

m_action_selector::s_action_selector->event->state->s_operation_ref
m_action_selector select ev state_in = select ev state_in || "evaluate
    " the selector

m_binding::s_binding->event->state->(s_operation_ref,state)
m_binding bnd ev state_in
 = (m_action_selector (binding_actionselector bnd) ev varbind_state,
     varbind_state), if m_event_selector (binding_eventselector bnd)
     ev varbind_state
 = (Nil, state_in), otherwise
   where
    varbind_state = m_statement_list [Assignment var (ObjRef (value
        ev vardecl_state)) | (var, value) <- contextvarbindings]
        vardecl_state
    vardecl_state = m_local_var_decls (map fst contextvarbindings)
        state_in || Declare the variables in local scope
    contextvarbindings = binding_varbindings bnd
```

Listing A.16: Meaning functions of composition primitives

The definition of the default binding, which is included in the default initial state returned by *initialstate* (defined in listing A.7), is shown in listing A.17. The event selector used by the default binding matches any event (i.e., always returns True), regardless of any event properties or the current program state. The corresponding action selector returns an operation reference that, when invoked, will evaluate the operation indicated by the lookup-type of the current event, and its operation selector. In addition, it defines a context variable "this", which is bound to the target of the current event.

```
default_binding::s_binding
default_binding = make_binding 0 h_event_match_all h_event_to_current
    [("this", h_event_target)]

h_event_target::s_contextvarbinding
h_event_target event state_in = get_target event

h_event_match_all::s_event_selector
h_event_match_all event state_in = True || Match any event regardless
    of the event properties or current state

h_event_to_current::s_action_selector
h_event_to_current event state_in = OperationRef (get_lookuptype event
    ) (fst (get_operation_selector event)) (snd (
    get_operation_selector event))
```

Listing A.17: Definition of the default binding

This completes the description of the denotational semantics of *Co-op*. In the section below, we show how the definitions above can be used to simulate the semantics of some actual *Co-op* programs and composition mechanisms. The complete source code of the above specifications can also be downloaded from the *Co-op* website; a Miranda interpreter is available free of charge for most operating systems in common use, from the Miranda homepage [6].

A.3 Program simulation example

In this section, we define small example *Co-op* programs, expressed in terms of the types defined in section A.1, and show how their semantics can be simulated based on the definitions in section A.2, by using a Miranda interpreter.

For this purpose, listing A.18 defines a simplified version of the inheritance-based example found in chapter 5, listing 5.4. Note that the listing below only defines the module structure, we will later discuss the inheritance structure, and how this can be expressed using the Miranda-based semantics.

```
module Widget {
  display() { return [this draw]; }
}

module Window {
  var @title;

  draw() { return "Drawing Window"; }
  setTitle() { @title = "A window title"; }
  getTitle() { return @title; }
}

module DialogWindow {
  var @text;

  draw { return "Drawing DialogWindow"; }
  setText() { @text = "A window text"; }
  getText() { return @text; }
}
```

Listing A.18: Simplified *Co-op* example

The above set of modules can be modeled in Miranda as shown in listing A.19.

```
|| display() { return [this draw]; }
op_widget_display = make_operation "display" [] ([], [ReturnStatement
    (EventExpression (LocalVariable "this") ("draw", ""))])

|| module Widget { display() { .. } }
```

```
mod_widget = make_module "Widget" [] [op_widget_display]

|| draw() { return "Drawing Window"; }
op_window_draw = make_operation "draw" [] ([], [ReturnStatement (
    Literal (StringLiteral "Drawing Window"))])

|| setTitle() { @title = "A window title"; }
op_window_setTitle = make_operation "setTitle" [] ([], [Assignment "
    @title" (Literal (StringLiteral "A window title"))])

|| getTitle() { return @title; }
op_window_getTitle = make_operation "getTitle" [] ([], [
    ReturnStatement (InstanceVariable "@title")])

|| module Window { var @title; draw() { .. } setTitle() { .. }
    getTitle { .. }}
mod_window = make_module "Window" ["@title"] [op_window_draw,
    op_window_setTitle, op_window_getTitle]

|| draw() { return "Drawing DialogWindow"; }
op_dialog_draw = make_operation "draw" [] ([], [ReturnStatement (
    Literal (StringLiteral "Drawing DialogWindow"))])

|| setText() { @title = "A window text"; }
op_dialog_setText = make_operation "setText" [] ([], [Assignment "
    @text" (Literal (StringLiteral "A window text"))])

|| getText() { return @title; }
op_dialog_getText = make_operation "getText" [] ([], [ReturnStatement
     (InstanceVariable "@text")])

|| module DialogWindow { var @text; draw() { .. } setText() { .. }
    getText { .. }}
mod_dialog = make_module "DialogWindow" ["@text"] [op_dialog_draw,
    op_dialog_setText, op_dialog_getText]
```

Listing A.19: *Co-op* example, expressed in Miranda

To demonstrate the semantics of the aspect evaluator in the presence of multiple bindings and constraints, we express the single-inheritance mechanism as defined in listing 5.3, in terms of our Miranda-based semantics definition. Listing A.20 shows a generic implementation of the binding *extends*. As input, this function takes two types (representing the *child* and *parent*), as well as an input state. As output, it returns the input state, to which an inheritance binding and constraint have been added. Note that the constraint $inherit\_constraint$ is defined as a `Skip`-constraint between the (previously defined) $default\_binding$ and the binding $inherit\_bind$, defined here. As in listing 5.3, the inheritance binding matches events of which the lookup type matches the specified *child* type (as implemented here by h_match_child), and forwards the call to the specified *parent* type, as implemented here by function *h_to_parent*.

```
extends::mytype->mytype->state->state
extends child parent state_in
 = add_constraint inherit_constraint (add_binding inherit_bind (inc_nextid state_in))
   where
    inherit_bind = make_binding (get_nextid state_in) (h_match_child child) (h_to_parent parent) [("this", h_event_target)]
    inherit_constraint = (SkipConstraint default_binding inherit_bind)

h_match_child::string->s_event_selector
h_match_child child event state_in = (get_lookuptype event) = child

h_to_parent::string->s_action_selector
h_to_parent parent event state_in = OperationRef parent (fst (get_operation_selector event)) (snd (get_operation_selector event))
```

Listing A.20: Miranda implementation of binding *Extends*

Using the definitions from listings A.19 and A.20, we can now construct various test cases. Listing A.21 defines a module `Main`, with an operation `main` that creates an object of type `DialogWindow`, and invokes the operation `draw` on it (lines 4+5). Line 6 creates a state that contains this module `Main`, as well as the previously defined modules `Widget`, `Window`, and `DialogWindow`. Finally, line 7 applies the "extends" function to this state twice, establishing the inheritance relations from `DialogWindow` to `Window`, and from `Window` to `Widget`.

```
1 || module Main() {
2 ||  main() { var testdialog; testdialog = [DialogWindow new]; return [testdialog draw]; }
3 || }
4 op_main_1 = make_operation "main" [] (["testdialog"],[Assignment "testdialog" (NewExpression "DialogWindow"), ReturnStatement (EventExpression (LocalVariable "testdialog") ("display", ""))])
5 mod_main_1 = make_module "Main" [] [op_main_1]
6 modules_1 = foldr add_module initialstate [mod_widget,mod_window,mod_dialog,mod_main_1]
7 program_1 = extends "DialogWindow" "Window" (extends "Window" "Widget" modules_1)
```

Listing A.21: *Co-op* semantics test case (1)

The semantics of the above program can be simulated using a Miranda interpreter (e.g., as can be found on the Miranda homepage [6]), by loading the definitions presented in this appendix, and then evaluating the expression *m_program program_*1, as shown in listing A.22:

```
|| Result of evaluating Miranda expression "m_program program_1":
(StringObject "Drawing DialogWindow","Stack: [([(\"_main\",Object 3
    \"Main\")],Object 0 \"null\")]; Instancevars: []; Bindings: [<
    abstract ob>,<abstract ob>,<abstract ob>]")
```

Listing A.22: Simulation of test case 1

As the definition of *m_program* specifies, its output consists of the return value of operation `main`, in addition to the final state when the program finished running. In this case, the operation `main` returns the string-literal “Drawing DialogWindow”, which is the result of invoking [*testdialogdraw*], which is forwarded from module `DialogWindow` via `Window` to module `Widget`. The implementation of operation `draw` (in module Widget) then returns the value of [*thisdraw*], which (since the `this`-object is set to the original `testdialog` object of type `DialogWindow`) is first sent to `DialogWindow` again, and leads to invocation of the operation `draw` defined by that module. The final state shows that after finishing evaluation of the program, the stack only contains the internally defined variable *_main* (which is the correct behavior); no instance variables have been assigned a value in this example, and there are three active bindings in this program, i.e., the default binding, and the two inheritance bindings.

Listing A.23 shows the same example, except that it defines a variable `testwindow`, of type `Window` (rather than `DialogWindow`). In this case, evaluation of *m_programprogram_*2 correctly returns the value “Drawing Window”.

```
|| module Main() {
||   main() { var testwindow; testwindow = [Window new]; return [
    testwindow display]; }
|| }
op_main_2 = make_operation "main" [] (["testwindow"],[Assignment "
    testwindow" (NewExpression "Window"), ReturnStatement (
    EventExpression (LocalVariable "testwindow") ("draw", ""))])
mod_main_2 = make_module "Main" [] [op_main_2]
modules_2 = foldr add_module initialstate [mod_widget,mod_window,
    mod_dialog,mod_main_2]
program_2 = extends "DialogWindow" "Window" (extends "Window" "
    Widget" modules_2)
```

Listing A.23: *Co-op* semantics test case (2)

Finally, to demonstrate the handling of instance variables, we show a final definition of operation `main`, in listing A.24. This operation invokes the operations `setTitle` and `setText`, which set the values of the instance variable `@title` (in module `Window`) and `@text` (in module `DialogWindow`) to a (predefined) value. Note that, although the instance variables are defined in separate modules, they are represented as part of the same object of type `DialogWindow`. This is demonstrated by requesting the values of these two variables through their respective accessor methods, and assigning these values to the instance variables `@out1` and `@out2` in module `Main`.

```
|| module Main {
||   var @out1, @out2
||   main() {
||     var window;
||     window = [DialogWindow new];
||     [window setTitle];
||     [window setText];
||     @out1 = [window getTitle];
||     @out2 = [window getText];
|| } }
st_1 = Assignment "window" (NewExpression "DialogWindow")
st_2 = EventStatement (LocalVariable "window") ("setTitle", "")
st_3 = EventStatement (LocalVariable "window") ("setText", "")
st_4 = Assignment "@out1" (EventExpression (LocalVariable "window")
    ("getTitle", ""))
st_5 = Assignment "@out2" (EventExpression (LocalVariable "window")
    ("getText", ""))
op_main_3 = make_operation "main" [] (["window"],[st_1,st_2,st_3,st_4,
    st_5])
mod_main_3 = make_module "Main" ["@out1","@out2", "@out3"] [op_main_3]
modules_3 = foldr add_module initialstate [mod_widget,mod_window,
    mod_dialog,mod_main_3]
program_3 = extends "DialogWindow" "Window" (extends "Window" "
    Widget" modules_3)
```

Listing A.24: *Co-op* semantics test case (3)

The resulting final state can be seen in listing A.25:

```
|| Evaluate: m_program program_3
(Object 0 "null","Stack: [([(\"_main\",Object 3 \"Main\")],Object 0
    \"null\")]; Instancevars: [(4,(\"@title\",StringObject \"A window
     title\")),(4,(\"@text\",StringObject \"A window text\")),(3,(\"
    @out1\",StringObject \"A window title\")),(3,(\"@out2\",
    StringObject \"A window text\"))]; Bindings: [<abstract ob>,<
    abstract ob>,<abstract ob>]")
```

Listing A.25: Simulation of test case 3

In this case, the operation `main` did not set a return value, so its return value is undefined (“null”). The stack again contains the variable `_main`, which is identified by its object id, 3. The final state shows the values of all instance variables to which a value has been assigned: the variable *@title*, part of object 4 (which, since the objects ID's are created sequentially, represents the variable *window* as defined in operation `main`) has the value "A window title". Variable `@text`, also part of the same object, contains value "A window text". These values are also assigned to the instance variables `@out1` and `@out2`, demonstrating that instance variables indeed work as intended.

This concludes the discussion of the semantics of *Co-op*. In this appendix, we have defined the abstract grammar and meaning functions (denotational semantics) of *Co-op*, and shown that the semantics match the language implementation as discussed in chapter 5, by means of simulating several representative examples.

Samenvatting

Er bestaat een grote verscheidenheid aan programmeertalen, die substantieel verschillen in hun ondersteuning van verschillende modularisatie- en compositie-mechanismen. Zulke verschillen kunnen karakteristieke eigenschappen van software, zoals bijvoorbeeld herbruikbaarheid, flexibiliteit, analyseerbaarheid en stabiliteit, in belangrijke mate beïnvloeden. Het ontwerpen van programmertalen omvat vaak het maken van afwegingen tussen zulke karakteristieken.

In dit proefschrift bestuderen we verschillende state-of-the-art programmeertalen, met name in de context van aspect-georiënteerd programmeren. Aspecten zijn voorgesteld als een manier om de modularisatie van software te verbeteren, met name als er zogeheten “crosscutting concerns” voorkomen in het ontwerp van een programma. Crosscutting concerns zijn aspecten die van belang zijn voor het programma, maar andere functionaliteit doorsnijden en daardoor niet makkelijk gemodulariseerd kunnen worden; het doel van de verbeterde modulariteit door het gebruik van aspecten is dat de *onderhoudbaarheid* van software verbetert. In dit proefschrift pogen we verschillende karakteristieken te verbeteren, met name de karakteristieken die beïnvloed worden door het gebruik van aspect-georiënteerde talen.

Om de *analyseerbaarheid* van aspect-georiënteerde programma’s te verbeteren stellen we verschillende analyse-technieken voor. Om te beginnen introduceren we een algoritme dat de declaratieve toepassing van zogeheten “introducties” garandeert. Zulke introducties specificeren wijzigingen aan de structuur van een programma, en kunnen gebruikt worden als een krachtig compositie-mechanisme dat de herbruikbaarheid en flexibiliteit van programma-modules ten goede komt. Helaas kan het gebruik van introducties ook

leiden tot nieuwe soorten compositie-conflicten. Bijvoorbeeld, in sommige gevallen is het niet mogelijk om een eenduidige oplossing te vinden voor de specificaties van introducties door aspecten. We stellen geautomatiseerde tools voor om zulke conflicten te detecteren. Verder gebruik we graaf-gebaseerde formalismen om verschillende andere categorieën van problemen gerelateerd aan aspecten te detecteren. Bijvoorbeeld, aspect-georiënteerde constructies overtreden soms andere taal-regels, of kunnen leiden tot onbedoelde composities. Door relevante delen van aspect-georiënteerde programma's te modeleren met behulp van grafen, kunnen we bestaande analyse tools gebruiken om zulke conflicten te detecteren.

Behalve het verbeteren van de *analyseerbaarheid* van aspect-georiënteerde programma's, proberen we ook de *veranderbaarheid* te verbeteren, gezien vanuit het perspectief van *software engineers*, die programma's schrijven met gebruikmaking van aspecten, maar ook vanuit het perspectief van *language engineers*, die prototypes van aspect-georiënteerde talen of compositie-mechanismen implementeren. Met het oog hierop stellen we een aspect metamodel interpreter framework voor, dat gebruikt kan worden om vele aspect-georiënteerde talen te definiëren, waaronder ook domein-specifieke aspect-talen, waarvan we verschillende voorbeelden geven. Met gebruikmaking van dit framework is het ook mogelijk om composities te maken van aspecten die in verschillende (bijvoorbeeld domein-specifieke) aspect-talen zijn uitgedrukt, zolang die in termen van ons framework zijn beschreven. Dit is mogelijk door de gemeenschappelijke structuur en control flow van aspecten zoals gedefinieerd in ons framework, alsook de aanwezigheid van een constraint-mechanisme dat gebruikt kan worden om (potentiële) conflicten op te lossen.

Uiteindelijk stellen we een krachtige compositie-infrastructuur voor, die de definitie van een grote verscheidenheid aan compositie-mechanismen ondersteunt. Hieronder zijn verschillende soorten inheritance (overerving, zoals gebruikt in object-georiënteerde programmeertalen), maar ook aspect-gebaseerde of domein-specifieke compositie mechanismen. Deze infrastructuur is gedefinieerd in termen van een programmeertaal, die zelf geen vaste compositie-mechanismen bevat, maar een generiek mechanisme ondersteunt dat het toestaat om compositie-mechanismen op te bouwen uit een gemeenschappelijke set van primitieven, die uitgedrukt zijn als first-class objecten. Dit zorgt ervoor dat de taal veel *flexibeler* kan zijn dan bestaande talen die alleen een vaste set van compositie-mechanismen ondersteunen.

Bibliography

[1] Annotation Processing Tool, http://java.sun.com/j2se/1.5.0/docs/guide/apt/.

[2] Hibernate - relational persistence for java and .net http://www.hibernate.org/.

[3] JBoss Application Server, http://www.jboss.org.

[4] JTransformer Framework, http://roots.iai.uni-bonn.de/research/jtransformer/.

[5] Spring framework - http://www.springsource.org/.

[6] The Miranda Programming Language http://miranda.org.uk/.

[7] How to avoid traps and correctly override methods from java.lang.Object, http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html, 1999 (URL verified March 2009).

[8] Java Aspect Metamodel Interpreter - http://jami.sf.net/, 2007.

[9] Co-op homepage, http://wwwhome.cs.utwente.nl/~havingaw/coop/, 2008.

[10] Composestar project - http://composestar.sf.net, January 2009.

[11] A. Abran, P. Bourque, R. Dupuis, and J. Moore. *Guide to the Software Engineering Body of Knowledge-SWEBOK*. IEEE Press Piscataway, NJ, USA, 2001.

[12] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In O. L. Madsen, editor, *Proc. 7th European Conf. Object-Oriented Programming*, pages 372–395. Springer-Verlag Lecture Notes in Computer Science, 1992.

[13] M. Akşit and A. Tripathi. Data abstraction mechanisms in sina/st. In *Proceedings of the conference Object-Oriented Systems, Languages and Applications*, volume 23 of *ACM Sigplan Notices*, pages 267–275, 1988.

[14] AspectJ project - `http://www.eclipse.org/aspectj/`. http://www.eclipse.org/aspectj/.

[15] AspectJ Team. The AspectJ 5 Development Kit Developer's Notebook `http://www.eclipse.org/aspectj/doc/next/adk15notebook/`.

[16] A. Assaf and J. Noyé. Dynamic AspectJ. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, New York, NY, USA, 2008. ACM.

[17] U. Aßmann and A. Ludwig. Aspect weaving by graph rewriting. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, Oct. 1999.

[18] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible aspectj compiler. *Transactions on Aspect-Oriented Software Development I*, 3880/2006:293 – 334, February 2006.

[19] A. H. Bagge and K. T. Kalleberg. DSAL = library+notation: Program transformation for domain-specific aspect languages. In *Proceedings of the Domain-Specific Aspect Languages Workshop*, October 2006.

[20] G. Banavar and G. Lindstrom. An application framework for module composition tools. In *In ECOOP '96, number 1098 in Lecture Notes in Computer Science*, pages 91–113. Springer Verlag, 1996.

[21] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[22] L. Bergmans and M. Akşit. Composing multiple concerns using composition filters. Technical report, University of Twente, The Netherlands, 2001.

[23] L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In Filman et al. [54], pages 63–95.

[24] C. Bockisch. Alia4J - Aspect Language Implementation Architecture for Java, `http://www.alia4j.org`.

[25] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. *ACM SIGPLAN Notices*, 41(10):109–124, 2006.

[26] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, New York, NY, USA, 2007. ACM Press.

[27] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006 (NODe06)*, volume LNI-P-88 of *Lecture Notes in Informatics*, pages 49–64. Springer-Verlag, September 2006.

[28] N. Bouraqadi, A. Seriai, and G. Leblanc. Towards unified aspect-oriented programming. In *Proceedings of ESUG 2005 (13th international smalltalk conference)*, 2005.

[29] G. Bracha and W. Cook. Mixin-based inheritance. In *Conf. Object-Oriented Programming: Systems, Languages, and Applications; European Conf. Object-Oriented Programming*, pages 303–311. ACM, 1990.

[30] M. Bräuer and H. Lochmann. Towards semantic integration of multiple domain-specific languages using ontological foundations. In *Proceedings of 4th International Workshop on (Software) Language Engineering (ATEM'07) co-located with MoDELS 2007*, October 2007. To appear.

[31] J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-specific models and pointcuts using a logic metalanguage. *Computer Languages, Systems and Structures*, Volume 34(Issues 2-3):66–82, July–October 2008 2008.

[32] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In *1st Conf. Generative Programming and Component Engineering*, volume 2487 of *lncs*, pages 110–127, Berlin, 2002. Springer-Verlag.

[33] J. Brichau, M. Mezini, J. Noyé, W. Havinga, L. Bergmans, V. Gasiunas, C. Bockisch, J. Fabry, and T. D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe Deliverable D39, Vrije Universiteit Brussel, 27 February 2006 2006.

[34] S. t. Brinke. Implementing COOL in JAMI. In *Proceedings of the 9th Twente Student Conference, available at* `http://referaat.cs.utwente.nl/new/papers.php?confid=10`, June 2008.

[35] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th international conference on Software engineering, ICSE2006*, pages 242–251. ACM New York, NY, USA, 2006.

[36] A. Bryant, A. Catton, K. De Volder, and G. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 10–18. ACM New York, NY, USA, 2002.

[37] V. Cepa and M. Mezini. Declaring and Enforcing Dependencies Between .NET Custom Attributes. In Karsai and Visser [80], pages 283–297.

[38] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[39] P. Cointe. Reflective languages and metalevel architectures. *ACM Comput. Surv.*, page 151, 1996.

[40] A. Colyer. AspectJ. In Filman et al. [54], pages 123–143.

[41] B. C. d. S. Oliveira. Modular Visitor Components. In *Proceedings of the 23th European Conference on Object-Oriented Programming (ECOOP 2009)*, 2009.

[42] C. Dalager, S. Jorsal, and E. Sort. Aspect oriented programming in JBoss 4. Master's thesis, IT University of Copenhagen, Feb. 2004.

[43] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 60–71. ACM New York, NY, USA, 2008.

[44] M. D'Hondt and T. D'Hondt. Is domain knowledge an aspect? In C. V. Lopes, A. Black, L. Kendall, and L. Bergmans, editors, *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[45] E. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[46] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [96], pages 141–150.

[47] R. Douence and L. Teboul. A pointcut language for control-flow. In Karsai and Visser [80], pages 95–114.

[48] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *Proceedings of ADI'06 Aspects, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Jul 2006.

[49] P. E. A. Durr. *Resource-based Verification for Robust Composition of Aspects*. PhD thesis, University of Twente, Enschede, June 2008.

[50] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, New York, NY, USA, 2008. ACM.

[51] ECMA. Standard ECMA-334 - C# Language Specification `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.

[52] R. E. Filman. What is aspect-oriented programming, revisited. In L. Bergmans, M. Glandrup, J. Brichau, and S. Clarke, editors, *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, June 2001.

[53] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Comm. ACM*, 45(1):116–122, Jan. 2002.

[54] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[55] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[56] F. Forster and F. Steimann. AOP and the Antinomy of the Liar. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *Proceedings of the Fifth workshop on Foundations of Aspect-Oriented Languages (FOAL), at AOSD 2006*, TR #06-01, pages 53–62. Dept. of Computer Science, Iowa State University, March 2006.

[57] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[59] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, 1995.

[60] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[61] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM.

[62] S. Greco, D. Saccà, and C. Zaniolo. DATALOG Queries with Stratified Negation and Choice: from P to DP. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 82–96, London, UK, 1995. Springer-Verlag.

[63] Groove homepage, `http://groove.sf.net`.

[64] J. Guy L. Steele. Growing a language. *Higher Order Symbol. Comput.*, 12(3):221–236, 1999.

[65] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In P. Costanza, G. Kniesel, K. Mehner, E. Pulvermüller, and A. Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, Feb. 2002. Technical report IAI-TR-2002-1.

[66] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[67] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007: Proceedings of the 21th European Conference on Object-Oriented Programming*, pages 501–524, 2007.

[68] W. Havinga. Groove model generator for the detection of conflicts, `http://trese.cs.utwente.nl/~havingaw/groove_conflict_detection`.

[69] W. Havinga. Designating join points in composestar - a predicate-based superimposition selector language for compose*. Master's thesis, May 2005.

[70] W. Havinga, I. Nagy, and L. Bergmans. Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions. In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, Sept. 2005.

[71] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and Resolving Ambiguities caused by Inter-dependent Introductions. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 214–225, New York, March 2006. ACM.

[72] W. K. Havinga, L. M. J. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Paphos, Cyprus*, volume 5142/2008 of *Lecture Notes in Computer Science*, pages 180–206, Berlin, 2008. Springer Verlag.

[73] W. K. Havinga, I. Nagy, L. M. J. Bergmans, and M. A. sit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In O. de Moor, editor, *Proceedings of International Conference on Aspect Oriented Software Development, AOSD 2007, Vancouver, Canada*, ACM International Conference Proceedings Series, pages 85–95, New York, March 2007. ACM Press.

[74] W. K. Havinga, T. Staijen, A. Rensink, L. M. J. Bergmans, and K. G. van den Berg. An abstract metamodel for aspect languages. In J. Brichau, S. Chiba, D. H. Lorenz, E. Tanter, and K. D. Volder, editors, *Open and Dynamic Aspect Languages, Bonn, Germany*, page 9. Aspect-Oriented Software Association, March 2006.

[75] W. K. Havinga, T. Staijen, A. Rensink, L. M. J. Bergmans, and K. G. van den Berg. An abstract metamodel for aspect languages. Technical report, Enschede, May 2006.

[76] R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Net.Object Days 2002*, Oct. 2002.

[77] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 162–171, Los Alamitos, 2000. IEEE Computer Society.

[78] ISO/IEC. ISO 9126: Software Engineering - Software Product Quality - Part 1: Quality Model. 2001.

[79] Java 2 Platform Standard ed. API Documentation, `http://java.sun.com/j2se/1.5.0/docs/api/`.

[80] G. Karsai and E. Visser, editors. *Proc. Generative Programming and Component Engineering: Third International Conference*, volume 3286 of *Springer-Verlag Lecture Notes in Computer Science*. Springer, Oct. 2004.

[81] E. Katz, S. Katz, and W. H. et al. Detecting Interference Among Aspects. Technical Report AOSD-Europe Deliverable D116, Technion Israel Institute of Technology, Haifa, Israel, February 2007.

[82] S. Katz. Diagnosis of harmful aspects using regression verification. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, Mar. 2004.

[83] B. Kessler and É. Tanter. Analyzing interactions of structural aspects. In *Workshop on Aspects, Dependencies and Interactions @ECOOP 2006*, July 2006.

[84] G. Kiczales. It's not metaprogramming. *Software Development Magazine*, (10), 2004.

[85] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[86] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[87] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586/2005 of *Lecture Notes in Computer Science*, pages 195–213. Springer-Verlag, 2005.

[88] G. Kniesel and U. Bardey. An Analysis of the Correctness and Completeness of Aspect Weaving. In *Proceedings of IEEE Working Conference on Reverse Engineering (WCRE 2006)*. IEEE Computer Society, 2006.

[89] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.

[90] G. Kniesel and T. Rho. Generic aspect languages - needs, options and challenges. In L. Seinturier, editor, *2nd French Workshop on Aspect-Oriented Software Development (JFDLPA 2005)*, Sept. 2005.

[91] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects. In W. Cazzola, S. Chiba, and G. Saake, editors, *ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE)*, pages 111–126, June 2004.

[92] S. Kojarski and D. H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. *SIGPLAN Notices*, 42(10):515–534, 2007.

[93] S. Kojarski and D. H. Lorenz. Identifying feature interactions in multi-language aspect-oriented frameworks. pages 147–157, Minneapolis, MN, May 20-26 2007. IEEE Computer Society.

[94] R. Laddad. AOP@Work: AOP and metadata: A perfect match, Part 1—Concepts and constructs of metadata-fortified AOP. Technical report, IBM Developer Works, Mar. 2005.

[95] R. Laddad. AOP@Work: AOP and metadata: A perfect match, Part 2—Multidimensional interfaces with metadata. Technical report, IBM Developer Works, Apr. 2005.

[96] K. Lieberherr, editor. *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, Mar. 2004.

[97] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[98] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[99] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM.

[100] D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *ECOOP 2007 Second International Workshop on Aspects, Dependencies and Interactions*, 2007.

[101] O. L. Madsen, B. Mø-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[102] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *proceedings of APLAS'03 - the First Asian Symposium on Programming Languages and Systems*, volume 2895/2003 of *Lecture Notes in Computer Science*, pages 105–121. Springer-Verlag, 2003.

[103] H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743 of *lncs*, pages 2–28, Berlin, July 2003. Springer-Verlag.

[104] K. Mehner, M. Monga, and G. Taentzer. Interaction analysis in aspect-oriented models. In *14th IEEE International Conference Requirements Engineering*, pages 69–78, 2006.

[105] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, June 2006.

[106] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[107] I. Nagy, L. Bergmans, W. Havinga, and M. Aksit. Utilizing design information in aspect-oriented programming. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[108] M. E. Nordberg III. Aspect-oriented dependency management. In Filman et al. [54], pages 557–584.

[109] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding implicit invocation to languages: Three approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, London, UK, 1993. Springer-Verlag.

[110] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proc. OOPSLA '01 Conf. Object Oriented Programming Systems Languages and Applications*, pages 283–299. ACM Press, 2001.

[111] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.

[112] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, Dec. 1972.

[113] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, 2005. ACM Press.

[114] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[115] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FSE)*, pages 147–158. ACM, Oct. 2004.

[116] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In Lieberherr [96], pages 16–25.

[117] M. Shonle, K. Lieberherr, and A. Shah. XAspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37, New York, NY, USA, 2003. ACM Press.

[118] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[119] Sun Microsystems. Java 1.5 documentation- `http://java.sun.com/j2se/1.5.0/docs/`.

[120] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *lncs*, pages 98–113, Vienna, Austria, Mar. 2006. Springer-Verlag.

[121] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, Sept. 2005.

[122] D. Thomas. Message oriented programming. *Journal of Object Technology*, 3(5):7–12, 2004.

[123] S. Thompson. Programming language semantics using miranda. 1995.

[124] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional programming languages and computer architecture*, volume 201, pages 1–16. Springer, 1985.

[125] K. G. van den Berg, J. M. Conejero, and R. Chitchyan. AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation. Technical Report AOSD-Europe-UT-01 D9, AOSD-Europe, Enschede, May 2005.

[126] W. van Dijk and J. Mordhorst. CFIST, Composition Filters in Smalltalk. Graduation Report, HIO Enschede, The Netherlands, May 1995.

[127] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999.

[128] N. Wirth. Programming in MODULA-2 (3rd corrected ed.). *Springer Texts And Monographs In Computer Science*, page 202, 1985.