Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure.

Kris Gybels

August 27, 2001

Document version notes

When refering to parts of this document, please use section numbers rather than page numbers.

I have made several versions of this dissertation available. Apart from this note they all have the same content as the version I submitted to the faculty (with the exception of some minor corrections). The difference is in their layout only. The different versions are listed below:

Standard version: uses a one-sided layout suitable for both printing and screen reading. Available as a PostScript file and as a PDF file with hyperlinks.

Double sided version: uses a layout suited for two-sided printing which can be annoying for screen reading due to the flipping of margins every page but looks beautiful if you want to save some paper by printing two pages per sheet.

Tree friendly version: a version with 2 document pages per printable page. Uses a slightly larger font to avoid too much eye straining due to the scaling of the pages and smaller margins to make it look better and save some more paper.

HTML version: for the true environmentalist!

The different versions are available online at:
`http://wendy.vub.ac.be/~kgijbels`

Acknowledgements

I thank my promotor, Prof. Dr. Theo D'Hondt, for his belief that something interesting might actually emerge from my mind, and for not getting too mad at me when I had barely anything to show for at the end of the first exam session. Words of gratitude also go to Kris De Volder for proof reading and providing me with some valuable last minute hints to improve my writing. A big "thank you" also goes to Wolfgang De Meuter for his comments on one of the drafts of this dissertation. Kim Mens, Roel Wuyts and Tom Tourwé also helped with proof reading parts of this document. On a more general note, I thank my parents for having given me the opportunity to get a higher education of which this work presents the finalization. I also thank my brother for lending me one of his computers for such a long time to replace my old one so I could work more comfortably.

Last, but most importantly, many many thanks go out to Johan Brichau who allowed me to waste countless hours of his time with requests for proof reading and discussions. Without his guidance and continuing encouragement, I would have given up hope of ever finishing before even the first letter was typed in.

Contents

Chapter 1

Introduction

In this dissertation we discuss the benefits and feasibility of using a logic meta programming language making use of a dynamic joinpoint structure to express the cross-cutting of concerns in order to achieve better separation of concerns. Our thesis statement can be broken into three core parts: the use of a logic language, the use of a meta language and the use of dynamic joinpoints. These core parts have been highlighted further in this introduction.

Maintaining large software systems often proves to be problematic. When requirements for the system change a maintenance team is set on it to make the necessary changes. To do so they first have to figure out where in the system's program the requirements are addressed. To make this job easier all important requirements in a program should be addressed clearly and separately from other requirements. This is an important principle in software engineering, known as separation of concerns [31,21].

The currently adopted way to achieve separation of concerns is modular decomposition. The goal is to decompose a system into smaller, relatively independent modules. When decomposed well, each module will handle a subset of the initial requirements and each requirement will be handled by only one module. The process can be repeated until a level is reached where each module handles a single requirement or just part of a requirement. Conversely, the actual software can be constructed by implementing all of the modules of the lowest level and composing them to form the final product.

In current software development practice, modular decomposition is focused on decomposing a system along functional requirements. The modules used are procedures, methods, objects, classes, libraries, packages etc. Other requirements must also be decomposed using the same constructs, which often proves to be difficult. Furthermore, the different decompositions of a system along different requirements may conflict. The result is that it often proves to be impossible to cleanly separate all requirements and some of them tend to spread over several modules and crossing the levels of the hierarchical structure. Such requirements are called *cross-cutting* concerns. In terms of code this means that for example a single method addresses several requirements. In other words, such code is an intertwining of requirements, clearly violating the principle of separation of concerns.

Classical solutions to the intertwining problem suggest that since not all concerns are separated, there were simply not enough levels yet in the hierarchical decomposition. The design patterns introduced by Gamma et al. [27] for example are focused on bringing more separation by introducing extra classes, methods etc in a program. One problem with this approach is that it leads to high fragmentation. Clearly, hierarchical decomposition can be taken too far in that the units become so small that they only address a tiny part of a requirement and seemingly have no functionality at all. The classical approaches also still assume that components actually intended to capture functionality are suited to handle all requirements, not just the functional ones.

Concerns that do not fit well in functional modules have been dubbed aspects and aspectual decomposition research is intent on finding new decomposition and composition mechanisms to handle these aspects. Some tendency towards aspectual decomposition can be observed in many design techniques and languages, including the popular UML [20].

In order for aspectual decomposition to be effective at separating cross-cutting concerns, it should not be limited to just the design phase of software construction. New programming languages or extensions to existing ones are required. Several research efforts have been undertaken towards this goal. In Subject-Oriented Programming one composes a single class from different subjects. Composition Filters can be used to extend or replace an object's behavior without manually modification of the class of the object. The technique we concentrate on mostly in this dissertation is called Aspect-Oriented Programming.

Aspect-Oriented Programming is based on the idea that aspects can be separated by taking them out of the functional modular decomposition and describing how the aspect cross-cuts the modular decomposition. The points in the decomposition the aspect cross-cuts are called the joinpoints and the language used to describe the joinpoints is called the pointcut language. Combining the modular decomposition and the aspects is done by an aspect weaver, a special type of compiler or interpreter.

Aspect-Oriented Programming poses the problem of what language to use as the pointcut language. We believe that most existing examples of AOP systems did not use an expressive enough pointcut language and that preferably a Turing complete language is used. Since the goal of the language is to describe, a declarative logic based language seems most suited. Since such languages cannot just be used to describe, but also reason about what they describe we get a very powerful mechanism for describing cross-cutting. Part of the goal of this dissertation is to show the applicability of using *a Turing complete logic language* for the description of cross-cutting.

Another problem posed in Aspect-Oriented programming is what to use as joinpoints. The question is whether joinpoints should be elements directly extracted from the modular decomposition. In other words: should cross-cutting be described in terms of the modules such as methods, classes, packages etc. an aspect cross-cuts? An alternative is to extract a structure from the modular program which is implicitly present therein, such as an execution graph or a data flow graph. Such an alternative joinpoint structure can make the description of cross-cutting clearer as aspects have been found to defy functional modular decomposition exactly because they more closely follow structures such as call graphs rather than classical modular structures such as class hierarchies.

A question related to the problem of what to use as joinpoints is whether the joinpoints should be static or dynamic. Static joinpoints are related to program source code, while dynamic joinpoints require execution of a program.

So far different AOP approaches have mostly used static joinpoints. Dynamic joinpoints are avoided because they may lead to an inefficient weaver. We support the thesis that dynamic joinpoints can be useful in describing cross-cutting concerns. Rather than dismissing them as inefficient, the weaver should use *optimization techniques* for the pointcut language to make the use of *dynamic joinpoints* feasible.

A problem that occurs when extracting a joinpoint structure from a modular decomposition lies in the nature of the language used to express that decomposition. When using very open languages such as Smalltalk this is made difficult. The difficulty arises because of the flexibility of Smalltalk and other such languages. Many constructs that are language primitives in other languages are replaced by the use of programming conventions. The recognition of such programming conventions can in principle be built into the weaver. However, programming conventions are open to change or personal preference, so the recognition of these conventions should be open to change as well. We therefore express such recognition in the pointcut language, so that it can be easily adapted by any programmer. This requires the use of a *meta programming language* as the pointcut language. A meta language is a language used to reason about programs.

To support the thesis, we have constructed a small AOP system on top of an existing logic meta language system intended to reason about object oriented programs. We have added the capability to express cross-cutting and mechanisms for composing aspects with the object oriented program. Our approach has been heavily based on an existing AOP system, called AspectJ [35, 34]. The AspectJ pointcut language is not a Turing complete language, so we can sometimes make a comparison.

1.1 Document structure

This dissertation is structured as follows:

In the next chapter we discuss Aspect-Oriented software development. We discuss how cross-cutting comes about and what problems it leads to. Existing approaches to deal with cross-cutting concerns will be described, with the emphasis being on Aspect-Oriented Programming. Especially the AspectJ system will receive attention. At the end of the chapter the use of pointcut languages in the different AOP approaches will be discussed.

In chapter three logic meta programming languages are introduced. The general concepts involved in meta programming are explained. We then introduce two LMP languages, TyRuBa and SOUL. At the end of the chapter we discuss an application of TyRuBa to AOP.

Chapter four revolves around the use of a logic pointcut language, embedded in SOUL. The language is based on the pointcut language used in AspectJ, introduced in chapter two. The basic constructs of the language are presented and it is explained how these are used to describe cross-cutting. We then show an application of SOUL's meta programming facilities for the recognition of programming conventions, so as to extend the primitive pointcut language.

The fifth chapter presents Andrew, our AOP system for Smalltalk. Andrew allows for the implementation of aspects, using the pointcut language presented in chapter four to express cross-cutting.

Chapter six discusses the implementation of the Andrew weaver. We present a simple optimization technique used to cope with the dynamics of the pointcut language.

Chapter seven presents some evaluation criteria for pointcut languages, in how well they allow aspects to be separated, potentially making their implementation reusable across applications.

Chapter eight presents an experiment.

The final chapter presents our conclusions and identifies areas for future work.

Chapter 2

Aspect-Oriented Software Development

In this chapter a particular problem observed in today's software development practices is explained. The principle of separation of concerns is introduced and it will be shown how cross-cutting concerns make following this principle difficult. Cross-cutting concerns often wind up spread around a program's code. We explore the nature of these cross-cutting concerns. Several ways of dealing with the cross-cutting problem will be described. One of these is Aspect-Oriented Programming. With Aspect-Oriented programming the cross-cutting problem is solved by explicitly expressing how a concern cross-cuts a program. At the end of the chapter we will make some comments on the languages used to express cross-cutting. In later chapters we will introduce the use of a logic programming language to express and reason about cross-cutting.

2.1 Separation of concerns

The construction of a large software system is generally split into two phases: the design phase and the implementation phase. In the design phase, the large complex software system is split into smaller, less complex parts. In the implementation phase, the reverse is done: the small parts are written and then composed to produce the overall system. The idea is to "Divide and Conquer" [22].

The above description of software construction is very general and one question it brings to mind is: what does it mean for parts to be less complex? To address this question we must know how this complexity in large software systems arises in the first place. We follow the discussion of Bass, Clements and Kazman [11] who relate this complexity to an overload in requirements. They state that a large software system has many requirements posed on it: it must achieve a lot of functionality and on top of that it must do it fast, make efficient use of existing hardware etc. These requirements are also called the concerns[1] of the software engineer. Complexity arises because there are so many concerns to be dealt with and several of these may conflict.

[1]Concern can be seen as a generalization of terms such as requirements, goal, feature, concept, purpose etc. of a software system. [19]

We can now state that a part of a decomposition is less complex when it deals with less concerns. Furthermore, none of the parts should deal with a concern that is already dealt with in another part. This principle is known as *Separation of Concerns* and is generally attributed to either Parnas [46] or Dijkstra [21].

A design process provides software engineers with conceptual techniques for decomposing a system so as to achieve separation of concerns. It is often believed that a design process should result in just one decomposition of a system. Bass et al. have the following two remarks on this:

1. There is no single "right" decomposition of a software system.

2. No decomposition can ever achieve full separation of *all* the concerns.

They therefore propose that software engineers make many different decompositions of a system, in effect providing them with several viewpoints on the software system to be constructed. Concerns that are not well separated in one decomposition can be separated in another decomposition, thus increasing the software engineers understanding of the concerns.

They also describe that decompositions decompose a system into units that are related along some structure, called the *decomposition structure* . We cite two examples of such structures to give the reader some insight: the calls structure and the control flow structure. In the calls structure the units are procedures which are related through a calls or invokes relation. In the control flow structure the units are system states which are related through a becomes-active-after relationship.

Bass et. al. are not the only ones to encourage the viewpoints approach to software engineering. The popular design language UML for example employs a very similar idea. The UML provides several types of diagrams which are intended to capture different properties of a software system. Many of the diagram types can easily be related to the decomposition structures described above. A similar observation is made by Kiczales et al. for the OMT, a predecessor of UML [36].

Having discussed the design phase of software construction in some detail and the emerging viewpoints approach, we can now turn to the implementation phase. In this phase, a program is written in some programming language. A programming language provides mechanisms to define abstractions of system sub-units, and then compose those abstractions in different ways to produce the overall system. In keeping with the earlier terminology, we can define this mechanism as the *composition structure* of the programming language. Software construction is eased when the units employed in the programming language align well with the units employed in the design phase.

Researchers at Xerox PARC have observed [36] some problems in achieving separation of concerns in modern programming languages. To relate their remarks to those of Bass et. al. we formulate them as follows:

1. Current programming languages employ only one composition structure.

2. The composition structure employed by current languages composes a system out of units related to function or behavior.

These observations are in direct conflict with those of Bass et. al. This means that current programming languages cannot separate concerns well. To identify this problem, the PARC researchers defined the terms cross-cutting and code-tangling:

1. Cross-cutting happens when a concern that is best separated in a specific decomposition structure is to be implemented in a composition structure that does not align well with that decomposition structure. The two structures are said to cross-cut. The concern is called a cross-cutting concern with respect to the composition structure.
2. Code tangling is a symptom of cross-cutting at the source code level. Code tangling means that pieces of code dealing with different concerns are intermingled. Code tangling thus violates the separation of concerns principle.

The above discussion is fairly general and we expect the reader to wonder what it all means. Why is it so bad to not abide by the separation of concerns principle? What does tangled code look like? How does cross-cutting happen? In the next section we will employ a more example-based approach to introducing these concepts. We will concentrate on code tangling and how it causes problems.

2.2 Code tangling

In this section we consider the example of a banking application. We consider a decomposition of this system using an object oriented style. In the OO style, the goal is to decompose the system into units that are entities in the problem domain of the software system to be constructed. In our banking example this problem domain consists of clients, accounts, safes, loans, stocks and bonds etc. The units are composed along a “cooperates with” relationship: they send each other messages to achieve some common goal. Thus functional requirements for the banking application are modeled as messages between the units. Some of these requirements are “clients must be able to withdraw money from their account” and “money must be transferable from one account to another”. Implementing this decomposition is fairly straightforward as there are numerous object oriented programming languages around which provide constructs that directly support the OO notion of units (objects) and the way they are composed through message sends.

A banking application typically has several non-functional requirements associated with it. One such requirement is that the system should be able to process several transactions simultaneously, unless they conflict because they for example withdraw money from the same account. Another one is that the system should be secure and should not allow clients to withdraw money from accounts they are not authorized to access. Yet another is that a log of all withdrawals and deposits must be kept. As a last example, banks are highly concerned with consistency and their software must ensure that for example transferring money between accounts should keep the total amount of money in the bank constant.

Problems arise when we try to implement the non-functional requirements in the OO version of the banking application. Concurrent processing for example is done by using threads or processes. Code to create and start such threads is easily added to our application. However the process structure of the application naturally cross-cuts the object structure: no object can be seen as belonging to a single thread of execution and every thread is a sequence of method executions in different objects. The problems this may cause are well known: a single object might be accessed by two threads simultaneously leading to erroneous state updates in the object. To alleviate this problem, objects must be protected with semaphores from which processes must acquire locks before they can access the objects.

Now that we have an idea of the problem that arises because of the cross-cutting between the object structure and process structure and its solution, we must still find a way to implement this solution. The solution can be implemented in a straightforward fashion: simply add an instance variable `semaphore` to every object. Since the process structure accesses an object by executing a method in it, the locking and unlocking of the semaphore can be implemented as extra snippets of code at the beginning and start of methods.

Implementing the other non-functional requirements can be done similarly to the concurrent processing requirement. Pieces of code to deal with security checks, consistency checks and logging checks can be inserted at the necessary places in the source code.

The method of software construction we have outlined above is very common. Carver describes it as the minimal subsets method [19]. Software construction begins by implementing a system that meets a minimal subset of the original requirements. The subset is minimal in the sense that it is the absolute minimum set of requirements that the system should meet to be useful. Once that system has been implemented, the other requirements are addressed by incrementally adding them to the system's code. When the system is not too large, a typical division between the minimal subset and the incremental subset is done on the basis of functional and non-functional requirements, as described here. For larger projects the minimal subset is a subset of the functional requirements. Either way, the minimal subset is always related to the functional requirements as we would not consider a system implementing only non-functional requirements to be useful.

The minimal subset method can be described in steps as follows:

1. Temporarily scratch some requirements from the total set of requirements on the system.

2. Decompose the system along the remaining requirements.

3. Implement the constructed decomposition.

4. Add the remaining requirements one by one to the existing implementation.

It seems that the minimal subset method does well at achieving separation of concerns. In the first step concerns are separated by simply throwing away some of them. In the last step these concerns are still separated because they are handled one-by-one. In between, the concerns of the minimal subset are separated in an appropriate decomposition.

While the minimal subset method achieves separation of concerns in the development process itself, this is not at all reflected in the final code of the system. The problem lies in the last step: current programming languages concentrate on composing systems out of units related to function or behavior, but have little support for expressing additions that must be made to this behavior in order to handle more concerns. This is further made problematic by the fact that several of the concerns not included in the minimal subset may naturally cross-cut the decomposition based on the minimal subset requirements.

In practice, the last step of the minimal subset method can be described more as "patching up code" than as "adding the handling of more concerns". In an OO program this means the adding of extra fields to objects, adding extra methods, injecting small code snippets at some points in the code etc. This leads to code tangling which entails several problems.

To illustrate the problems involved in code tangling we show the code of one particular class in our banking application, the `Account` class. Figure 2.1 shows the code of the class before patching occurred. Figure 2.2 shows the same code, but patched up to handle logging and concurrency. In the last figure we have employed a visual aid which is often found to be helpful in identifying code tangling: the injected code has been colored so as to reflect which concern it applies to. The colors in the figure are mixed or entangled, which is the origin of the term code tangling.

Tangled code clearly violates the separation of concerns principle. This affects several quality attributes of the code, also known as the "ilities" [45, 23], some of these are:

Comprehensibility: reading the code from top to bottom requires our mind to continuously switch between different concerns, something the human mind is known to be not very good at. Focusing in on a single concern and comprehending how that concern is handled is difficult because the code for it is mixed with that of other concerns.

Reusability: to be able to reuse the code pertaining to a specific concern we must remove the unwanted concerns it is tangled with. To reuse the `Account` class in an application requiring concurrency but not logging, we must first remove the logging code. To reuse the `Account` class in a simple board game would likely require removing both logging and synchronization. This requires meticulous inspection of the code to figure out what is required and what is not. Reusability of code specific to concerns and even concerns at the conceptual level is further discussed in [48].

Evolvability or adaptability: requirements for a software system are likely to evolve after it has been constructed. This means that the way the concerns are handled in the code must be adapted. This requires tracing the handling of a single concern which might be spread throughout the entire source code because of cross-cutting.

Another way to look at code tangling and cross-cutting is to use *Class Responsibility Collaboration* analysis [54]. This is a simple analysis technique used in class based OO programming which involves writing the responsibilities of each class on a paper card. Small cards are used to reflect that objects of each class should not be given too many responsibilities. For the simple version of

```
withdraw: amount
    balance := balance - amount
deposit: amount
    balance := balance + amount
```

Figure 2.1: Methods for a simple Account class

```
initialize
   monitor ← Semaphore forMutualExclusion.
withdraw: amount
   Transcript show: 'withdrawing ' , amount asString; cr.
   monitor critical: [
     balance ← balance - amount ]
   Transcript show: 'withdrawn ' , amount asString; cr.
deposit: amount
   Transcript show: 'depositing ' , amount asString; cr.
   monitor critical: [
     balance ← balance + amount ]
   Transcript show: 'deposited ' , amount asString; cr.
balance
   ↑ monitor critical: [
    balance ]
```

Figure 2.2: Methods of the Account class patched to also handle logging and synchronization

the `Account` class this would be "manage a certain amount of money". In the patched version it has also been given the responsibility for "doing synchronization" and "doing logging". Adding more concerns would mean adding more responsibilities. Furthermore, we will find that the synchronization and logging responsibilities will occur on many more cards, for the synchronization concern probably on *all* the cards.

Because we have used only a single class in our example, code tangling caused by cross-cutting concerns may not seem like that much of a problem. A larger case study unfortunately falls out of the scope of this work. A more in-depth study of analyzing code tangling in an existing program is given by Carver [19]. A study of constructing an application involving several cross-cutting concerns using some of the technology further described in this chapter was done in the construction of Atlas [33]. Böllert did a similar case study but concentrated on the synchronization and logging concerns [13].

2.3 Aspects

2.3.1 Modular (de)composition

In the first section we discussed that Xerox PARC researchers have observed that current programming languages provide a single composition structure, based on expressing units of function or behavior. They find that the abstraction and composition mechanisms in these languages have a common root in the idea of a procedure. They therefore refer to these languages as *generalized procedural languages* (GP languages) [36]. In this respect, functions, objects, modules etc. are seen as procedures because they are composed in a calls-upon or sends-message-to structure and each generalized procedure encapsulates a functional unit of the overall system. We will use the term *module* rather than procedure to refer to the units of composition provided by a GP language, and the term *modular composition* to refer to its composition structure.

A characteristic of the modular composition provided by GP languages is that it is hierarchical. Ever bigger units are built out of smaller units. In a procedural language, bigger procedures are built out of smaller procedures. In a class based OO language, classes are made out of methods, classes are in turn the constituents of libraries or frameworks etc. This is based on the inverse decomposition technique of dividing a goal into subgoals, which can in turn be divided into smaller subgoals.

We find that another way of describing GP languages and their composition mechanisms is that they are based on cooperation. One module explicitly calls upon another to achieve some goal. We will therefore call the composition mechanism provided by GP languages *cooperative composition*.

2.3.2 Definition of aspects

We have defined the cross-cutting of a concern as relative to a specific composition structure in the first section of this chapter. However, since only one composition structure is provided by most programming languages, we might as well talk about cross-cutting as an absolute property of a concern. The following definition is based on the definition originally given by Kiczales et al. [36]:

> A concern that cross-cuts a composition structure such as provided by a generalized procedural language is an *aspect*.

The intent of *Aspect-Oriented Programming* is to provide new abstraction and composition mechanisms which do not follow the GP model so as to improve separation of aspects. This is a new research area and several propositions have been made so far. Some of them will be discussed later in this chapter.

Some examples of common aspects were given in the previous section: synchronization, logging and security. Other commonly given examples are: failure handling, persistency, debugging support, context-sensitive behavior etc.

From the list of example aspects we can deduce that what are traditionally seen as non-functional requirements are often likely candidates to be aspects. Before the term aspect was invented, Hürsch and Lopes [31] spoke of *basic concern* and *special purpose concern*. The definition for the distinction fits well with the intuitive notion software engineers have of functional and non-functional requirements. It can also be seen as fitting with the minimal subset method, which in turn is mostly based on the idea of functional and non-functional requirements. The definition of Hürsch and Lopes is given below:

> “The *basic concern*, from which we try to separate all other concerns, captures the essence of the computation as it is relevant to the application domain. The basic concern provides functionality

in a traditional way and does not depend on other aspects of the computation such as distribution, persistence, failures, and so forth. The basic concern species what is really important for an application program.

A *special purpose concern* is any other form of computation used to manage or optimize the basic concern. In a sense, special purpose concerns provide support for the basic concern and therefore play an auxiliary role. Special concerns can also be essential for the performance of the overall computation."

We find we can also formulate the definition of Hürsch and Lopes as follows: a special purpose concern makes a contribution to the basic computation. To handle such concerns we need to have a *contributive composition* mechanism. We define contributive composition as the ability of a module to influence the semantics of another module, or in other words the ability of a model to insert behavior internally into another module.

2.4 Handling aspects without contributive composition

In this section we first take a look at existing mechanisms in OO languages which can be used to achieve better separation of concerns in the face of cross-cutting concerns.

2.4.1 Using inheritance

Inheritance in object oriented programming provides a form of contributive composition. A subclass can extend the functionality of it's parent class by providing new methods or overriding methods. It is fairly straightforward in our account example to put the added functionalities in subclasses of the account class. We can implement two classes based on the account class that handle logging and synchronization: `AccountWithLogging` and `AccountWithLoggingAndSynchronization`. The first simply overrides the methods from the `Account` class to do logging and then does a super send. The other class further subclasses the `AccountWithLogging` class and wraps synchronization code around a super call to the overridden methods.

Inheritance is however far too limited to handle aspects. First of all, the idea of inheritance is not truly based on contribution but rather on specialization. Secondly, inheritance is available only on the level of classes and is not enough to handle the system wide cross-cutting property of aspects. In the banking application example, the concurrent processing requirement and logging will affect more than just the account class. Handling this through inheritance would mean creating a specialized `-WithLoggingAndSynchronization` class of every class in the system. This would lead to a lot of code duplication.

2.4.2 Design patterns

Many of the design patterns introduced by Gamma [27] can be described as object oriented idioms for handling aspects. The general intent is to make classes more reusable by factoring requirements over different classes which are then related to each other in ways specified by the pattern. In other words: design patterns were crafted with separation of concerns in mind.

As a simple example, the security policy of our banking application could be factored out into a strategy object. This modularizes the security policy into a component of it's own, making it easier to adapt or even reuse.

Design patterns suffer from several problems however. Design patterns are design elements only, once implemented it is difficult to trace which design patterns were used where. Design patterns also tend to introduce a lot more complexity. An extreme case is that some design patterns essentially introduce "object schizophrenia" by splitting a class into more classes. This is exactly what *Subject-Oriented Programming* (SOP), to be described next, tries to avoid. An analysis of other problems with design patterns and the SOP solutions is given on the SOP website [4].

Do design patterns propose a model of cross-cutting contributive composition? The answer to this question is two fold. In the security policy example, the strategy pattern succeeds at separating the security policy from the remainder of the code, but only partially. It still requires method calls to be done at certain points in the rest of the code. This does reduce code tangling to single message sends. However, if security must be tightened this is likely to require security checks to be done at more points, which would mean manually adding more of these message sends. While many design patterns help at separating concerns, they do so through a model of *invited contribution.* This is also referred to as the "preplanning problem" [4] of design patterns. Design patterns only allow for adaptation of a module if the mechanisms of the design pattern are present in the module, such as the calls to the strategy object or the accept-methods associated with the Visitor pattern. In other words: many design patterns are about providing hooks or points of adaptation in a component.

2.5 Requirements for contributive composition

In the previous sections we rather informally discussed the various properties of some of the widely adopted composition mechanisms and how these are insufficient for capturing aspects well. From this discussion we can extract a set of desirable properties for contributive composition mechanisms intended to handle aspects:

Separation of concerns: the aspect must be representable in a module separate from the modules it contributes to. This rules out direct code patching.

Unrequested contribution: aspects must be able to contribute to modules without the implementers of these modules having to make special arrangements to allow for the contribution.

No manual adaptation: no manual adaptation of a module should be necessary to allow for the contribution.

Cross-cutting contribution: the composition mechanism must allow a single aspect to contribute to several modules at once. These modules do not necessarily need to be at the same level or within the same parent module in a traditional hierarchical decomposition.

2.6 Existing contributive composition mechanisms

In this section we will take a look at some existing contributive composition mechanisms which extend object oriented programming. We introduce Subject-Oriented Programming and Composition Filters.

2.6.1 Subject-Oriented Programming

Subject-Oriented Programming [3] grew out of the idea that several perspectives can be taken on with respect to a concept from the problem domain of an application. A book publisher for example has two departments: the marketing and manufacturing department. Each of these are interested in different properties of a book. The marketing department needs to know the title, author, price, subject etc. of the book, while the manufacturing department is interested in the type of binding and paper to use for the book. It would be desirable for a single book to be represented as a single entity in the publisher's software systems. This can get complicated however when all of the subsystems related to one department are developed by different development teams. Subject-Oriented Programming allows the different teams to develop different views of a book. When the larger system is composed of the subsystems, the views are merged through the use of correspondence and composition rules.

Subject-Oriented Programming is a form of contributive composition as each of the perspectives contributes to the modeling of a single concept. The idea is somewhat similar to inheritance, but the composition is horizontal rather than vertical as in inheritance. For our banking application we can define logging and synchronization as perspectives on accounts. A worked out example for a `Stack` class is given by Czarnecki [20].

Subject-Oriented Programming meets the requirements set out in the previous section, though it was not really designed with cross-cutting in mind. SOP has however been extended to *Hyperspaces* [45] which does seem to handle this. Discussing Hyperspaces falls out of the scope of this work however.

2.6.2 Composition Filters

Composition Filters (CF) extends Object-Oriented Programming with message transformation specifications [8, 7, 37]. In Composition Filters an object consists of an inner object and an interface layer. The inner object is a conventional object as in traditional OO programming languages, consisting of instance variables and methods. The interface layer encloses the inner object and can enhance or modify its behavior through the manipulation of incoming and outgoing messages. These enhancements are specified through the use of filters. It is possible for a filter to reject messages, change them into another message or buffer them. Filters can also change the target object of a message. The action taken by a filter depends on its type. Several standard types of filters are provided: wait filters for buffering messages, error filters for throwing exceptions etc. Whether a filter is activated for a particular message depends on whether the message matches a particular pattern specified by the filter and conditions on the current state of the inner object.

Composition Filters have for example been used to implement synchronization and logging. A worked out example is again given by Czarnecki [20].

The problem with Composition Filters with respect to our criteria for contributive composition is the same as that with SOP: there is no support for cross-cutting. Filters always relate to a single class of objects, this is obvious from the association of a filter with a layer around an object. However, recently the CF model has also been extended to allow for more general cross-cutting through the use of superimpositions [12]. We do not discuss this further.

2.7 Aspect-Oriented Programming

This section is devoted to discussing some approaches to contributive composition which are being developed or have been developed in research groups today. All of these approaches are intended to abide the desirable properties listed previously. All of these approaches have some common ideas and terminology and they are collectively known as Aspect-Oriented Programming, or AOP for short. We will first describe the common ideas.

Central to any AOP approach is the idea of a weaver. The weaver is responsible for combining the semantical contributions of an aspect with the semantics of the modules it affects. To do so, a weaver requires an aspect to describe two things: the implementation of the extra functionality it adds and the modules it adds this to. This latter part of the aspect description is stated in the form of a description of hooks in the modules. This does not violate the unrequested contribution property as these hooks are provided by the weaver, not the implementors of the modules. In AOP terminology the hooks are called *joinpoints*. We further define:

- *Joinpoints* are the hooks in modules, provided by a weaver, to which the aspect can add functionality or of which it can change the existing semantics in another way.
- The language used to describe joinpoints in the aspect description is called the *pointcut language*.
- A *pointcut expression* is then a description of a joinpoint or joinpoints written in the pointcut language.

The weaver can be seen as a new kind of program evaluator or as an addition to an existing evaluator. In the latter case the weaver is often an automated code patcher. Such a weaver combines aspects with functional modules expressed in some existing programming language through source code transformations, producing a program in that same language which combines the semantics of the two types of components. The produce of the transformation is then evaluated by an existing compiler or interpreter for that language. This is seen by some as a defining property of a weaver. We would like to point out that this is more the result of the experimental state in which AOP research is currently in: developing a full evaluator for a system which might not be used very long is fairly expensive. In all of the AOP approaches we will discuss next, the weavers are based on source transformation. An brief discussion of a particular weaver is given in chapter 3 and a discussion of our own weaver is given in chapter 6.

The different AOP approaches can be classified depending on their generality with respect to applications and aspects. Reverse Graphics presents an example of an application and aspect specific approach. COOL is aspect specific but can be used in many applications. AspectJ is intended to be general with respect to both applications and aspects. Each of these approaches has some advantages and disadvantages.

2.7.1 Reverse Graphics

Reverse Graphics [44] is a processing language that allows sophisticated image processing operations to be defined by composing primitive image processing filters. Each filter produces one new output image from one or several input images. New filters can be created as compositions of existing filters by using the output of one filter as input to another.

The filters model to image processing is straightforward to implement as simple procedure calling, but such an implementation would suffer from efficiency problems. Most of these efficiency problems are related to the excessive creation of intermediate images, images that are not the final output of the program but which are just used as input to another filter and then discarded. When processing large images this leads to high disk swapping (virtual memory) and CPU cache trashing. Another bottleneck is the occurance of redundant computations where the same filter is applied to the same images on several occasions. Consider the use of a negative image for example, this is an operation that is likely to recur in many filters. Finally, there is the problem of excessive looping. The primitive filters produce new images by applying operations on a pixel-by-pixel-basis to their input images and storing the result in the new image: they are expressed as looping constructs over the input images.

Many optimization techniques are applicable to image processing, but these do not fit well with the filters model. One is loop fusion. If the output image of a primitive filter is only to be used as input to another primitive filter, the loops of the two filters can fused into one and the operations applied by the two filters are combined on a pixel-by-pixel basis rather than on an image by image basis. Loop fusion can get difficult however if the filters do not iterate over the image in the same way. Loop fusion can be combined with inlining. Inlining refers to replacing a user defined filter with it's definition. When applied recursively, every user defined filter winds up being expressed only in terms of primitive filters to which loop fusion can then be applied. In some cases this can result in one gigantic loop applying only pixel-wise operations.

Memoization is another applicable optimization technique. Memoization consists of remembering the images a filter has been applied to and the produced output. The next time the filter is applied to the same images, it can simply return the previously generated output rather than computing it again. The trick is of course to remember only those images which will be used again, otherwise a lot of memory will be used unnecessarily. Note that memoization is different from simple caching: with caching there is no definite knowledge about which of the cached items will be used again, while with RG the knowledge about future usage of images is implicitly present in the image processing program. This knowledge can be expressed as the data flow graph of the program.

Optimizing an RG program by hand by applying the above listed techniques leads to high code tangling. Memoization and loop fusion will be mixed. Knowing when to apply either is difficult as they conflict: loop fusion takes away intermediate images while memoization memorizes intermediate images. Careful consideration of whether recomputing or remembering will be more costly is required. Adapting or reusing the code will be nearly impossible as all high level structure has been lost due to code inlining. It can only be done by taking the original code, modifying it and redoing the entire optimization process.

One may wonder whether the optimization cannot be done by a compiler. RG was originally implemented as a library of data abstractions for images and a set of procedures for the basic primitives. It thus depended on a compiler to provide optimizations. However, while there are many optimizing compilers around, techniques such as memoization and loop fusion are too far stretched to be implementable in a general purpose compiler.

To solve the efficiency problem, the RG designers applied some AOP ideas. Actually RG laid the ground for AOP and the original idea was to implement a special purpose compiler. As we already indicated however, implementing a compiler can be fairly expensive. Therefore the RG optimizer acts as a source transformer and does not directly generate executable code. To allow for easy adaptation of the optimizations, the source transformer does not apply optimizations directly, but rather represents an abstract view of the original source in terms of call graphs and data flow graphs to a set of optimizer programs. These optimizer programs can modify this abstract representation, after which the modified representation is translated back to source by the transformer. Optimizer programs for loop fusion, inlining and memoization have been implemented successfully.

Of course, the source transformer is a weaver. The abstract representations of the source it provides to the optimizer programs are joinpoints. The optimizer programs are descriptions of aspects. There is no explicit pointcut language, rather the general purpose language used to write the optimizer programs in is directly used to manipulate the joinpoints. We have used the different terminology here to show more clearly the relation with what it is the RG optimizer is to achieve: efficiency. Note that efficiency is a nonfunctional requirement relevant to any software system. It is usually included in the specifications for the software and if it's not it's just because it's implicitly assumed to be there.

One may wonder since efficiency is relevant to software in general whether the RG weaver can be used to optimize programs in general. This is not the case as the weaver is fairly tightly bound to the RG image processing system. The weaver is also bound to the aspects, even though they are provided separately from the weaver itself. RG is an example of an application and aspect specific AOP approach.

2.7.2 COOL

Our next AOP example is COOL, the coordination language. COOL is intended to handle an aspect we have already encountered in the banking example: synchronization. COOL is a part of D, which is a language framework for distributed programming [42,40]. Another AOP part of D is RIDL, the remote interface description language. We will limit our discussion here to COOL, as both it and RIDL are examples of aspect specific but generally applicable AOP approaches.

COOL is intended to provide synchronization between threads in Java programs. Java programmers may be surprised: most surely Java already provides for this? More experienced Java programmers may nod in agreement when we say that the synchronization provisions of Java are fairly limited. Java provides the keyword `synchronized`. This keyword can be used at the statement level or the method level. The former usage is however discouraged by the Java design team because it leads to tangling of synchronization and functionality at the statement level[2]. An undesirable property indeed. This leaves us with the method level usage of the synchronized keyword which has limited applicability, while in some cases one really cannot get by without statement level synchronization.

The `synchronize` keyword in Java expresses that a body of code locks the object in whose context it is executed for as long as that code is executing. A locked object can not be accessed from any thread other than the thread that acquired the lock. This is a fairly coarse grained locking mechanism. Typically an object has several methods which update its state as well as methods that only access its state. It's safe for two threads to call an accessing method on an object simultaneously, but all other combinations must be avoided (access/update, update/access, update/update). This is not possible with Java's locking mechanism as it can only be used to express that all methods of the object must be excluded, not just the update methods. Excluding too many methods opens a pitfall for deadlocks.

Another problem is the handling of guards. A guard expresses that some condition must be true before a method can be used. Consider a Stack, it cannot be popped if there is nothing to be popped. With single threaded programming there is only one reasonable thing to do when an empty stack is popped: generate an error. With multi threaded programming there is the possibility of waiting for another thread to push something on the stack. The latter option is an often found one in Java programs and is implemented by including a piece of guard checking code at the start of synchronized methods. The purpose of the code is to check the guard, if it's false the lock on the object is released and the thread is made to wait. Other threads must be make sure to notify the system when an update they did to an object might have made a guard true. The other option can be handled similarly but would throw an exception.

There are two problem points to be identified about Java's synchronization mechanism which COOL intends to solve. One is that it is insufficient. The other is that it leads to tangling of the synchronization aspect with the functional code, especially with respect to how false guards are handled. The way false guards are handled is fixed in the code and is not open to change. We have already seen how inheritance and the strategy design pattern might help in solving this problem somewhat.

Synchronization is handled in COOL by defining a coordinator. One coordinator can be associated with every class and the coordinator is expressed entirely separate from the class. The coordinator lists sets of methods from the

[2]The Java tutorial, available online [2], includes the following note: "Note: Generally, critical sections in Java programs are methods. You can mark smaller code segments as synchronized. However, this violates object-oriented paradigms and leads to confusing code that is difficult to debug and maintain. For the majority of your Java programming purposes, it's best to use synchronized only at the method level."

class which are mutually exclusive or self exclusive. Self exclusive methods disable other threads from calling that same method on the same object. Methods which are in the same mutually exclusive set as another method are disabled when that latter method is called. Locking thus happens on the method level of a single object rather than at the object level. A coordinator can define guards and associate these guards with methods. The coordinator also provides for the state switching code of the guards. A coordinator for a bounded stack is given in figure 2.3.

Note that is easy to change the handling of the synchronization and guard handling policy with COOL. It is in fact easy to remove the synchronization altogether when we wish to use the stack in a single threaded application where the locking of objects would pose an unnecessary overhead.

Cross-cutting in COOL is limited to the class level. The pointcut language of COOL is also not very extensive: a joinpoint is just a method in the class and the pointcut language consists of naming the methods in the mutex/selfex-sets.

2.7.3 AspectJ

AspectJ is our last example of an AOP approach for this section. AspectJ is intended as a simple and practical AOP extension to Java. The idea is to move AOP out of the research labs and assess it's usability in industrial programming. The goal is to see how day-to-day programmers would use the technology and whether they can successfully achieve the benefits promised by AOP: more reusable code with clearer separation of concerns. We will discuss AspectJ more extensively than the previous two examples, as it forms the direct basis for our own AOP approach. The discussion is largely based on overviews given by the AspectJ development team [35,41,34], as well as the documentation found on the project's web site[3].

We have observed that with AspectJ, there is a move away from the identification of aspects with nonfunctional requirements. While RG and COOL are directly related to such requirements as efficiency through optimization and efficiency through concurrency, many AspectJ examples we have seen handle parts of the functional requirements which can wind up as cross-cutting concerns. This is not necessarily a bad thing, AOP is intended to add new decomposition and composition mechanisms to the software engineer's tool box. In the end, it is up to the software engineer to decide which tools best achieve his goals.

Joinpoint model

The AspectJ joinpoint model is defined in terms of a program's execution graph and class graph. The execution graph contains nodes for the reception of a message by an object, the sending of a message by an object and the accessing and updating of an object's state. The nodes are linked in a way that follows the order in which they are executed by the program. The class graph is a rather direct representation of the classes in the program, which are linked through normal inheritance relationships. The class graph can also be seen as to include the methods defined on the classes. In AspectJ terminology, the execution graph allows for dynamic cross-cutting while the class graph allows for static cross-cutting.

[3]www.aspectj.org

```
class BoundedStack {

  final static int MAX = 100;

  Object[] contents = new Object[MAX];
  int top = 0;

  void push(Object element) {
    contents[top++] = element;
  }

  void pop(Object element) {
    contents[--top] = element;
  }

}

coordinator BoundedStackCoord : BoundedStack {

  selfexclusive{pop, push};
  mutexclusive{pop, push};

  cond boolean full = false;  /* guard indicating full-state of stack */
  cond boolean empty = true;  /* guard indicating empty-state of stack */

  push : requires !full;
    on_exit {
      if (top == MAX) full = true;
      if (top == 1) empty = false;
    }

  pop : requires !empty;
    on_exit {
      if (top == 0) empty = true;
      if (top == MAX - 1) full = false;
    }

}
```

Figure 2.3: COOL Coordinator expressing the synchronization aspect on a bounded stack.

Dynamic joinpoints

To help the reader in gaining an insight into the joinpoint model of AspectJ we will illustrate with a simple example. Figure 2.4 shows an outline of the code of the example and an informal depiction of it's execution graph is shown in figure 2.5. The example is taken from [35].

The example in figure 2.5 shows the creation of three objects: two points and a line connecting them. These objects are depicted in the figure as big circles, with their methods as rectangles inside them. The execution graph is depicted on top of the objects to show how it cross-cuts the object structure. The graph consists of joinpoints, shown as small circles connected by edges in a comes-after relationship. The joinpoints in the example have been numbered, an explanation for each joinpoint is given below:

1: A message send joinpoint at which the message slide is sent to the object ln1.

2: A message reception joinpoint at which the message slide is received by the object ln1.

3: A method execution joinpoint at which the method matching the slide message and the number and the type of its arguments is executed.

4: A state access joinpoint where the field `a` of object ln1 is referenced. The value in the field is the object pt1.

5: A message send joinpoint at which the message slide is sent to the object pt1.

6,7: Similar to 2 and 3.

8: A message send joinpoint where the message slide is sent to the object ln1.

9,10: The reception and execution joinpoints for the message sent in 8.

11: A state access joinpoint where the field `x` of object pt1 is read. After this point, control returns back through joinpoints 11, 10 and 9 to 8.

12: A message send joinpoint where the message setX is sent to object pt1.

etc. until control finally passes back through to joinpoint 1.

A summary of the different types of dynamic joinpoints is given in table 2.1

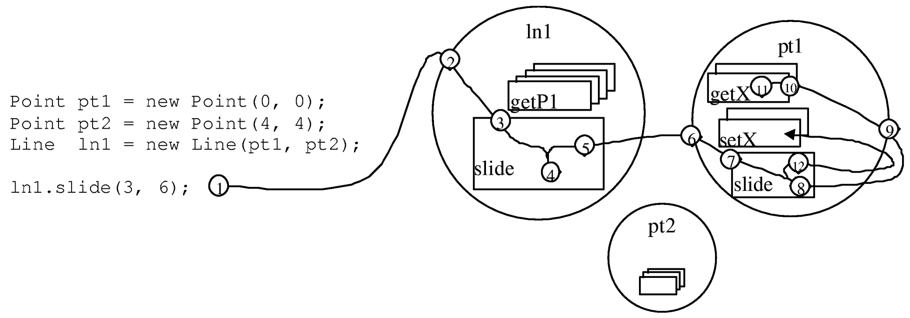
Pointcuts on dynamic joinpoints

The pointcut language for dynamic joinpoints in AspectJ is used to describe a set of joinpoints by stating the conditions they must meet. Primitive conditions are provided which can be combined using the normal boolean operations.

Some of the primitive conditions match directly on the type of a joinpoint and some signature associated with joinpoints of that particular type. One example is the reception condition, which can only match on message reception joinpoints. Any reception joinpoint only matches the reception pointcut if the

```
class Point {
  private int x, y;

  /* code for constructor, getters and setters should
     go here */

  public void slide(int dx, int dy) {
    setX(getX() + dx);
    setY(getY() + dy);
  }
}

class Line {
  private Point a, b;

  /* code for constructor, getters and setters should
     go here */

  public void slide(int dx, int dy) {
    a.slide(dx, dy);
    a.slide(dx, dy);
  }
}
```

Figure 2.4: Skeletal code for the dynamic joinpoints example



```
Point pt1 = new Point(0, 0);
Point pt2 = new Point(4, 4);
Line  ln1 = new Line(pt1, pt2);

ln1.slide(3, 6);
```

Figure 2.5: An informal depiction of dynamic joinpoints

message send constructor invocation	An object sends a message to another object. The joinpoint is associated with the first object. Message sends to classes are also joinpoints of this type, as well as invocations of constructors.
message reception constructor invocation reception	An object receives a message from another object. The joinpoint is associated with the first object.
method execution constructor execution	A method or constructor is executed. The joinpoint is associated with the object in which the method is executed.
state access	A field of an object or class is read.
state update	A field of an object or class is assigned to.
exception handler execution	An exception handler is executed.
class initialization	A field of a class is being initialized.
object initialization	A field of an object is initialized at object creation time.

Table 2.1: The different types of joinpoints in the dynamic joinpoint model of AspectJ

signature specified by the condition matches as well. The message signature specifies the name, number and type of arguments of the message being received at the reception joinpoint. Similar primitive pointcuts are available for the message send and method execution joinpoints, as well as the field getting and setting joinpoints. In the latter case the signature mentions the name and type of the field. Finally primitives for exception handling joinpoints and initialization joinpoints are provided, but we will not discuss these further. Some examples of joinpoint type matching pointcuts are given below:

- `calls(void Account.withdraw(int))`
  Matches message send joinpoints, where the message withdraw is sent to an object of type `Account`, with an int-argument and no return value.

- `executions(void Account.withdraw(float))`
  Matches executions of the method withdraw specified by the class `Account`, where the argument is of type float and the return value is void.

- `receptions(void Account.withdraw(*))`
  Matches receptions of a message withdraw by objects of type `Account`, where the type of the argument does not matter.

- `calls(* *.*(*,*,*))`
  Matches any message send taking exactly three arguments.

- `calls(* Account.*(..))`
  Matches any message send to objects of type `Account`.

- `gets(int Account.balance))`
  Matches state access joinpoints where the field balance of type int is accessed in an object of type `Account`.

- `sets(String Account.*)`
  Matches any state update joinpoints where a field of type String is assigned to in an object of type `Account`. (or static fields in the class `Account`)

Note that AspectJ allows the use of wildcards in the signatures occurring in pointcuts. A wildcard is denoted by an asterisk. The wildcard simply means "matches anything". This can be the message name, a package name or type name depending on where the asterisk occurs. For argument lists the double dot wildcard ".." can be used to match any number and type of arguments. We have used some wildcarding in the examples above.

Other primitive pointcuts specify some conditions on the origin of the joinpoint. One specifies the type of the object that is associated with the joinpoint. The within-pointcuts match joinpoints based on the lexical extent of the joinpoint. To understand this latter type of pointcut keep in mind that dynamic joinpoints come about through the execution of statements in the programs source code. Every dynamic joinpoint corresponds roughly to some statement or set of statements in the program. The lexical extent pointcuts state some condition on the location of this statement in the program, either the method or the class in which the statement is contained. Finally there are the control flow pointcuts, these specify another pointcut. If a joinpoint matches this other pointcut, then all joinpoints coming after the first joinpoint in the execution graph match the control flow pointcut, unless control has already passed back through the first joinpoint. Some examples of the origin type joinpoints are given below:

- `instanceof(Account)`
  Matches any joinpoints which are executed in the context of an object of type `Account`.

- `within(Account)`
  Matches any joinpoints which are associated with declarations or statements in the class `Account`. (also statements in static methods, whereas instanceof will not match those)

- `withincode(* Account.withdraw(..))`
  Matches any joinpoints which are associated with statements in any of the withdraw methods specified by the `Account` class.

- `cflow(calls(* Account.withdraw(..)))`
  Matches any joinpoints which occur after a withdraw message has been sent to an object of type `Account`, and before the object is done handling this message.

As already indicated, primitive pointcuts can be combined using boolean operators. New formed pointcuts can be given names through pointcut declarations. A simple example:

```
pointcut internalsTouched() :
        gets(* Account.*) || sets(* Account.*);

pointcut touchedFromOutside() :
        internalsTouched() && !instanceof(Account);
```

The first pointcut in the example matches joinpoints where a field of an `Account` object is accessed or updated. The second pointcut matches the same joinpoints, except if the accessing or updating is done by an `Account` object. This can be used to track objects that manipulate (public) fields of `Account` objects directly, but which are not `Account` objects themselves. This direct accessing practice is discouraged in OO programming, but is sometimes applied for reasons of efficiency. It can be helpful during debugging to check these direct accesses.

Advice

Advices on dynamic joinpoints are used by aspects to contribute new behavior or possibly replace existing behavior. An advice specifies a piece of regular Java code. An advice essentially expresses that the execution graph of this code must be inserted into the execution graph of the global program. The joinpoints at which the code is to be inserted is of course designated with a pointcut.

Advices can be inserted at different places in the execution graph. When the pointcut associated with the advice matches a joinpoint, the advice is inserted before, after or around the joinpoint. Before means the code of the advice is executed before the code of the joinpoint. After means the code of the advice is only executed after control flow passes back through the joinpoint. Finally, around advice can be used to compose with joinpoints in a more complicated fashion. It can even be used to replace that part of the execution graph which consists of the nodes coming after the joinpoint and before control passes back through it, with the execution graph of the advice's code. A simple example of an after advice is given below, it uses a pointcut we defined earlier:

```
after() : touchedFromOutside() {
    System.out.println("Account object touched from outside");
}
```

Aspects

Aspects in AspectJ are defined by aspect declarations and have a form similar to that of Java classes. They can even have methods and instance variables to complement the advice declarations and pointcut declarations.

An example aspect is given below. It counts the number of times a message is sent to any `Account` object. It consists of a variable to keep the counter, a named pointcut that matches all reception joinpoints on `Account` objects, an advice defined on that pointcut to increment the counter, and a method which does the actual incrementing of the counter.

```
aspect AccountInvocationCounter {
  int counter = 0;

  pointcut messages() : receptions(* Account.*(..));

  before() : messages() {
    incrementCounter();
  }
  void incrementCounter() {
    counter++;
  }

}
```

Static joinpoints and introductions

So far we have concentrated on the mechanisms associated with dynamic joinpoints in AspectJ. We will now discuss the static joinpoints. These are associated with the class graph of the program. Introductions are like advices, but express additions or modifications to the class graph rather than the dynamic joinpoints. Introductions are fairly easy to understand so we will not give a more formal discussion but will illustrate them with an example.

```
aspect AccountExtension {
  Date Account.lastAccessed;

  Date Account.getLastAccessed() {
     return lastAccessed;
  }
}
```

The example above introduces a new instance variable named `lastAccessed` into the class `Account`. A new method for accessing the variable is introduced as well. Advices could be added to the aspect for setting the `lastAccessed` variable whenever an object is sent a message.

Other mechanism in AspectJ

We extensively discussed the most important mechanisms in AspectJ in this section. There are some which we have not discussed for reasons of brevity. One of these is the possibility to create aspects as specializations or refinements of other aspects, similar to subclassing in OO. The refined aspect can override advices and pointcuts from the parent aspect.

Another mechanism is the use of variables in advices which come from a pointcut. This is briefly mentioned in chapter 6.

Finally there is the use of aspect instances. We will discuss aspect instances in chapter 5 in the context of our own AOP system.

2.7.4 Pointcut languages

Having illustrated some AOP approaches, we can make some comments on the pointcut languages used. The three examples use a different style of pointcut language, which we can classify as follows:

Simple by name referencing: joinpoints are referenced by a name which they naturally have in the program. This scheme is employed in COOL, which references methods by their name.

Query language: the pointcut language used by AspectJ resembles that of a query language like SQL with the weaver providing a “database” of joinpoints. Characteristic for query languages is that they are not Turing complete. In other words: query like pointcut languages cannot be used to perform computation about the joinpoints to determine which ones to match.

General purpose language: in Reverse Graphics, the weaver passes the joinpoints as data to the optimizer programs. Since these programs are written in a general purpose programming language, they can perform computation on the joinpoints.

The power of the pointcut language is of course also influenced by the joinpoints the weaver provides. The most simplistic kind of joinpoint model would be stated in terms of source lines, a joinpoint would simply be a single line of source code. This would actually be a very powerful model in terms of the modifications it allows an aspect to make, but is not very practical. Joinpoint models should have more structure. This structure can either be modeled directly as the modular structure of the program to which aspects are applied, such as a class graph. It can also be a structure which is implicitly present in the program but which is not the structure along which the program is constructed, call graphs and data flow graphs are examples. This latter type of joinpoint model fits better with the idea of aspects, as aspects are cross-cutting concerns exactly because they follow more naturally a structure which is implicitly present but not explicitly stated. However, it also requires us to identify which structure it is a specific aspect follows. Much research in AOP still needs to go into this latter topic.

2.8 Summary

In this chapter we gave an overview of how Aspect-Oriented Programming has emerged into software development.

To deal with the complexity involved in constructing a software system we try to decompose it into smaller pieces which are then composed to form the system. To effectively have less complex pieces, each piece should separate some concerns from other concerns which are addressed by other pieces. This is known as the principle of separation of concerns.

The need for having different kinds of (de)composition structures is now recognized. At the more practical level of software implementation, problems such as code tangling caused by cross-cutting concerns have been observed.

Aspect-Oriented Programming has been proposed as a way to deal with cross-cutting concerns at the source code level. The general idea is to use an Aspect Weaver to combine aspects with the components implemented in a traditional programming language. Generally the weaver extracts a structure which is implicitly present in the program, such as a call graph, data flow graph or execution graph which is then presented to the aspects. The weaver extracted structure is the joinpoint structure. An aspect influences the semantics of the program by manipulating the semantics of the joinpoints in some manner.

We have explored a few different examples of AOP and considered the different ways in which they allow aspects to manipulate a joinpoint structure. We specifically concentrated on how the joinpoints to be manipulated are picked out: either through the use of some general purpose (imperative) language or a query language.

Chapter 3

Logic meta programming

In this chapter we concentrate on the use of logic meta programming (LMP). First the general ideas involved in meta programming are explained. We then describe logic meta programming and two systems for doing logic meta programming: TyRuBa and SOUL. At the end of the chapter an application of LMP to AOP is described. In the next chapter we will explore the use of logic meta programming to express cross-cutting.

3.1 Introduction

3.1.1 Meta programming

We will first explain the general idea underlying meta programming and define some of the related terminology. The discussion is based on Maes's Ph.D.Thesis [43].

A *program* describes a computational system which is about a certain problem domain. This view on programming was already introduced in chapter 2, where we introduced the problem domain of a bank which consists of clients, accounts, transactions etc. These latter are the entities and concepts present in the problem domain. A *computational system* reasons and acts upon representations of these concepts, referred to as the data of the computational system.

A very interesting class of computational systems are those whose problem domain consists of computational systems. Such a system is referred to as a *meta system*. The program of the meta system is called a *meta program*. Such a program thus describes a computational system that manipulates representations of computational systems. Often meta programming is restricted to manipulating the program of a computational system, in which case meta programming can simply be described as "writing programs that manipulate programs as data".

Meta programming may seem like a fairly exotic idea but is actually rather commonly applied. A compiler or an interpreter is an example of a meta program. The macro system provided by some programming languages is an example of where meta programming is used to generate programs. Other types of code generators are also meta programs. Another example is the GUI building tool provided by many integrated development environments. These allow programmers to construct a GUI through a GUI[1]. With a press of a button program code to construct that GUI can be automatically generated.

Programs are expressed in a programming language. A *programming language* provides constructs for representing a computational system's data and how it is manipulated. Two types of programming languages can be discerned: general-purpose languages and domain specific languages. Domain specific languages are tuned to programs dealing with a specific problem domain, whereas general-purpose languages lack such tuning. Scheme is an example of a general-purpose language, whereas Postscript is tuned to controlling a printer and Matlab to mathematical processing.

A *meta language* is a domain specific programming language for dealing with meta systems. Such a language thus has representational capabilities which facilitate the representation and manipulation of programs and even their computational systems.

A meta language is typically not just tuned for dealing with meta programs in general, but for meta programs which are about programs in a specific programming language. This latter language is referred to as the *base language* of the meta language. Programs expressed in this language are then *base programs*.

3.1.2 Logic programming

In the previous section we defined an important property of a programming language in that it provides constructs for representing and manipulating entities from a problem domain. Programming languages can be classified in different paradigms based on how they view a problem domain:

Imperative programming: in imperative programming, whether it is procedural programming or OO programming, emphasis is put on the state of entities and how this state evolves over time. A program specifies a sequence of steps to be taken where each step changes the state of some entity.

Functional programming: this paradigm is based on applying transformations to entities. Like a mathematical function, each transformation results in a new entity. A program specifies a transformation which is composed out of smaller transformations.

Logic programming: logic programming is concerned with specifying the basic knowledge of a problem domain, and how new knowledge can be derived. A program is comprised of a set of facts which represent known relations between entities in the problem domain and a set of rules representing how new facts can be derived from known facts. A specific problem is formulated as a query to an inference engine. A query is a fact-like statement on the problem domain for which the engine tries to derive the fact itself or its negation.

Each of these paradigms has advantages and disadvantages. We will only discuss those of logic programming here. One attractive property of logic programs is the ease with which they can be written and understood. This is partly

[1]A meta-GUI if you like. In fact the entire IDE is a meta system.

due to the fact that knowledge never gets lost in logic programs as it does in imperative programs where changing state means old state information gets lost. Logic programming languages are also typically very open, meaning it is easy for anyone to add new facts and rules to make an existing program more specific. Finally, logic programs have no notions of input or output parameters and can be used in different ways. This latter property is referred to as the multi-way property. The major disadvantage of logic programs is that they can be quite slow.

3.2 Logic meta programming applications

From the discussion in this chapter's introduction, it should be clear that a logic meta language is a programming language based on the logic paradigm for manipulating programs. A logic meta language thus provides constructs for representing programs as facts, rules to extract knowledge from the program as well as ways of generating programs.

In this section we discuss two particular examples of logic meta programming systems. One is TyRuBa and the other is SOUL. There are some differences between the two:

base language: TyRuBa employs Java as its base language while SOUL uses Smalltalk.

emphasis: In TyRuBa LMP is mostly used to generate programs, while SOUL is mostly used to reason about them.

Both SOUL and TyRuBa are based on Prolog, by far the most popular logic programming language around. A gentle introduction to Prolog is given by Flach [25]. Most of the examples given in this dissertation are quite simple and do not require much knowledge of Prolog.

In the context of TyRuBa we will discuss how programs are represented as facts, how reasoning about the program is done and how programs are generated.

With SOUL we present an extensive framework for reasoning about programs.

3.2.1 TyRuBa

TyRuBa stands for Type Rule Base. It was conceived by Kris De Volder and is described in-depth in his Ph.D.Thesis [52]. The goal of TyRuBa was to support a more active use of types in statically typed programming languages when using generic types. Generic types are best known as C++ templates. TyRuBa concentrates mostly on representing and instantiating template programs.

Representing base programs as facts

An important question to be addressed by any designer of a meta language is how the base programs will be represented. In the case of LMP the representation will be as a set of facts. The most simplistic representation would consist of a single fact, expressing there's a program with a certain text:

```
program(``... program text ...'').
```

This representation is not very good as it makes nothing explicit about the inherent structure of the program. As was explained in the previous chapter, a program is a composition of ever smaller structures. A program written in a class based OO language is not just a string of characters, but rather a combination of classes each of which in turn consists of variables and methods. Such structure is preferably reified in the meta language's representation mechanism. The structure is not necessarily to be based on an abstract syntax tree, as this includes information which is usually irrelevant to the semantics of the program, such as the specific order in which classes are defined.

In a logic meta language, the meta language designer can include rules for deriving the structured representation from the one-string-fact representation as part of the language definition. Or a separate code generator could be used which transforms a program into base facts. In the particular case of the SOUL meta system which we will present shortly, the meta system is embedded in a programming environment for the base language which already has a structured representation of programs available and no derivation from program text is necessary.

TyRuBa representation of Java programs

TyRuBa uses the code generator approach to generate logic representations of Java programs. There is no standard representation associated with TyRuBa. We will briefly discuss a particular representation based on an example given by De Volder which is also discussed further in this chapter [53]. As an example of a particular instance of this representation we use the familiar `Account` example from the previous chapter .

The presence of a class is represented using a `class` fact:

```
class(Account).
```

Instance variables associated with classes are represented using `var` facts, we show the general form and a particular example:

```
var(?Class, ?VarType, ?VarName, { ... declaration code ...}).

var(Account, int, balance, { int balance = 0; }).
```

The presence of methods in the classes are indicated with `method` facts:

```
method(?Class, ?ReturnType, ?MethodName, ?ArgumentTypesList,
        { ... declaration head ... },
        { ... method body ... }).

method(Account, void, withdraw, [int],
        { public void withdraw(int amount) },
        { balance -= amount; }).
```

Notes on TyRuBa syntax

Knowledgeable Prolog users may be a bit confused about TyRuBa's syntax. The most important differences are:

variables: variables in TyRuBa are denoted with a question mark, rather than having the name of the variable start with a capital as in Prolog. Thus while `Name` is a variable in Prolog, it is just a constant in TyRuBa, while `?Name` is a variable in TyRuBa.

compounded terms: compounded terms are written with `'<'` and `'>'` rather than the round brackets used by Prolog. Thus the fact `number(complex(1,2))` becomes `number(complex<1,2>)` in TyRuBa.

quoted code blocks: this is an example of the tuning of the TyRuBa language to meta programming. Quoted code blocks are denoted with `'{'` and `'}'`, everything in between is quoted code. Quoted code blocks are used to represent "raw" Java code, as in the method and variable representation used above. However, quoted code blocks are *not* just strings. Rather they are treated by TyRuBa as lists, just like Prolog lists, but the different terms of the list are separated with spaces instead of commas. This means that variables and compounded terms can appear in quoted code.

Generating code

TyRuBa is very suited for template based meta programming, consider the following simple fact:

```
complex_template({ int real = ?real; int imaginary = ?imaginary; },
                 complex<?real, ?imaginary>).
```

The fact represents a simple template for generating assignments of complex variables. The following query would give us a particular instantiation of the template:

```
:- complex_template(?code, complex<1, { 2 + 3 }>).
```

The solution to this query is:

```
?code --> {int real = 1; int imaginary = 2 + 3 ; }
```

Due to the unification process the variable `?code` has become bound to the complex template. The variables occurring in this template have been respectively bound to the literal `1` and the code block `{ 2 + 3 }`, producing the final instantiated template.

When using the representation for Java programs presented earlier, TyRuBa can be used to generate Java programs. The idea is to simply have rules conclude that certain classes, variables or methods must exist. This can be combined with a representation of an existing program to apply transformations to that program. For example, the following rule adds a printClassName method to every class:

```
method(?Class, void, printClassName, [],
        { public void printClassName() },
        { System.out.println("?Class"); }) :-

        class(?Class)
```

At the end of this chapter we discuss how TyRuBa can be used to implement a weaver for an Aspect-Oriented Programming system. We now turn our attention to SOUL.

3.2.2 Smalltalk Open Unification Language

The Smalltalk Open Unification Language was invented by Roel Wuyts [55]. Part of his goal in using SOUL was to extract design information from programs.

SOUL is complementary to TyRuBa, and comes with an extensive set of rules to support the extraction of design information. This set is called the "Declarative Framework". This section concentrates on presenting the declarative framework. But first we discuss some syntactic differences between SOUL and TyRuBa.

SOUL syntax

SOUL has a slightly different syntax from TyRuBa and standard Prolog:

Clearer notation: The `:-` notation is replaced with the keyword `if` and the keywords 'Rule', 'Query' and 'Fact' are used to distinguish between rules, queries and facts respectively.

Lists: lists are denoted with `'<'` and `'>'` rather than the square brackets used in Prolog and TyRuBa.

Compounded terms: compounded terms are written as in normal Prolog, not as in TyRuBa.

Variables: variables are denoted with question marks, as in TyRuBa.

SOUL has quoted code blocks as TyRuBa has, but adds something new: Smalltalk terms. Unlike quoted code which represents source code in a simple to manipulate format, a Smalltalk term is code in the base language that is to be executed as part of the reasoning process. The difference is easily shown with the complex number template:

```
Fact complex_template({ int real = ?real; int imaginary = ?imaginary; },
                      complex(?real, ?imaginary)).
```

When we issue the following query, we get the same result as in TyRuBa:

```
Query complex_template(?code, complex({ 1 }, { 2 + 3 })).

[?code->{int real = 1 ; int imaginary = 2 + 3 ; }]
```

When we replace the quoted blocks with Smalltalk terms however, we get a very different result:

```
Query complex_template(?code, complex([ 1 ], [ 2 + 3 ])).

[?code->{int real = 1; int imaginary = 5; }]
```

As we see from the result, the expression `2 + 3` has been evaluated rather than simply being substituted into the template.

The expression between the squared brackets is an actual Smalltalk expression. The value it is evaluated to can be bound to a SOUL logic variable. This coupling between the SOUL logic language and Smalltalk is called *linguistic* *symbiosis*. Linguistic symbiosis between a meta language and its base language was introduced by Steyaert in his Ph.D. dissertation [49].

Note that *any* Smalltalk expression[2] is allowed in a Smalltalk term, and it is allowed to return *any* value. Through unification the value can be bound to a SOUL variable. In effect, the value "travels" from the base level (Smalltalk) to the meta level (SOUL).

Smalltalk values can also "travel" from the meta level to the base level. This is done by using SOUL variables inside Smalltalk terms similar to how Smalltalk variables are used. An example query:

```
Query equals(?x, [2]), equals(?y, [ ?y + 3 ]).

[?x->2 ?y->5]
```

The traveling of values between the meta and base level is called the up/down mechanism and is further described by Wuyts in his Ph.D. dissertation [55].

We note that a Smalltalk term can also be used as a clause instead of as a term. In that case the code inside the term is expected to evaluate to true or false, indicating the success or failure of the query. The following rule can be used to test whether a number is smaller than another number. Of the two given queries the first succeeds and the other one fails:

```
Rule smaller(?x, ?y) if
        [ ?x < ?y ].

Query smaller([5], [7]).
Query smaller([9], [3]).
```

Finally, we discuss the `generate` predicate. This predicate takes two arguments, the second of which is a Smalltalk term. The term is expected to return a collection of values [3], i.e. an instance of `Array` or another subclass of `Collection`. The values of the collection are successively bound to the variable given as first argument to the `generate` predicate. Thus the predicate succeeds as many times as there are values in the collection. An example:

```
Query generate(?x, [Array with: 1 with: 2 with: 3]).
```

The query has three solutions:

```
?x -> 1

?x -> 2

?x -> 3
```

[2]Expressions that result in side effects are allowed, but should usually not be used as logic languages are normally side-effect less.

[3]This is the case in QSOUL version 2, earlier versions of SOUL expected a stream. The difference is minor and of no importance here.

SOUL representation of Smalltalk programs

SOUL has a somewhat different way of representing Smalltalk programs than TyRuBa's. Not just the representation itself is different but also the way it is obtained. SOUL does not use a preprocessor to obtain a set of logic facts representing a base program, but rather uses a combination of its linguistic symbiosis mechanism and Smalltalk's reflective capabilities.

Reflection refers to the capability of a computational system to access and manipulate the computational system that is interpreting it. Usually the access is limited to the data of the interpreter. Since this data represents the computational system the interpreter is interpreting, a computational system can get access to itself, especially to its own program. Simply put, reflection is about "programs manipulating themselves as data".

We will not go into a theoretical discussion on reflection here but will restrict ourselves to its practical use in Smalltalk and SOUL. We expect any reader somewhat familiar with Smalltalk to have some awareness of the reflectional constructs provided by Smalltalk. If this is not the case we refer to chapter 16 of the standard book on Smalltalk [28] for a practical discussion of reflection. We only have a single simple example here which should not be too hard to understand.

In the following Smalltalk code, values are checked to be classes, the result is also shown:

```
1 isKindOf: Class --> false
Array isKindOf: Class --> true
```

Thus SOUL can easily provide a rule to check whether something is a class[4]:

```
Rule class(?x) if
        [ ?x isKindOf: Class ]
```

The other way around, getting all the classes in the base program is done using the `generate` predicate and is shown below. This way the multi-way property of logic programming is preserved.

```
Rule class(?x) if
        generate(?x, [ Smalltalk allClasses ])
```

Similar rules for the instance variables in classes and the methods are available. These are fairly similar to the facts we used in TyRuBa. They are summarized in table 3.1.

The SOUL declarative framework

As mentioned, the most important difference between SOUL and TyRuBa is that SOUL comes with a set of rules intended to reason about programs. This set of rules is the *declarative framework*. To bring some clarity into the different types of predicates the framework provides they have been divided into layers:

[4]The implementation of the rules as given here differs from the actual implementation in SOUL. The idea is however the same.

`class(?class)`	?class is a class
`superclass(?super, ?sub)`	?super is a superclass of ?sub
`instVar(?class, ?iv)`	?iv is the name of an instance variable in class ?class
`method(?class, ?method)`	?method is a method implemented in class ?class

Table 3.1: SOUL representation layer predicates

logic layer: the logic layer contains the kind of basic predicates one expects from standard Prolog environments. These include predicates for handling lists, arithmetic and the logic meta predicates (`var`, `atom`, `findall` and the like).

representational layer: the representational layer holds the predicates used to present a logic, structured representation of base programs. The most important were already given in table 3.1.

basic layer: the basic layer contains a lot of auxiliary predicates that facilitate reasoning about programs. The representational layer provides the most primitive information on the structure of a program, the information that is directly apparent from the source code. The basic layer contains mostly predicates to extract more high leveled information from this primitive information.

design layer: the design layer contains predicates to extract information on design patterns used in the base program, as well as programming conventions used.

We next give a short overview of some predicates in the basic layer and the design layer, to give the reader an idea of what is possible with the declarative framework.

The basic layer

Some simple examples of predicates in the basic layer are the `subclass` and `hierarchy` predicates. These are simple enough to allow us to show their implementation here:

```
Rule subclass(?subclass, ?superclass) if
        superclass(?superclass, ?subclass).
```

The `subclass` rule simply expresses that any class that is a superclass of another class, must have that latter class as its subclass. Next are the `hierarchy` rules:

```
Fact hierarchy(?root, ?root).

Rule hierarchy(?root, ?directSubclass) if
        subclass(?directSubclass, ?root).
```

```
Rule hierarchy(?root, ?indirectSubclass) if
        subclass(?directSubclass, ?root),
        hierarchy(?directSubclass, ?indirectSubclass).
```

These rules simply express how two classes are related through a subclassing chain. Due to the multi-way property, the rule can be used in many different ways: to check whether two classes are related through a chain, to generate all the classes above or below a certain class:

```
Query hierarchy([Collection], [Array]).
Query hierarchy(?x, [Array]).
Query hierarchy([Collection], ?y).
```

Of course many more predicates are provided in the basic layer. We will explain some of these when we use them in examples in the coming chapters.

Design layer

The design layer contains predicates for extracting design information and programming conventions.

A simple programming convention employed in all Smalltalk programs is that of accessor and mutator methods. Unlike other languages such as Java and C++, Smalltalk has no concept of “private” and “public” variables. Instance variables are in a sense always “private”, and so-called accessor and mutator methods must be provided to access these variables directly from the outside. An accessor method is one that has the same name as a variable and does nothing more than returning that variable’s value. A mutator method is similar, but takes an argument and sets the variable’s value.

The `accessor` and `mutator` predicates can be used to check whether a class implements an accessor or mutator for a specific variable, or to generate the variables it provides such methods for etc. We do not show the implementation here, just the form of the predicates:

```
accessor(?class, ?method, ?varname).
mutator(?class, ?method, ?varname).
```

As an example of an even more advanced predicate, consider the `visitor` predicate:

```
visitor(?visitor, ?element, ?accept, ?visitSelector)
```

The variable `?visitor` is a class that implements the visitor part of the Visitor design pattern [27]. The variable `?element` can be bound to classes representing elements from the tree structure the visitor visits. The variable `?accept` refers to the selector, a Smalltalk symbol, that is the name of the accept method for the visitor in the element class. Conversely, `?visitSelector` is the selector of the method implementing the visit method in the visitor class for the particular element class.

The `visitor` predicate can be used in many ways. It can be used to find the accept and visit methods for a particular combination of a visitor and element class for example. It can potentially even be used to find all classes implementing a visitor design pattern.

`mutuallyExclusive(?cl, ?m1, ?m2)`	The methods ?m1 and ?m2 in the class ?cl are mutually exclusive. (?m1 and ?m2 can be the same method)
`requires(?cl, ?m, ?e)`	The method ?m in class ?cl should not be run until expression ?e is true. ?e is a Java expression.
`onEntry(?cl, ?m, ?s)`	The statement ?s must be executed when entering the method ?m in class ?cl.
`onExit(?cl, ?m, ?s)`	The statement ?s must be executed when exiting the method ?m in class ?cl.

Table 3.2: Basic synchronization strategy predicates

User layer

There is not really a “user layer” in SOUL, though we can obviously add new rules to SOUL or even change existing ones. As a simple example of how we can extract new information from programs, consider the following rule which simply expresses that a variable is “public” if accessor and mutator methods are provided for it:

```
Rule public(?class, ?var) if
        accessor(?class, ?accessmethod, ?var),
        mutator(?class, ?mutatemethod, ?var).
```

3.3 Aspect-Oriented Logic Meta Programming

3.3.1 TCOOL: The basics

In his Ph.D. dissertation [52] and in a separate paper [53] De Volder introduced the notion of *Aspect-Oriented Logic Meta Programming*. The idea was to show that Logic Meta Programming is a suitable formalism for implementing generic weavers. The idea is based on the observation that most AOP approaches use weavers based on source code transformation, something TyRuBa was meant to do well.

The example De Volder presented [53] to support his claim is based on COOL, we will refer to it as TCOOL to make the distinction. The general idea behind TCOOL is the same as that of COOL: allow for the separate description of basic functionality and synchronization strategy so as to prevent code tangling. As in COOL, the basic functionality is implemented in Java. The description of the synchronization strategy is done using simple facts written in TyRuBa[5]. The format of the basic facts that can be used is shown in table 3.2.

Using the basic facts one can easily represent synchronization strategies, the following example expresses that the `push` and `pop` methods of a `Stack` are mutually exclusive and that the `push` method is mutually exclusive with itself. The last fact in the example expresses that the push method can only be executed if there is room in the `Stack`:

[5]De Volder notes these can be typed in directly by the programmer, or can just be parsed from COOL source.

```
mutuallyExclusive(?cl, ?m1, ?m2) :-
        mutuallyExclusiveList(?cl, ?methods),
        element(?m1, ?methods),
        element(?m2, ?methods),
        NOT(equal(?m1, ?m2)).

mutuallyExclusive(?cl, ?m, ?m) :-
        selfExclusiveList(?cl, ?methods),
        element(?m, ?methods).
```

Figure 3.1: Rules to simplify specification of synchronisation strategies in TCOOL.

```
mutuallyExclusive(Stack, push, pop).
mutuallyExclusive(Stack, pop, pop).
requires(Stack, push, { !full() }).
```

The weaver for TCOOL is implemented using TyRuBa's program transformation capabilities. In discussing TyRuBa we showed how templates can be used in it. Essentially the TCOOL weaver uses this template processing capability by taking the original bodies of methods as written without synchronization and inserting the body into a synchronization template. We refer to De Volder's paper for the details.

3.3.2 More advanced TCOOL

Some extra rules are provided on top of the basic facts to make specifying synchronization strategies easier, and even more resemble the way it is done in COOL. There is a rule to allow for the specification of sets of mutually exclusive methods instead of the pairwise notation used with the basic facts. A rule for specifying sets of self exclusive methods is also provided. These simple rules are shown in figure 3.1.

The extra rules allow one to specify synchronization more compactly. One can for example specify an entire list of methods which are all pair-wise exclusive with just one fact using `mutuallyExclusiveList`. The rules just introduced automatically derive the equivalent representation in the more basic facts.

We can now easily express the familiar synchronization on a Stack class in the extended TCOOL. The facts are shown in figure 3.2.

3.4 Using inspects/modifies

De Volder points out that in AOLMP it is easy to introduce an even more abstract description of synchronization than the mutex/selfex-sets based representations given previously. He notes that while COOL is very declarative in that it does not specify *how* synchronization is to be done (i.e. through locking and unlocking of semaphores), it still specifies *what* is to be done rather than *why*. For example, the reason for making some of the methods mutuallyExclusive with all the other methods is that they modify `Stack` objects. We can

```
selfExclusiveList(Stack,[push,pop,print]).
mutuallyExclusiveList(Stack,[push,pop,peek]).
mutuallyExclusiveList(Stack,[push,pop,empty]).
mutuallyExclusiveList(Stack,[push,pop,full]).
mutuallyExclusiveList(Stack,[push,pop,print]).

requires(Stack,push,{!full()}).
requires(Stack,pop,{!empty()}).
```

Figure 3.2: Synchronization for a Stack in TCOOL.

```
modifies(Stack,push).
modifies(Stack,pop).
inspects(Stack,peek).
inspects(Stack,empty).
inspects(Stack,full).

mutuallyExclusive(?class,?inspector,?modifier) :-
        inspects(?class,?inspector),
        modifies(?class,?modifier).

inspects(?class,?method) :-
        modifies(?class,?method).
```

Figure 3.3: High level description of necessity for synchronization in TCOOL.

easily represent this information as a set of facts, with rules specifying how synchronization strategies in the standard TCOOL format are to be derived. This is shown in figure 3.3.

We briefly describe the semantics of the two rules in figure 3.3. The first rule states that any modifying method must be mutually exclusive with any inspective method. The second rule states that modification implies inspection. Thus the first rule implicitly specifies that any two modifying methods must also be mutually exclusive.

The modifies/inspects-declarations are a bit more coarse grained than the mutex/selfex-declarations. In classes with many variables, two methods may modify and inspect a non-overlapping set of variables, in which case there is no need for them to be mutually exclusive. The modifies/inspects-scheme is however easily modified to allow for this situation, by including the variables the methods affect in the representation[6].

3.4.1 Evaluation

We briefly discussed TCOOL showing first the basic mechanism for expressing synchronization in TCOOL. We then showed how more high-leveled representations can be built. Note that the implementation for allowing these representa-

[6]The example as originally given by De Volder already somewhat allowed for this, we have simplified the example here however.

tions did not require any changes to be made to the TCOOL weaver. Instead, rules were used to infer the basic facts representation supported by the weaver from the more high-leveled representation.

This is one of the advantages of using Logic Meta Programming to do Aspect-Oriented Programming. With some care one can build a higher leveled representation mechanism for aspects on top of a lower leveled one. One need not implement a weaver, but can rely on an existing one.

In the next chapter and in chapter 8 we will present experiments using some similar ideas. In the next chapter we use LMP to extend the basic joinpoint model provided by our own weaver to a more advanced one. In chapter 8 we use our AOP system to implement a higher leveled aspect language for the notification aspect.

3.5 Summary

This chapter introduced logic meta programming. We discussed some of the terminology involved in meta programming.

We gave two examples of logic meta programming systems: TyRuBa and SOUL. TyRuBa is an application of LMP to template based programming, while SOUL uses LMP mostly for extracting design information from programs.

We discussed an example of using TyRuBa to do Aspect-Oriented Programming. We showed a benefit of using LMP to do AOP in that users can build higher leveled representations of aspects on top of a more low leveled one without the need to change the lower leveled weaver. This idea will recur in the next chapter and in chapter 8.

Chapter 4

A logic pointcut language

We now turn our attention to the use of a logic meta programming language to express cross-cutting. We introduce a simple pointcut language similar to that of AspectJ, but embedded in a logic language, consisting of a set of primitive pointcut predicates. The logic language is the meta programming language SOUL introduced in the previous chapter. We will show how the pointcut language can be used to form pointcut expressions. We then make use of SOUL's meta programming facilities to introduce new pointcut predicates which are primitives in AspectJ.

4.1 Introduction

In chapter two we described several Aspect-Oriented Programming systems. At the end of the chapter we discussed the pointcut languages used in these systems. We classified these into three categories: simple by name referencing, general purpose language and query language. We ruled out the simple by name referencing scheme as unsuitable for an AOP system which is not tied to a specific aspect. The use of a general purpose language to determine joinpoints as is done in the Reverse Graphics system is attractive because it allows to perform arbitrary computation in the joinpoint determination process. A query language is very suitable because these typically are declarative in nature, rather than "computing" the cross-cutting structure of an aspect across joinpoints they are used to *describe* the cross-cutting structure of the aspect.

In the previous chapter we described logic programming languages, and more specifically logic meta programming languages. Logic programming languages are in effect general purpose query languages. This makes a logic language very usable as a pointcut language. It allows for arbitrary computation on the joinpoints yet has a descriptive nature which clearly brings out the cross-cutting of an aspect.

In this chapter we present a framework of pointcut predicates in a logic language. The framework is based on a joinpoint model similar to the dynamic joinpoint model employed in AspectJ. The framework itself consists of a few simple primitive pointcut predicates to reason about such joinpoints.

As the logic language for our pointcut language framework we used SOUL. The SOUL declarative framework is therefore also available in our pointcut language. This allows to combine reasoning about the class-structure of a Smalltalk application, with reasoning about its execution joinpoint structure.

The combination of the pointcut framework and the SOUL declarative framework allows an aspect programmer to easily define new, high-level pointcut predicates. This is done simply by defining new rules which compose primitive pointcut predicates and SOUL declarative framework predicates. This in effect allows the aspect programmer to extend the joinpoint model of the pointcut language. We will present some examples of such new joinpoints in this chapter.

4.2 Primitive joinpoints and pointcuts

In this section we describe the joinpoint model and the primitive pointcut predicates of our pointcut framework. The joinpoint model is a simplified variant of the joinpoint model found in AspectJ. The possible joinpoints are:

reception joinpoints: points in the execution graph where an object receives a message.

send joinpoints: the sending of a message by an object.

assignment joinpoints: the updating of the state of an object by the execution of an assignment statement.

reference joinpoints: the referencing of the state of an object by the execution of a reference statement.

block execution joinpoints: points in the execution graph where a Smalltalk block is executed.

The reason for having a more simplified joinpoint model than that of AspectJ is simply because Smalltalk is simpler than Java. While Java reifies concepts such as “object initialization” and “exception handling” by providing explicit syntactical constructs for implementing these concepts, Smalltalk does not. Smalltalk programmers rather use programming conventions such as the use of specific messages and putting methods into a specific protocol to express these concepts. In the next section we will use the SOUL declarative framework to recognize the use of such programming conventions to extend the joinpoint model.

We now describe the primitive pointcuts from our framework which are used to reason about the primitive joinpoints. Note that all of the predicates are fully multiway and can be used to generate joinpoints, as well as check whether a given joinpoint meets specific conditions etc.

We note that a joinpoint as used in these predicates is a special type of object provided by the weaver. In the description of the predicates below we always use the variable `?jp` to hold these objects. More detail will be given on these objects in the chapter on the Andrew weaver’s implementation. For now, it is sufficient to consider them as objects which can be checked for being a type of joinpoint and queried for values associated with the joinpoint through the predicates discussed below.

- `joinpoint(?jp)`
  The most basic predicate, simply expressing that `?jp` must be a joinpoint.
- `reception(?jp, ?selector, ?arguments)`
  Used to express that `?jp` is a message reception joinpoint, where the message with selector `?selector` is received with the arguments in the list `?arguments`.
- `send(?jp, ?selector, ?arguments)`
  The joinpoint `?jp` is a message send joinpoint where the message with selector `?selector` is sent and passed the arguments in the list `?arguments`.
- `reference(?jp, ?varName, ?value)`
  The joinpoint `?jp` is a reference joinpoint where the variable with name `?varName` is referenced at the time it has the value `?value`.
- `assignment(?jp, ?varName, ?oldValue, ?newValue)`
  The predicate used for assignment joinpoints, where `?varName` is the name of the variable being assigned, `?oldValue` is the value of the variable before the assignment and `?newValue` is the value of the variable after the assignment.

The predicates given above deal directly with the different types of joinpoint. As in AspectJ there are also predicates dealing with the lexical extent of joinpoints etc.:

- `within(?jp, ?class, ?selector)`
  This predicate is used to determine the joinpoints which are the executions of statements in a specific method having the given selector and implemented in the given class.
- `inObject(?jp, ?obj)`
  As all code in Smalltalk is executed in the context of an object, every execution joinpoint is associated with some object. This predicate is used to determine that object.
- `associatedJoinpoint(?jp, ?statement)`
  This predicate determines the joinpoints related to a statement. These are roughly the joinpoints that are executions of that statement, though the exact semantics depends on the statement. This predicate is to be used in combination with SOUL's rules intended to reason about statements. We will give an example further in this chapter.

4.3 Defining pointcuts

Having presented the primitive pointcut predicates in the previous section, we will now show how these are used to define pointcuts. In our AOP system, a pointcut is just a logic query about joinpoints. A simple example pointcut is:

```
Query joinpoint(?jp)
```

When this query is executed it will return a set of solutions for the variable `?jp`. These are the joinpoints that *match* the pointcut. The query can of course also be used inversely: when the variable `?jp` is bound to a specific joinpoint, the query will check whether it is a matching joinpoint. This shows the relation between logic pointcut queries and the condition system employed in AspectJ to specify pointcuts.

The example pointcut above is fairly uninteresting because any joinpoint matches it. In the following example the set of matching joinpoints is restricted to just reception joinpoints:

```
Query reception(?jp, ?selector, ?arguments)
```

The solution set of the above query will include all reception joinpoints. The `reception` predicate will also have unified the variable `?selector` with the selector of the message that is received at the reception joinpoint. The selector is a Smalltalk symbol. The `?arguments` variable is bound to the list of arguments passed with the message. The list is a SOUL list, containing the actual Smalltalk values that are used as arguments. Take the following piece of Smalltalk code:

```
result := 1 + 2.
result printOn: Transcript.
```

When this code is executed, it will lead to at least two joinpoints that match the reception pointcut above. When the first statement is executed, the number object 1 will receive the message "+" with the single argument 2. When the second statement is executed, the object pointed to by the variable result (3) is sent the message `printOn:` with the object pointed to by the variable Transcript as an argument. We show the bindings for the variables `?selector` and `?arguments` from the solution set of the pointcut for the two joinpoints described here:

```
{ ?selector -> +
  ?arguments -> <1> }

{ ?selector -> printOn:
  ?arguments -> <a TranscriptStream> }
```

Unification of unbound variables in pointcuts is similar to the wildcard mechanism employed in AspectJ, but is much more powerful. The use of the asterisk in AspectJ is a lot like the use of an anonymous variable in logic programming. In Prolog an anonymous variable is denoted with the underscore, in SOUL with a single question mark. An anonymous variable "matches anything". As each anonymous variable is unique, the value that was bound to the variable is inaccessible. With named variables on the other hand, we can still use the value further in the pointcut by unifying the variable with other variables or values. An example use of this ability is shown in the following pointcut, which contains a list of selectors for which we want to capture the message reception joinpoints:

```
Query reception(?jp, ?selector, ?),
      member(?selector, <[#printOn:], [#writeOn:], [#storeOn:]>)
```

The above pointcut will simply match any message reception joinpoint where the message being sent is one of `printOn:`, `writeOn:` or `storeOn:`. An anonymous variable is used to specify that the arguments sent with the message do not matter.

4.4 Extending the joinpoint model

In this section we will extend the primitive joinpoint model and the set of primitive pointcut predicates. The intent is to introduce new kinds of joinpoints which are provided by the AspectJ weaver but not ours. These are the instance variable initialization and exception handler execution joinpoints.

As was explained in the previous section the reason for not having these joinpoints in the primitive model provided by the weaver is that they are associated with statements in the program which depend on the use of programming conventions rather than primitives provided by the programming language. For example extracting information on exception handlers is easy for a Java weaver. Java programmers use the special try/catch syntax to express exception handling. In Smalltalk on the other hand, exception handling is done with normal message sending using code blocks which can also be used for other purposes.

In principle, the recognition of such programming conventions can be built into the weaver. The problem is however that programming conventions are open to change or personal preference, so the recognition of these programming conventions needs to be open to change as well. We believe that by building the new pointcut predicates on top of the primitives and by making use of the logic formalism and SOUL declarative framework the predicates should be easily modifiable by any programmer without having to delve into the guts of the weaver.

4.4.1 Variable initialization joinpoints

To give an example of the use of the primitive pointcut predicates and the SOUL declarative framework to extend the primitive joinpoint model, we consider the matter of object variable initialization joinpoints.

We first describe the most widely adopted programming convention for creating and initializing objects in Smalltalk. Classes are responsible for creating their instances, methods for doing so are provided by the class and are located in the method protocol called 'instance creation'. As with constructors in Java or C++ these methods take initialization arguments which are used to initialize instance variables in the instance to be created. The creation method simply does a `'self basicNew'` which creates a bare, uninitialized instance. Then, this instance is sent the same message as was sent to the class, with the same arguments, to initialize the instance. The method that is executed in response to this message is then responsible for using these arguments to initialize the object, which it simply does by assigning the arguments to the correct instance variables.

To detect the use of the above described programming convention, the SOUL declarative framework includes the predicate `instanceCreationMessage`. This predicate has the following form:

```
instanceCreationMessage(?class, ?selector)
```

Though the actual implementation of this predicate is somewhat more involved, it basically comes down to the following rule:

```
Rule instanceCreationMessage(?class, ?selector) if
        metaClass(?class, ?meta),
        selectorInProtocol(?meta, ?selector, ['instance creation'])
```

The `selectorInProtocol` predicate is a primitive predicate from the representational layer in the SOUL declarative framework. It simply represents the protocols in which methods are organized in the Smalltalk system.

Given the `instanceCreationMessage` predicate, some of the predicates described in the previous chapter and the primitive pointcut predicates, we can easily implement a predicate for determining instance variable initialization joinpoints. The idea is simply to find those assignment execution joinpoints which are the execution of an assignment statement in an instance initialization method. The rule is given below:

```
Rule initialization(?jp, ?class, ?varName, ?initVal) if
        class(?class),
        instVar(?class, ?varName),
        assignment(?jp, ?varName, ?preInitVal, ?initVal),
        within(?jp, ?class, ?selector),
        instanceCreationMessage(?class, ?selector)
```

To show an application of this rule, consider the following simple class:

```
Person class methods for protocol 'instance creation'

  withAge: age

        ^ self basicNew withAge: age

Person instance methods for protocol 'initialization'

  withAge: initAge

        age := initAge

Person instance methods for protocol 'accessing'

  age: newAge

        age := newAge
```

In the above code, the `withAge:` class method is an instance creation method, and the `withAge:` instance method is an instance initialization method. When the instance initialization method is executed, this leads to the execution of the single assignment statement within that method. This execution is thus an assignment joinpoint. This assignment joinpoint is however also a variable initialization joinpoint because it matches the conditions of the rule described above. Note that the execution of the assignment assignment in the mutator method `age:` is only an assignment joinpoint, *not* an initialization joinpoint.

4.4.2 Exception handling joinpoints

As a second example of how the joinpoint predicates can be extended, we consider exception handling joinpoints.

Exception handling in Smalltalk is done through the use of the `on:do:` message[1]. The code which may lead to an exception is embedded in a block, which is sent the `on:do:` message. The second argument to this message is another block which is the exception handler. The exception handling block takes one argument, which is an exception object. The first argument to the `on:do:` message is the class of the exception object as expected by the exception handler. Thus, when an exception object with that class is created and signaled, the exception handling block will be called with the exception object as the argument. An example is the following simple method:

```
test: x

  [ ^ 100 / x ]
        on: ZeroDivide
        do: [ :exception | ^ 'zero divide' ]
```

When the above method is called with an argument, it will return the result of dividing one hundred by that argument. Of course, when zero is given as an argument a "division by zero" exception is signaled. The method catches this exception and returns the string 'zero divide' to avoid a system error being shown on the screen.

We would now like to add to our pointcut predicates a predicate to determine those points in the execution of a program where an exception handler is executed. Since an exception handler is just a a block we already have part of the required functionality in our set of primitive pointcut predicates, namely the `blockExecution` predicate. The problem is of course to determine those blocks which are used as exception handlers.

To solve the problem of figuring out which blocks are used as exception handlers we have some options. The first option is to make use of the `blockExecution` predicate's argument list functionality to get the argument that is passed to a block and check whether this is an exception object. Exception objects must always be instances of a subclass of the class `Exception` so we can use a Smalltalk term to do the testing, the predicate becomes:

```
Rule exceptionHandlerExecution(?jp) if
  blockExecution(?jp,<?exceptionObject>),
  [ ?exceptionObject isKindOf: Exception ]
```

The problem with the above rule is that it depends on run time information, namely the exception object that is passed to the exception handler. When such pointcuts are used with the aspect weaver we will present in the next chapters, the weaver will produce fairly inefficient code. We will not get into the details of

[1]We describe here the standardized way to do exception handling in Smalltalk [18]. Before exception handling was standardized, different Smalltalk vendors provided different exception handling mechanisms. Some still support their own mechanism in addition to the standardized one.

the weaver's operation in this chapter, but the problem is obvious: when using the above rule the weaver will have to insert code in every block that takes exactly one argument to see if the argument is an exception object.

To get a more efficient `exceptionHandlerExecution` pointcut predicate, it is better to make use of more static information. The idea is to base the determination of exception handler blocks not on the argument that is passed to it, but rather on the fact that the block is created as part of an `on:do:` message. This can be determined statically from the source code, using the SOUL rules to reason about statements in methods. The actual joinpoint is then obtained using the `associatedJoinpoint` predicate.

We first show the new implementation of the rule, which we will further explain below:

```
Rule exceptionHandlerExecution(?jp, ?exceptionClass, ?exceptionObject) if
  method(?method),
  enumerateParseTree(?statement, ?method),
  equals(?statement, send(?receiver, [#on:do:],
         <variable(?exClassName), ?blockStatement>)),
  associatedJoinpoint(?jp, ?blockStatement),
  blockExecution(?jp, ?exceptionObject),
  classNamed(?exceptionClass, ?exClassName)
```

The above rule simply implements what we have described above in natural language. To be more specific: the (execution) joinpoint of any block created as the second argument to an `on:do:` message is an exception handler joinpoint. We will now describe in more detail the operation of this rule.

The `method` predicate used in the rule is one from SOUL's representational layer and gives us access to the methods present in a program and the statements in these methods. The methods are not represented as quoted code blocks but rather as parse trees using compounded terms. The representation of the `test:` method given above is shown in figure 4.1. The representation is fairly straightforward: the method's class, selector, arguments, temporaries and statements are given and combined in the form of a term with functor `method`. The statements are combined in a list etc.

The next step in the rule is to get at the message send statements, specifically those where the message `on:do:` is sent. This requires traversing the method's parse tree. This can be done with the `enumerateParseTree` predicate. Its signature is given below:

```
enumerateParseTree(?statement, ?method)
```

The predicate will simply unify the variable `?statement` with every statement from the method. We are only interested in statements of the following form:

```
send(?receiver, [#on:do:], ?arguments)
```

Picking out these statements is done at the fourth line of our rule by unifying the `?statement` variable with the particular form of statement we are looking for. The following clause is used to accomplish this:

```
method([NDRTSimpleClass14],
       [#test:],
       arguments(<variable(x)>),
       temporaries(<>),
       statements(<
          send(block(arguments(<>),
                     temporaries(<>),
                     statements(<return(send(literal(100),
                                             [#/],
                                             <variable(x)>))>)),
               [#on:do:],
               <variable(ZeroDivide),
                block(arguments(<variable(ex)>),
                      temporaries(<>),
                      statements(<return(literal(['zero divide']))>))>)>))
```

Figure 4.1: Test method represented as compounded terms in SOUL

```
equals(?statement, send(?receiver, [#on:do:],
       <variable(?exClassName), ?blockStatement>)),
```

The clause will also unify the variable `?exClassName` with the name of the exception and the variable `?blockStatement` with the statement that creates the exception handler block.

Next up in the rule is a clause to get the joinpoint associated with the block used as the second argument in the `on:do:` message. This is done using the `associatedJoinpoint` predicate. When given a block creation statement, the predicate determines all the block execution joinpoints which are the execution of that block statement.

Finally the rule obtains the exception object that is passed to the exception handling block through the use of the `blockExecutionPredicate`. The exception object is bound to the variable `exceptionObject`. The very last clause in the rule gets the exception class whose name was extracted from the first argument in the `on:do:` message.

We note that when using an imperative general-purpose language as the pointcut language as in RG, one would have to write the determination of exception handler joinpoints in a very operational manner as described here. The declarative meaning that is obvious from the implementation of this operation as a logic rule would not be as clear.

4.5 Summary

In this chapter we discussed the definition and use of our logic pointcut language. We first explained what joinpoint model underlies the language. We then introduced the primitive pointcut predicates and showed how these can be used to write pointcut expressions as SOUL queries.

The remainder of the chapter was spent showing some experiments in how the SOUL meta language in which our pointcut language is embedded can be used to extend our joinpoint model with new types of joinpoints. Defining these new types of joinpoints involved recognizing the use of certain programming conventions used in Smalltalk. We repeat an important paragraph which gives our motivation for not including the new joinpoints as primitives provided by the weaver:

> In principle, the recognition of such programming conventions can be built into the weaver. The problem is however that programming conventions are open to change or personal preference, so the recognition of these programming conventions needs to be open to change as well. We believe that by building the new pointcut predicates on top of the primitives and by making use of the logic formalism and SOUL declarative framework the predicates should be easily modifiable by any programmer without having to delve into the guts of the weaver.

Chapter 5

Aspect-Oriented Programming with Andrew

To experiment with our logic pointcut language framework, we constructed an Aspect-Oriented Programming system for Smalltalk. In keeping with our choice of using AspectJ as the basis for our pointcut language, the entire system is based on AspectJ. This allows to evaluate the effectiveness of the more advanced pointcut language. The system is named Andrew[1].

This chapter explains how Andrew is used to implement aspects, consisting of advices defined over pointcuts expressed in the pointcut language introduced in the previous chapter. We first discuss the Andrew language and user interface. We then give an example of using the system to implement a simple cross-cutting feature in a Telecom simulation.

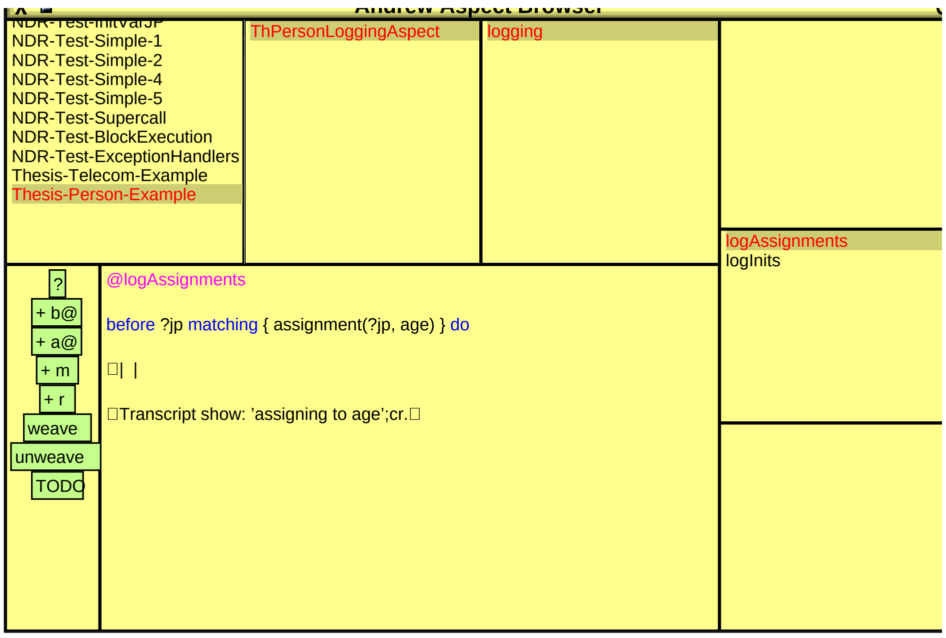
In the next chapter we will complete the discussion of the Andrew AOP system by discussing the aspect weaver which combines aspects and classes.

5.1 The language and user interface

As is the philosophy in Smalltalk, Andrew's aspect language is fairly minimal and specifying aspects is done mostly through a user interface. A screen capture of this user interface is shown in figure 5.1. The user interface is mostly based on the standard class browser in Squeak Smalltalk, of which a screen capture is shown in figure 5.2.

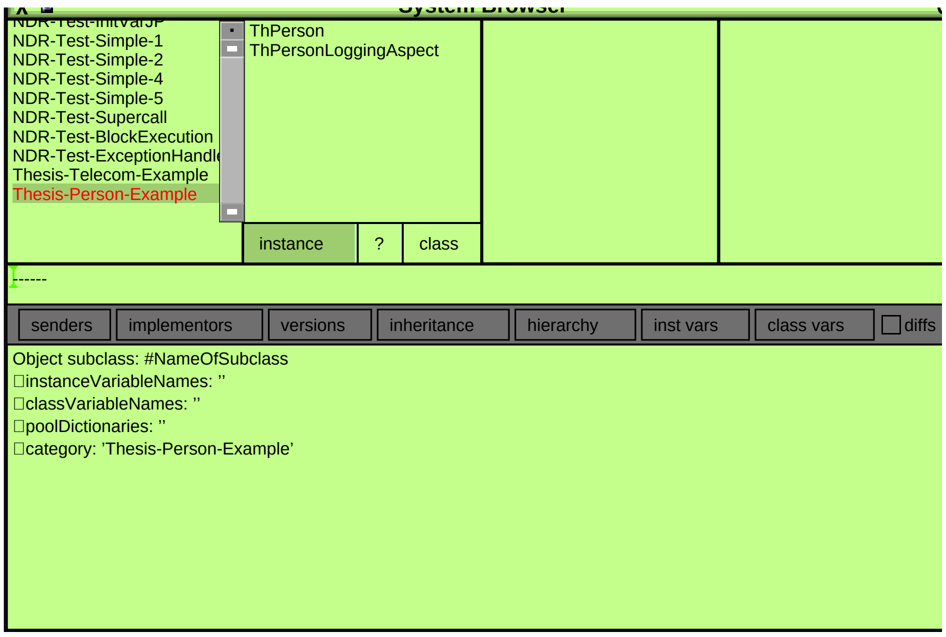
We will give a brief explanation of Andrew's user interface. The three left-most list panes at the top are: system category list, aspect list and aspect category list. These panes are similar to the corresponding panes in the class browser, but deal with aspects rather than regular classes. The three list panes on the right are similar to the method list pane in the class browser, but show from top to bottom: the rules in an aspect, the advices and the methods. The large textual pane at the center is used to edit advices, methods, rules and the

[1]The name Andrew comes from the famous short story "Bicentennial Man" by Isaac Asimov about an android wanting to be human [10], which was later turned into a novel by Robert Silverberg and also a movie (the short story is a recommended read, the novel not really and just avoid the movie which in true Hollywood style ruined the story). The name was adopted because of the association with the catch-phrase "Andrew, AOP with a SOUL".



```
@logAssignments

before ?jp matching { assignment(?jp, age) } do

| |

Transcript show: 'assigning to age';cr.
```

Figure 5.1: A screen capture of the Andrew AOP system's user interface



```
Object subclass: #NameOfSubclass
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Thesis-Person-Example'
```

Figure 5.2: A screen capture of the Squeak class browser for comparison with the aspect browser

aspect's definition. The buttons on the left serve to define new advices etc. and start the weaving process.

We now further describe the language used to define aspects, predicates and rules, advices and methods.

5.1.1 Defining aspects

Defining a new aspect is similar to defining a new class in regular Smalltalk. An aspect is created as a subaspect of another aspect by sending the latter the appropriate message. The general format of this message is shown in the following example:

```
NDRAspect subaspect: #TestAspect
        ofEach: #TestClass
        instanceVariableNames: 'x y'
        aspectVariableNames: 'Z'
        poolDictionaries: ''
        category: 'Aspect-Testing'
```

The aspect `NDRAspect` is the standard root aspect provided by Andrew, much like the class `Object` serves as the root for classes. The creation of an aspect as a subaspect of another aspect allows for the building of general aspects which can be refined. This is however not further discussed in this dissertation.

In the example, an aspect named `TestAspect` is created. Instances of this aspect will have two instance variables, named `x` and `y`. A variable common to all instances of the aspect is the variable `Z`. Finally, the aspect is put into the system category `Aspect-Testing`.

The `ofEach` modifier associates aspects with a specific class. Whenever an object of that class is created, an aspect instance is associated with that object. The aspect instance can be seen as an extension to the object holding extra state for the object. The aspect instance lives and dies with the object. Though not discussed in chapter 2, we note that the aspect instance mechanism is also available in AspectJ.

The `ofEach` modifier can also be left out, in which case a single instance of the aspect is created which is said to be associated with the entire system.

Highley et al. [30] note that writers tend to abbreviate the terms aspect instance and class instance to just instance, which can get confusing. In other cases, the term aspect is used to refer both to aspects and aspect instances, which they also find unacceptable. They propose to use the abbreviation *aspin* to denote aspect instances, just as the word object denotes class instances. We will henceforth adopt their convention.

5.1.2 Defining advices

As in AspectJ, advices specify code that is to be run before or after certain joinpoints. An example of an advice definition in Andrew is given below:

```
@logReceptions

before ?jp matching { reception(?jp, ?selector) } do
```

```
        Transcript show: 'message ' , ?selector , ' received' ; cr.
```

The advice above simply specifies the logging of all receptions of messages. The advice is accordingly named `logReceptions`, the name is specified after the at sign. The at sign is just an indicator for advices. The second line of the advice specifies its type: a before or after advice. At the second line the joinpoints for which the advice is run are also specified. The pointcut is written between braces. The pointcut is essentially a SOUL query as described in the previous chapter, but without the `Query` keyword. After the keyword `do` follows the advice body, which is written as ordinary Smalltalk code, but with the possible use of logic variables from the pointcut.

The body of an advice is executed in the context of an aspin. Thus variables in the body refer to instance variables of the aspin. Local variables can also be used and are declared using the usual Smalltalk syntax. Finally, the advice body can also refer to variables from the pointcut, using the question mark syntax. In the above example this is used to print the name of the message that is received, which is contained in the variable `?selector`.

5.1.3 Defining predicates

Aspect rules are simply SOUL logic rules, which are used to define new predicates. Predicates and rules replace the 'named pointcut' mechanism of AspectJ which are used to abstract away from complex pointcut expressions by giving these a name so that they can be used in other pointcut expressions. Note that rules are far more general than named pointcuts, for one thing: rules can be about anything, not just joinpoints.

We already discussed in the previous chapter how new pointcut rules can be defined. Rules are included in an aspect as follows[2] :

```
#addition

        addition(?jp) if
                reception(?jp, [#add:]).

        addition(?jp) if
                reception(?jp, [#addAll:]).
```

A new predicate is defined by specifying its name after an octhothorp. Then comes the body which lists the SOUL rules and facts that make up the implementation of the predicate. In the above example a new pointcut predicate `addition` is defined of which the implementation consists of two rules.

5.1.4 Defining methods

The last type of element aspects can define is the method. The methods used in aspins are not different from methods used in objects. They are defined in the

[2]The syntax for rules is slightly different from the syntax we used in the previous chapters. The new syntax is used in newer versions of SOUL. The new syntax leaves out the keywords 'Rule' and 'Fact'. We used the old syntax in previous chapters because we find it clarifying, but will adopt the new syntax here as this chapter is partially a guide to those wanting to actually try out Andrew.

aspect with the same syntax as in classes, they can access the instance variables of an aspin etc. Methods are usually used to encapsulate code that is common across advices in an aspect.

5.1.5 Aspins and no introductions

We did not include the introduction mechanism used in AspectJ in Andrew. The mechanism allows to introduce new methods and fields into classes, and change the interfaces classes implement.

We did not include the introduction mechanism in Andrew for two reasons. One is that we concentrated on the dynamic pointcut language in this dissertation. The second is that the mechanism seems to predate the use of aspins associated with objects in AspectJ. Though introductions of methods and variables can sometimes be useful, it is often cleaner to implement these in the aspect and associate an aspin with each instance of the class in which the introductions would have been done.

5.2 The Telecom Simulation Example

Figure not yet available in PDF.

Figure 5.3: Telecom simulation example class diagram

In this section we give an example of using Andrew to implement a simple cross-cutting feature. This example was based on an example given by the AspectJ team in their tutorial [29] and in a paper [39]. We chose this example to verify that Andrew can at least be used to implement aspects similar to those in AspectJ.

The example is based on a telecom simulation. Basically, the simulation consists of customers making calls to each other. Calls are established through connections, which can be local or long distance connections. Calls can be merged into conference calls so that multiple customers can participate in a conversation.

Four classes are used to implement the simulation. One is the class `Customer` to model customers. One is the class `Call` to model calls. The classes `Connection`, `LocalConnection` and `LongDistanceConnection` are used to model connections. The basic classes and the methods are shown in a simple class diagram in figure 5.3.

One feature that we want to add to the telecom simulation is to keep the amount of time customers have spent making calls, so that they can be charged for their calls. This requires to time the time span of each connection: when a connection is made, a timer is to be started, when it is dropped the timer is to be stopped. After the dropping of the connection the time spent making the call is to be added to the customer's total time.

Implementing the timing feature using normal OO programming would involve some code tangling because of cross-cutting. A new instance variable has to be added to `Connection` to hold the timer. Start and stop calls for the timer must be added to the connection making and dropping methods. Also to be added to the drop method is to inform the customer objects involved in the connection to increment their total connection time. We see that the timing feature is spread around in two classes and some methods, which leads to the problems described in chapter 2, such as the difficulty involved in removing or changing the timing feature. Some telecom companies for example do not wait with charging until the called party picks up the phone but instead start charging for a call as soon as the number is dialed in, even if the other party does not pick up. In that case the code for starting the timer would suddenly have to be added to the "start a call" method of the Customer class instead of the "connection completed" method of the Connection class.

Using Andrew we can implement the timing feature without code tangling. Instead of directly associating new instance variables with `Customer` and `Connection` to hold the total time and the timer, we will use aspins to do so. The aspect for the `Connection` class will also be responsible for starting and stopping the timer when a connection is made or dropped.

The implementation of the aspects is shown in figure 5.4.

We note that what is conceptually a single aspect has been implemented using two implementation aspects. We stress the difference between conceptual aspects and implementation aspects. The two implementation aspects can be put into a separate system category, so the category as a whole implements the conceptual timing aspect.

5.3 Summary

In this chapter we introduced an advice mechanism to extend our pointcut language to a full Aspect-Oriented Programming system. We explained the language and user interface used for implementing aspects. At the end of the chapter an example of using Andrew to implement a simple cross-cutting concern was given.

```
NDRAspect subaspect: #ConnectTiming
  ofEach: #Connection
  instanceVariableNames: 'timer '
  aspectVariableNames: ''
  poolDictionaries: ''
  category: 'Thesis-Telecom-Example'

initialize

  "initialize the timer"

  timer := Timer new

@startTimer

after ?jp matching { reception(?jp, [#complete]) } do

  timer start.

@endTimer

after ?jp matching { reception(?jp, [#drop]) } do

  timer stop.

  (ThCustomerTiming aspectInstanceFor: self associatedObject caller)
     addToTotalConnectTime: timer secondsElapsed.
  (ThCustomerTiming aspectInstanceFor: self associatedObject receiver)
     addToTotalConnectTime: timer secondsElapsed.



NDRAspect subaspect: #CustomerTiming
  ofEach: #Customer
  instanceVariableNames: 'totalConnectTime '
  aspectVariableNames: ''
  poolDictionaries: ''
  category: 'Thesis-Telecom-Example'

initialize

  totalConnectTime := 0

addToTotalConnectTime: connectTime

  totalConnectTime := totalConnectTime + connectTime

totalConnectTime

  ^ totalConnectTime
```

Figure 5.4: Implementation of the aspects of the Telecom example.

Chapter 6

Weaver implementation

This chapter completes the discussion of the Andrew AOP system. In chapter four we discussed the pointcut language used in Andrew. In the previous chapter we introduced its composition mechanisms. We now discuss the Andrew aspect weaver which composes aspects with classes. In the next chapter we discuss an evaluation criterium for the Andrew AOP system and in the final chapter before the conclusions we present an experiment.

6.1 Introduction

The general purpose of the aspect weaver is to combine aspects and classes. In Andrew, the most important part of this combination process is to make sure advices get executed at the joinpoints they describe. It also involves the creation of aspins and associate them with objects etc. This chapter concentrates mostly on the execution of advices at the appropriate joinpoints. The creation of aspins is briefly discussed at the very end of this chapter.

A particular problem we encountered in implementing the aspect weaver was whether to use an interpreter or compiler for the weaving. An interpreter is necessary because of the use of run time values in the pointcut language. This however lead to some problems with the efficiency of the weaving process. We therefore first discuss our motivation for including run time values in the pointcut language.

After the run time values motivation discussion we discuss the difference between using a compiler extension or an interpreter extension to implement an aspect weaver. We then discuss how we could have used reflection techniques to implement the aspect weaver as an extension to an interpreter. We explain the inefficiency this would cause. Then, we discuss how pointcuts can be checked for matching joinpoints using just information which is available at compile time using *partial checking*. Finally, we discuss the details of the Andrew weaver which uses both interpretative and compiler techniques to do the weaving.

6.2 Motivation for run time values

6.2.1 Original motivation

We must admit that while having become somewhat central to our thesis, the use of run time values in pointcuts has come about almost unnoticed. The original motivation for including the `?arguments` variable in the `send` and `reception` pointcuts and the similar variables `?oldValue` etc. in the `reference` and `assignment` pointcuts was to support a feature found in AspectJ which we did not discuss in chapter 2. AspectJ allows the use of run time values in advices, which are obtained from a pointcut. For example:

```
after(int amount) : receptions(int Account.withdraw(amount)) {
        System.out.println("Withdrawing amount: " + amount);
}
```

As is described in chapter 5, Andrew allows something similar by using variables from a pointcut, denoted with question marks, in advice bodies.

6.2.2 Preventing bankruptcy

The big difference between AspectJ and Andrew is the following: as the pointcut language in AspectJ can only be used to reason about or describe joinpoints, there is no way the run time value passed to the withdraw method can be used in the description of the pointcut itself, they can only be used in the advice. This is not so in Andrew.

In chapter four we used some pointcuts which used run time values. One was the `exceptionHandling` pointcut where we tested the class of the argument passed to a block to check whether the block was an exception handler. We noted that this would lead to fairly inefficient operation of the aspect weaver and that it was better to reason about the source code to determine exception handling blocks. However, a very interesting pointcut in which run time values can be useful is one to “prevent bankruptcy”.

We dig up the banking application example again from chapter 2: we would like to implement an aspect to “prevent bankruptcy”. In other words: we wish to implement an aspect to protect an account from going into the red. This requires capturing those joinpoints at which a withdraw message is sent to an `Account` object requesting a large enough amount of money to bring the account into the red. The only way we can express this in AspectJ is to put part of the pointcut in the advice:

```
before(Account account, int amount) :
  receptions(int Account.withdraw(amount)) &&
  instanceof(account)
{
        if (account.balance - amount < 0) {
                ...
        }
}
```

We find that the check for going into the red is really to be part of the pointcut. Such a pointcut can be easily written in Andrew:

```
reception(?jp, [#withdraw:], <?amount>),
inObject(?jp, ?account),
smallerThan([ ?account balance - ?amount ], [ 0 ])
```

We found that this more clearly separates advices (what to do) from pointcuts (when to do it). It however also lead to the problem that the entire pointcut is dependent on a value that can only be determined at run time. This creates some efficiency problems for the aspect weaving process which had to be resolved.

6.3 Implementation technologies

As with the evaluator for any programming language, there are basically two options for the implementation of an aspect weaver: as an interpreter or a compiler. As aspect languages are usually intended as extensions to a specific, already existing generalized procedural language the weaver will usually be more of an extension to an existing interpreter or compiler.

To extend a compiler we can either take its source code and change that, or do the extension in a more principled way in the form of a preprocessor or a precompiler. In the preprocessor approach the preprocessor somehow combines the semantics of the aspects with the part of the program written in a generalized procedural (GP) language, and then produces a program expressing the combination in the GP language. Usually this output is very similar to the kind of tangled code one would write by hand when not using AOP. The output of the preprocessor is then actually compiled into object code by a compiler for the GP language. The precompiler approach then is actually the same as the preprocessor approach. The difference between the terms is just based on whether the output of the weaver is a recognizable variant of the original program written in the GP language or is radically different. With recognizable we mean that the weaver just added some pieces of code here and there which come from the aspects. The difference between the terms preprocessor and precompiler is thus just based on whether the source transformations applied by the weaver are “simple” or “complex”, and is fairly subjective. In the latter case the compiler used by the precompiler is considered to be a high level assembler. An example of an aspect weaver that can be considered a precompiler is the RG weaver. In the case of the RG weaver the GP language taken as input by the weaver and the language output by the weaver are not the same, obviously making it a precompiler.

There are several examples of using preprocessors to implement aspect weavers. All of the examples discussed in chapter 2 use preprocessors, except for RG as discussed in the previous paragraph. The preprocessor approach to aspect weavers has become so recurring that often the term weaver is reserved to this particular type of weaver. Some authors also consider the use of source transformation a defining part of Aspect-Oriented Programming.

Because the preprocessing approach to weaver implementation is so popular, some proposals have been made towards generalized program transformation frameworks for the implementation of aspect weavers. One example was already discussed in detail in chapter 3 of this dissertation: the use of template based meta programming in TyRuBa. Fradet and Südholt [50, 26] propose writing aspects as transformations to abstract syntax trees. An aspect is written as a collection of production rules consisting of a pattern and a result. When part of the AST of a component program matches the pattern, the evaluation of the production result is inserted into the tree. When all transformations have been applied the new AST is written as output of the preprocessor.

To extend an interpreter we also have two options, again being simply to change the source of the interpreter or to use a more principled way. The preprocessor or precompiler approach can also be used for interpreters. There is however another principled way in which to extend an interpreter, which is to make use of reflection techniques. We will explain this in more detail in the next section.

Bouraqadi Saâdani also describes the two different approaches to weaving in a recent paper [47].

6.4 Reflection Techniques

We now describe a principled way in which program interpreters for OO language can be extended and how this could have been used for the implementation of the Andrew weaver. The formalism is that of Meta Object Protocols (MOP). Meta Object Protocols are related to the idea of meta programming and reflection in particular. We discussed reflection briefly in the context of SOUL in chapter 3 and summarized it in the phrase “the ability of a computational system to reason about its own program” . We can similarly describe meta object protocols as “the ability of a computational system to reason about and adapt the program of its interpreter”.

Meta Object Protocols are related to the idea of “Open Implementations”. The idea underlying “Open Implementations” is that a program (or better: a computational system) should allow itself to be extended with extra behavior. The system provides hooks at which new code can be installed without someone having to dig into the implementation of the system. Given this description it should come as no surprise that the idea of Open Implementations is a direct precursor to Aspect-Oriented Programming. The difference lies of course in how the hooks are provided: with OI the hooks are provided by the system which we want to extend while with AOP the hooks are provided by the weaver. In AOP the hooks are joinpoints, in OI the hooks are formed by the meta object protocol.

To explain meta object protocols further we consider an interpreter for Smalltalk, written in Smalltalk. As explained at the beginning of chapter 3 this interpreter is a meta program, having as its problem domain Smalltalk computational systems. One of the things from this problem domain the program of the interpreter must model is objects. Thus the interpreter’s program will include a class of which instances represent objects in the base computational system. We will call this class `MetaObject`. Objects have instance variables which can be modeled using a dictionary. `MetaObjects` provide operations for getting and setting instance variables. In code:

```
MetaObject>>initialize

        variables := Dictionary new.
```

```
MetaObject>>variableAt: name

        ^ variables at: name

MetaObject>>variableAt: name put: value

        ^ variables at: name put: value
```

Similarly, the interpreter also needs to model classes, methods etc. Together these objects and the operations they support are called the Meta Object Protocol of the interpreter. The interpreter also provides means for changing this code, for example by allowing the interpreted program to make subclasses of meta classes such as MetaObject in which methods can then be overridden. A more elaborate explanation and the technical details are beyond the scope of this work. We again refer to the work of Steyaert and others for further information [49, 43, 15].

In his PhD dissertation Bouraqadi Saâdani [15] proposed, as validation of his Smalltalk MOP called `MetaclassTalk`, the use of a Meta Object Protocol to implement aspects. `MetaclassTalk` extends the structural Meta Object Protocol that is already present in Smalltalk with an executional Meta Object Protocol.

Consider the following simple advice in Andrew:

```
@logReferences

  before ?jp matching { reference(?jp, [#balance]) } do
    Transcript show: 'accessing balance';cr.
```

Bouraqadi would implement the above advice by overriding the instance variable accessing method in the MetaObject class, as follows:

```
NewMetaObject>>variableAt: name

  ( name = #balance ) ifTrue: [
        Transcript show: 'accessing balance';cr.
  ]

  ^ super variableAt: name
```

Advices on other types of joinpoints can be implemented similarly by overriding the appropriate methods. Bouraqadi's MOP is powerful enough to handle all of the primitive joinpoints of Andrew.

The main problem with Bouraqadi's approach is that the description of how an aspect cross-cuts a program is not apparent from its implementation as an extension to a meta object. Another problem is that of efficiency, the check for the variable name which is accessed is done at *every* variable access.

Nevertheless, we could easily have used `MetaclassTalk` to implement the Andrew weaver. The idea is similar to implementing aspects directly as interpreter extensions. The execution of a statement is a joinpoint in the execution graph of a program. Whenever a statement is executed, the weaver must simply check whether there is a pointcut matching that joinpoint. To handle reference joinpoints for example, we must override the `variableAt:` method so as to check for matching pointcuts. This means binding the variable used to hold a joinpoint, which is indicated in advices, and then executing all pointcut queries to see which ones match. The problem is of course that this represents an enormous amount of overhead per execution step of the interpreter.

6.5 Partial checking

We will now show how pointcuts can be *partially checked* so as to optimize execution of a woven program. The idea of partial checking is a very simple variant of *partial evaluation*. Partial evaluation is a set of generally applicable techniques to optimize programs [32,51]. The idea is to rewrite a program when part of its input is known, so as to obtain a specialized version of the program. Partial evaluation is specifically intent on providing techniques for the rewriting so as to make the specialized variant execute faster than the original program.

When doing partial evaluation for logic programs, this involves partially checking queries. In this dissertation we only consider partial checking, we specifically do not rewrite logic queries so as to make their execution faster. The idea is rather to rewrite the woven program produced by a compile time weaver so as to make that faster, through the use of partial checking of logic pointcut queries.

The idea that pointcut queries can be partially checked is based on the fact that we use a lot of static information in them. Consider the following very simple pointcut:

```
assignment(?jp, [#test], ?, ?val), between([0], ?val, [20])
```

Obviously the execution of for example a message send statement will not match the above pointcut, only assignment joinpoints can. But not all assignment joinpoints will match either, only those that are the execution of an assignment statement assigning to the variable named `test`. The variable that is being assigned at a specific assignment joinpoint is trivially extracted from the statement in the source code that the joinpoint is the execution of. The value `?val` that is being assigned on the other hand is not trivially extracted from the source and requires execution of the program. For any assignment joinpoint assigning to `test` the matching of the joinpoint with the pointcut depends on the value being assigned being between 0 and 20.

We now see that information related to a joinpoint can be split into static information and dynamic information. Static information is trivially extracted from the statement that joinpoints are related to while dynamic information is not. In partial evaluation terms we have partial information on joinpoints available.

We extended the standard SOUL inference engine with the capability to deal with reasoning based on partial information. The extended inference engine can be used to evaluate pointcut queries when only static information is available on joinpoints. The inference process now determines whether the success or failure of the pointcut query depends on dynamic information, or whether the static information alone is sufficient to determine its success or failure. In the next section we show how this is used in our aspect weaver.

A discussion of the reasoning leading to partial checking as well as an overview of its workings is given in appendix A.

6.6 Preprocessor weaving with partial checking

We now explain the operation of the Andrew weaver. The operation of the weaver is split into two phases: a compile time phase and an interpretation phase. The idea used in this weaver is similar to that of using an executional Meta Object Protocol to implement the weaver. However, instead of having the interpreter call the weaver for every statement executed, the weaver will make itself be called only for those statements whose execution might match a pointcut. It does this by using partial checking to determine statements whose executional joinpoint might match a pointcut. This is done in the compile phase. The original program is transformed by putting pieces of wrapping code around matching statements. When interpreting the program, the wrapping code essentially makes calls to the weaver, which then checks whether the executional joinpoint associated with the wrapped statement really matches certain pointcuts. The use of the compile optimization phase allows to minimize both the amount of pointcut checking that is done at run time and the overhead associated with invoking the run time weaver which is present when using the MOP approach. In some cases the run time checking can be totally eliminated, this happens when pointcuts use only static information.

In this section we will go into further detail on the two phases of the weaver.

6.6.1 Compile time phase

Finding joinpoints

We now further describe the compile time phase of the Andrew weaver, we begin by dealing with the pointcut queries. In the compile time phase the pointcut queries reason about partial joinpoint descriptions. These partial joinpoint descriptions are in effect just statements from the component program. To be more precise: the partial joinpoint descriptions correspond to elements of the component program's parse tree which have something to do with the execution of the program such as statements, expressions and method declarations, as opposed to elements that capture structural information such as class declarations etc.

We now go into detail on the implementation of the primitive pointcut predicates. The rules implementing these predicates are split into compile time rules and interpretation rules. The reasons for this split are that the data structures representing joinpoints and partial joinpoints is somewhat different, and that the compile time rules cannot deal with run time values. We take the assignment predicate as an example:

```
assignment(?jp, ?varName, ?oldValue, ?newValue) if
  compileMode(?),
  partialJoinpoint(?jp),
  [ ?jp isAssignmentJoinpoint ],
  equals(?varName, [ ?jp variableName ]),
  dynamicValue(?oldValue),
  dynamicValue(?newValue)
```

The `compileMode` predicate is used to check which phase or mode the weaver is in. The `partialJoinpoint` predicate calls upon the weaver to generate or check for partial joinpoint descriptions. These are Smalltalk objects taken from the component program's parse tree. A Smalltalk term is used in the `assignment` predicate to access the variable name which will be assigned to. Lastly, the variables `?oldValue` and `?newValue` are declared to be dynamic values. The `dynamicValue` predicate is the primitive predicate used for the partial checking to indicate that a variable contains a (dynamic) value which is not known.

The other primitive predicates have similar implementations for their compile mode variants as the assignment predicate. In all cases the `dynamicValue` predicate is used to declare variables related to run time information as unknown. The static information is simply extracted from the statement.

We note that the implementation shown here of the assignment predicate is meant to give the reader an impression of its workings. The actual implementation we used is split over multiple rules to improve efficiency.

Producing woven code

To actually weave the program, the weaver executes all pointcut queries present in all aspects. When all joinpoints have been determined, the compile time weaver transforms the possibly matching statements into a call to the run time weaver. An example of an assignment statement and its transformation is given below:

```
test := 10 + 2

NDRAssignmentJoinpoint
   in: self
   possiblyMatchingAdvices: #(1)
   variable: #test
   oldValue: test
   newValue: (10 + 2)
```

The second statement above is the transformed call to the weaver. The NDRAssignmentJoinpoint variable refers to a class of which instances represent run time assignment joinpoints, the instances are *joinpoint objects*. When an instance is created it makes a call to the run time weaver. The transformed statement reifies the name of the variable that is being assigned etc. for use in the primitive pointcut predicate assign, which can extract this name from the joinpoint object. It also passes the old and new values of the `test` variable to the weaver. A list of advices with possibly matching pointcuts is passed as well[1]. Similar transformations are applied to other Smalltalk expressions such as the referencing of a variable, the sending of a message etc.

The run time variants of the primitive pointcut predicates simply check whether the joinpoint is of the right type and extract all values from it. The

[1]The list is passed as a literal array of numbers, the numbers are assigned to advices by the weaver

assignment predicate for example simply obtains the values for the variable name, old and new value from the joinpoint object which is an instance of NDRAssignmentJoinpoint. The run time operation of the weaver is described further in the next subsection.

Hiding woven code

One point left to address is what happens to the transformed code. Essentially the transformed code must simply be passed to the normal Smalltalk compiler, so as to obtain the final byte code form of the woven code. However, matters are a bit complicated by the fact that Smalltalk does not have a clear separation between the edit-compile-run phases of software development as is the case for most other languages. Instead the tools for these different phases all run at once in the Smalltalk system. The problem is to ensure that when the programmer edits his code, he sees the original, untransformed code that he wrote and not the code produced by the weaver. On the other hand, when executing code, the woven code must be executed.

Some techniques for dealing with the woven code problem are available. Böllert [13, 14] described the use of hidden subclasses to hide woven code. A subclass is made of a class in which woven code is to be installed, but the subclass is hidden from view by not putting it in a system category nor giving it a name, much like is the case with meta classes. When sending an instance creation message to the class, an instance of the hidden subclass will be made instead of the class itself. By putting the methods with woven code in the hidden subclass so as to override the original method in the class, the woven code will be executed instead of the original code. Another system that can be used is the method wrapper system [16] by John Brant et al. The idea used in this system can be described as "horizontal overriding" instead of the "vertical overriding" employed by Böllert. The difference between the two is that when a normal class does a supercall to a class with woven code it will invoke woven code with the wrapping system and non-woven code with the hidden subclass system. We note that with both systems, the idea was to still allow for execution of the original code.

Early versions of Andrew used the hidden subclass system. We later found that it was unsuitable for an AspectJ like weaver because of the "vertical overriding". The wrapper system seemed like a good alternative. The system we used is based on it but is much simpler because there is no need to allow for execution of the original code. It must only ensure that code editing is always done with the original code, but for the purposes of execution the woven code entirely replaces the original code. We do not go into the technical details further in this dissertation.

6.6.2 Run time phase

We briefly describe the operation of the run time weaver. In the previous section an example of the kind of code the compile time weaver produces was shown which makes calls to the run time weaver through the creation of *joinpoint objects*. For transformed statements the run time weaver takes the responsibility of ensuring the semantics of the original, untransformed statement. Essentially the run time weaver will evaluate the original statement, but will execute advice code before and after.

For pointcut queries where the successful matching of a joinpoint depended on run time information, the run time weaver will re-execute the pointcut query but with the full joinpoint description, the joinpoint object, now available. If the evaluation is succesfull, the weaver will execute the advice associated with the pointcut, otherwise it will not.

6.7 Aspin creation

For most of this chapter we concentrated on the discussion of how advices are executed at the right joinpoints, we now briefly discuss the creation of aspins. We specifically discuss the creation of aspins which are to be associated with objects, which is specified through the use of the `ofEach:` declaration in an aspect's definition.

To ensure the creation of aspins associated with objects, the weaver changes the basic instance creation methods of a class during the compile time phase. The changed methods make calls to the run time weaver whenever an object is created. For each aspect with an `ofEach:` declaration specifying the class of that object, an aspin is created. The aspins are linked to the object: they live and die with the object.

6.8 Future work

The execution of logic queries at run time is an obvious bottleneck for our weaver. Especially when relatively expensive SOUL predicates are involved. Consider the following pointcut:

```
reception(?jp, ?selector, <[0]>),
withinClass(?jp, ?class),
mutator(?jp, ?selector)
```

The pointcut captures the sending of mutator messages to objects where the value passed as an argument is 0. The run time execution of this pointcut will be limited to mutator messages because of the optimization done in the compile time phase. The only thing left to be determined at run time is whether the argument passed is zero. Yet, our current weaver will re-execute the *entire* pointcut.

Obviously our weaver needs to be optimized further to make using run time values in pointcuts more practical. A simple technique to achieve this would be to implement a caching feature in SOUL. However it would be much better to use more advanced partial evaluation techniques than what we have used so far in the form of partial checking. Partial evaluation provides techniques for analyzing our pointcuts so as to remove the parts that are not to be re-executed. There is also the possibility of compiling what is left into Smalltalk code. This would mean that the split between the run time part of a pointcut and the static part can be done automatically, achieving the same efficiency as with what is now done in AspectJ, yet retaining the coherency of a pointcut. As was shown in section 6.2 the aspect programmer has to do the split manually in AspectJ by putting constraints on run time values into the advice body.

6.9 Summary

In this chapter we discussed the implementation of the Andrew weaver. We discussed some general points related to weaver implementation: whether to extend a compiler or an interpreter and how both extensions can be done in principled ways.

We discussed the problem caused by the use of run time values in pointcuts, values which are only available at program execution time. This would seemingly require the use of a purely interpretative weaver which would be infeasible due to the amount of overhead in pointcut checking. We used *partial checking* of pointcuts to resolve this problem by using a compile time optimization phase for the weaver.

Using our approach we have made the use of run time values in pointcuts feasible. We have demonstrated our motivation for doing so with the "bankruptcy preventing" example.

Chapter 7

Evaluation

In this short chapter we present some criteria to evaluate the effectiveness of our logic pointcut language. The criteria are based on how well the pointcut language allows the aspects to be separated from the modules (classes).

7.1 The "knows-about" relationship

With the Atlas project Kersten and Murphy did a case study of implementing a web based learning environment. The system was implemented in Java and made use of AspectJ. In the report on the case study [33] they state that they often found it helpful to think about the "knows-about" relationship between aspects and classes. They define this relationship as follows: an aspect knows about a class when it names the class, a class knows about an aspect when it relies on the aspect to provide it state or functionality before it can be compiled. They identify four different cases of the knows-about relationship, which are summarized in table 7.1.

The class-directional "knows-about" relationship is the one we would most obviously expect. After all, aspects make a contribution to a program, so it makes sense that they'd be aware what they contribute to. In Andrew and

association link	flow of knows-about relationship	benefits/problems
closed	Neither the aspect nor class knows about the other	[+] easier to understand both classes and aspects [+] aspects are reusable
open	Both the aspect and class know about the other	[-] comprised understandability and reusability
class-directional	aspect knows about the class but not vice-versa	[+] classes are more reusable
aspect-directional	class knows about the aspect but not vice versa	[+] aspects are likely more reusable

Table 7.1: Classification of the different kinds of knows about relationship between aspects and classes

AspectJ this relationship occurs when a pointcut refers explicitly to a class. The class on the other hand is totally unaware of the aspect referring to it. Remember that in chapter 2 we stated that AOP provides a form of "unrequested contribution". A similar property is defined by Filman [24], who states the *obliviousness* property: the class affected by an aspect is unaware of the affecting[1].

The open relationship is different from the class-directional one in that the class also knows about the aspect. In the telecom example from chapter 5 an example of this situation could arise as follows: suppose we define a print method on the Customer class, printing a customer's name, phone number and total time spent making calls so far. The latter would require accessing the customer object's aspin for call time keeping. This would reduce the applicability of aspects to "requested contribution", the timing feature is an inherent part of the simulation which can not be removed as the class depends on it being present. However, aspects still help in cleanly separating the timing feature from the basic simulation, allowing it to be easily replaced with a different way of timing. Note that in the example of this relationship given by Kersten and Murphy the linking is more intricate and leads to diminished understandability of the code, which is not the case with our simpler example.

The aspect-directional relationship is the inverse of the class-directional relationship. We do not discuss this relationship further.

The closed relationship is the most desirable. It upholds the "unrequested contribution" property and furthermore makes aspects *loosely coupled* with the program. This loose coupling is a very desirable property because it potentially makes aspects *reusable*. The aspects can be made to cross-cut a general modular structure rather than a specific instantiation of that structure. Kersten and Murphy implicitly identify AspectJ's wildcarding mechanism as the key tool to achieve such a loose coupling. In general, to allow for a closed "knows about" relationship a pointcut language must allow generalizations to be made.

The "knows about" relationship can also be defined for methods, instance variables, packages (Java), system categories (Smalltalk) etc. instead of just classes. Again the relationship between these kind of modules and an aspect is directional with respect to a module if the aspect explicitly refers to the module by name or closed if it does not. The same points made above apply.

7.2 The enumeration problem

In studying AspectJ and related papers we came across a problem which is most obviously stated by Lippert and Lopes [38]. We cite:

> In using an aspect for implementing this post-condition we also need the hooks into the methods that return an object. AspectJ 0.4 does not provide a designation mechanism for expressing all methods that return an object reference concisely. We had to list them one by one.
>
> But, again, AspectJ 0.4 does not provide a designation mechanism for expressing all methods that take two object references concisely. We had to list them one by one.

[1] We are unsure yet whether the "unrequested contribution" and "obliviousness" properties are similar or rather complementary.

We have dubbed this problem the “enumeration” problem. If the pointcut language used is not powerful enough to express some cross-cutting pattern, one has to resort to enumeration of occurrences of that pattern. This clearly leads to a module-directional “knows about” relationship between aspects and modules and makes aspects less reusable.

We note that the comments of Lippert and Lopes refer to AspectJ version 0.4 which is an older version of AspectJ than the one we described in chapter 2. We have not verified but the cross-cuts they discuss might now be clearly expressed in AspectJ without enumeration, due to the adoption of the wildcarding mechanism in pointcuts.

7.3 The Andrew pointcut language

Again we have identified an area of benefit for the use of both a logic language and meta programming for the expression of cross-cutting. The use of LMP to express cross-cutting allows for reusable aspects by maintain a closed “knows-about” relationship:

- A logic language has the powerful pattern matching technique of unification.
- The meta language can be used to express a pattern of messages existing between classes, a pattern of implementation in a method etc.

We discuss some simple but convincing examples.

Pattern matching on the name of a message is not enough in Smalltalk to capture receptions of mutator messages. In Java this is usually possible because there is the convention of having the name of a mutator method begin with “get”. Thus one can use name pattern matching to capture such messages: “get*”. In Smalltalk however this is not possible because there is no such word shared by all mutator messages. One would have to resolve to simple enumeration of all the mutator messages. We have already shown however how the `mutator` predicate of the SOUL declarative framework can be used to recognize the use of the mutator method programming convention in a method. Thus we can use this meta predicate to capture the reception of mutator messages, avoiding the enumeration problem. The predicate is also much safer because it infers the mutator message property from the actual source of a method rather than from its name or location in a specific message protocol.

Another example is the capturing of accept messages associated with a visitor in the Visitor design pattern. Again it is possible to do name matching: “accept*”. At least if one assumes the methods will be called “accept”, they might also be called “visit”. The `visitor` predicate from the SOUL declarative framework can be used to detect the accept methods of the visitor pattern. This example involves reasoning about inter-class structures.

More complex examples should probably be thought of, but this falls out of the scope of this work. We find however that there is great promise for LMP in coping with the reusability of aspects by having aspects specify cross-cutting patterns with more complex mechanisms than simple wild card pattern matching.

7.4 Summary

In this short chapter we discussed the applicability of expressing cross-cutting using LMP with respect to the possibility of achieving a loose coupling between aspects and classes, methods etc. We gave a classification of the kind of coupling that can arise in the discussion of the "knows-about" relationship. We briefly discussed the "enumeration problem" and its effect on loose coupling. We argued why our approach to AOP can be beneficial. This however is an area requiring further investigation.

Chapter 8

An Experiment

This is the last chapter before the conclusions. In this chapter we discuss in detail an experiment we performed using our AOP system: "Observing with Aspects".

8.1 The problem

The Model/View/Controller (MVC) concept is well established in Smalltalk, from which the concept originated. MVC is a design principle stating that data modeled by a program and user interface representations of this data are two entirely different things and handling them should be done separately in the program's source. In other words, MVC promotes the separation of the modeling and user interface representation concerns. The MVC principle is also better known in a slightly different form as the Observer design pattern [27].

When following the MVC principle, one uses separate objects to implement models and views. A model captures data from the program's problem domain, while a view captures the UI representation of this data. Of course, whenever the model is changed, its view must be updated as well. A model must thus notify its view of modifications made to itself. It does so by sending a message to its view. Usually it is more interesting to have several views on a single model, so this must be allowed for as well.

To support the implementation of applications following the MVC principle, Smalltalk provides the dependency mechanism. This mechanism is provided by the `Object` root class and is thus available with every object. When one object is interested in changes to another object, it makes itself a dependent on the model object by sending it the `addDependent:` message using itself as argument. Conversely, whenever an object changes it sends itself the message `changed:`. The method for the `changed:` message is implemented in `Object` and simply sends a notification message to every dependent. The notification message is `update:`. The single argument involved in the `changed:` and `update:` messages is used to represent what of the model object changed. The convention is normally to pass the selector of the accessor method which is used to access the instance variable that was changed.

It appears that doing MVC with the use of the dependency mechanism separates well models and views. There are some problems however. We already described one problem in chapter 2 in the section on Design Patterns. The problem is that views require models to send notification messages. When a view is interested in particular variables of a model, but the model does not send notification messages for changes to these variables, the view will not get updated properly. Thus objects should send notification messages for every change to a variable. Taken to the extreme, this would mean that *every* object in a Smalltalk system should send the `changed:` message to itself for *every* change. This would clearly present a serious amount of overhead as most objects will likely not have any dependents at all. For objects that do have dependents, only one of them might be interested in a particular change and all the others will be notified unnecessarily.

Another problem that occurs with the dependency mechanism is that of when to actually send the `changed:` message. Brichau et al. noted that whether or not a change is to be notified may depend on the usage context of a method. They called this problem the jumping aspect problem [17]. To illustrate the problem they described it using a `ListModel` example:

```
ListModel>>add: element
  elements at: index put: element.
  index := index + 1.
  self changed: #elements.

ListModel>>addAll: elementsCollection
  elementsCollection do: [ :element | self add: element ]
```

The problem with the above version of the `ListModel` is that it will lead dependent objects to be notified of a change too frequently. When a whole collection of elements is added at once, the dependents will be called for each element added. A `ListModelView` for example would then change its screen representation each time. This leads to a very slow, and jittery user interface. It would be much better if the change notification was deferred till all elements have been added. This can be solved by moving the sending of the changed message to the `addAll:` method. However, this would mean that when a single element is added directly, the dependents would not get notified, which is clearly also undesirable.

Several solutions can be used to solve the above described problem. One is to use some state variable to indicate whether the changed message must be sent. Another is to split in the `add:` method into two methods: a private method which does the actual addition of a single element without doing any notification and one which can be used from the outside which calls upon the private method and then does change notification.

The problems we described with the dependency mechanism should make clear that the modeling and UI representation concerns are not as well separated as they at first appeared to be. The problem is that in order to provide good change notification, a model is to be made aware of how its views will be using it. Changes of no interest to any view should not be notified and notification must be optimized. Thus knowledge of views "leaks" into models. The sending of changed messages is also an example of code tangling. This tangling is minimal for the simple variant of the mechanism, but the optimized variants require adding variables or splitting methods which introduces more code tangling.

8.2 AOP solution

Aspect-Oriented Programming is a good candidate for dealing with the problem of change notification. The hooks in this case are the notification sends. Instead of having models supply the "hooks" to which views can attach themselves, a weaver can be used to provide these hooks. An AOP solution would thus consist of having views describe the changes in a model they are interested in and the weaver making sure the view gets notified of these changes. This solves the problem of having a model sending too many `changed:` messages which would induce overhead. A well designed weaver should also recognize the need for deferring notifications.

To our knowledge, there is currently no aspect language and weaver available to deal with the specific problem of separating views and models properly. The general aspect model provided by AspectJ and Andrew is however very suited to dealing with this problem. The AspectJ documentation provides an example of how to handle the problem [1].

We first discuss the approach taken in the AspectJ example of separating notification from the model [1]. In the example a general aspect `Subject` is used which deals with the basics of the notification protocol. Each aspin of this aspect is associated with an object for which it holds the dependents and it makes sure these get notified upon changes in the object. The `Subject` aspect is an abstract aspect, a specialization is needed for every specific class of models that is to provide notification. The specialization is required to fill in an abstract named pointcut which is to enumerate the message reception joinpoints after which notification is to happen. An after advice declared in the `Subject` aspect makes use of this pointcut to do the notification.

We note that the AspectJ example concentrates on separating the notification from the model without taking into account the problems we discussed above. The idea is that the specialization of the Subject aspect is provided by the programmer of the model. Thus it does not take into account that dependents might not at all be interested in particular changes. We will therefore take a different approach here, associating the description of the notification with the view rather than the model.

8.3 Notification in Andrew

8.3.1 Notifying on assignments

We consider first a simple case of a view and model. Suppose a view class `PersonsAgeView` is used to show a list of people with their name and age. People are modeled with the class `Person`. This class has at least two instance variables, `name` and `age`, for which it provides accessor methods which the view uses. The `Person` class may of course have many more instance variables, such as the address of a person, the phone number or a list of children. But these are of no interest to the view we consider.

The simplest way of having a view express an interest in changes to a model is to have the view monitor assignments to the model's instance variables. In our particular example the variables of interest are name and age. The monitoring of these variables in `Person` objects is readily expressed as a pointcut:

```
member(?varOfInterest, <age,name>),
assignment(?jp, ?varOfInterest),
withinClass(?jp, [Person])
```

Note that an assignment to a variable does not necessarily mean a change has occurred, as the value assigned might not differ from the value that was in the variable. At the cost of some run time weaving, we can include this check in the pointcut:

```
member(?varOfInterest, <age,name>),
assignment(?jp, ?varOfInterest, ?oldValue, ?newValue),
not(equals(?oldValue, ?newValue)),
withinClass(?jp, [Person])
```

There are some obvious problems with the assignment monitoring approach. One is that classes are not supposed to be aware of each other's instance variables. Furthermore, the values the view obtains from the model may not come at all from instance variables directly, but may be derived properties: instead of having a `Person` object hold a person's age, it might hold his date of birth. The age of a person is readily derived from his date of birth and today's date. Replacing our `Person` class with a new one using the date of birth scheme should in normal OO programming not break our system, yet it will break our pointcut.

Another problem is that instance variable monitoring does not handle deferred updates. Instead, using an after advice on the above pointcut which does the notification, dependents are made aware of a change as soon as it happens. This is even faster than is normally the case when notification is done through the traditional `changed:` message mechanism. With that mechanism, the sending of the `changed:` message is usually not done until the end of a method. One reason for this is that it minimizes code tangling by having the sending of the message "stuffed away" at the end of a method, separating it a bit from the method's normal functionality. Another reason is that a method may do assignments to many variables, with the object being in a broken state until all the assignments have been done. This is clearly a problem for our current pointcut and advice as well.

The last problem is that assignments are not the only way of changing an object. The list of children of a `Person` will not be changed by assigning to the variable "children" but rather by sending addition or removal messages to the list contained in the variable "children". To deal with this we might use a second pointcut similar in spirit to the first, but monitoring message sends rather than assignments. A simple implementation would be:

```
send(?jp, ?selector),
messageModifies(?selector)
```

The main problem with the above pointcut is of course in determining which messages will change an object and which won't, which is to be checked by the `messageModifies` predicate. We will get back into this further in this section.

8.3.2 Removing unnecessary notification

Instead of using the assignment monitoring approach we can 'remove' `changed:` messages from methods. As was described at the beginning of this section, the main problem with these messages is that sending them when no view is interested in the particular change being notified they represent nothing but overhead. With the assignment monitoring approach described before we tried to add notification only if it was necessary. What if we try to do it the other way around: remove notification when it is unnecessary?

Removing a message send can be done using an around advice with an empty body. As was explained in chapter 2 in the context of AspectJ, an around advice can totally replace a joinpoint in the execution graph. Replacing it with "nothing" thus removes it. Unfortunately, the current Andrew weaver does not provide around advices. We do not expect these to be too problematic to add however, so the remarks made here are still valid.

Expressing the joinpoints that need to be removed is done with a very simple pointcut:

```
send(?jp, [#changed:], <?changedProperty>),
within(?jp, [Person]),
not(viewInterested(?changedProperty))
```

The viewInterested predicate is easily implemented as a set of facts provided by views about which properties of instances of the Person class the view is interested in. A property is just a selector the view uses to get some value from a Person object to represent on the screen. For example, for the PersonsAgeView we would have to provide these facts:

```
viewInterested([#age]).
viewInterested([#name]).
```

Using the simple approach outlined above we have successfully removed unnecessary `changed:` messages, reducing the amount of run time overhead they incurred[1]. . We note that determining the value of `?changedProperty` can be done in the weaver's compile time phase. The argument passed to the `changed:` message is always a fixed literal, the Andrew weaver is advanced enough to recognize this. . There is therefore no run time weaving overhead.

8.3.3 Declaratively expressing notification joinpoints

Our removal approach works a lot better than our previous addition approach, there is however still the problem of the code tangling represented by the `changed:` messages. Though such a single line of code would not affect much of the "ilities" such as readability, there is still the problem that these messages represent some knowledge which is not kept together but rather spread around the different methods of a class. The `changed:` messages do not actually represent something functional, but they are very declarative knowledge expressing the execution joinpoints at which notification is needed for a change to some property. In the spirit of TyRuBa and TCOOL, we would like to represent this knowledge separately not as code, but as a set of logic facts.

Taking into account the fact that `changed:` messages are normally put at the end of a method, we can represent the knowledge they express as facts in the following form:

[1]Because of the lack of a full fledged around advice mechanism we added a simpler remove advice to Andrew

```
notify(?class, ?methodSelector, ?propertyChanged)
```

The ?methodSelector is the selector of the method which does a change, the ?class is the class the method is implemented in and ?propertyChanged is the property that the method changes as before.

Doing notification is now again a matter of adding the notification rather than removing it. We do notification for a property after the reception of a message which will change the property. The pointcut determining these reception joinpoints is as follows:

```
reception(?jp, ?selector),
withinClass(?jp, [Person]),
notify([Person], ?selector, ?property),
viewInterested(?property)
```

Providing the notify facts does not represent more work for a developer of a model class than does the providing of changed: messages. This work is not always that simple, as it requires tracing the dependencies between methods providing access to a property and instance variables. As described, we consider a property of a model used by a view some value returned by a method of that model, which can be the value of an instance variable or some derived value as can be the case with the age method of the Person class. Among the notify facts of the Person class would be these facts:

```
notify([Person], [#birthdate:], [#birthdate]).
notify([Person], [#birthdate:], [#age]).
```

The second fact required the implementer to take into account that age is derived from one's date of birth and thus that a change to the birthdate instance variable, done by the mutator method for this variable, would change the age property. Instead of putting this burden on the model implementer, we can have him provide more low level information. Namely the variables a method affects, and the variables it depends on. For the age, birthdate, and birthdate: methods this would become:

```
modifies([Person], [#birthdate:], [#birthdate]).
dependsOn([Person], [#birthdate], [#birthdate]).
dependsOn([Person], [#age], [#birthdate]).
```

The notify facts can then be derived from these lower level facts using these rules:

```
notify(?class, ?modifyingSelector, ?propertySelector) if
  modifies(?class, ?modifyingSelector, ?variable),
  dependsOn(?class, ?propertySelector, ?variable)
```

8.3.4 Automatically determining dependencies

Instead of having the implementer of a model class provide the information on modifications and dependencies of a method, we can derive this information from the source. We first note that this is theoretically impossible. Determining whether a program will change a specific variable is a problem reducible to the halting problem and therefore impossible to solve in general. Of course, the result also applies to humans. A developer also uses some incomplete heuristics to determine where to place `changed:` messages, or to determine the `modifies` and `dependsOn` facts. These heuristics can therefore be programmed.

In our experiment we implemented some simple rules to derive the modifies and dependencies information from source code. Essentially the rules look at variables referenced or assigned by a method. The rules also look at the use of accessor methods, and the sending of methods that modify an object which is done by recursive application of the rules. The rules have been tested to work on some simple classes. Of course, better implementations should be made to deal with more advanced classes.

8.3.5 Application to TCOOL

The automatic determination of variable modifications and dependencies of methods is relevant to TCOOL as well. TCOOL was explained in chapter 3. As was explained, De Volder showed how the necessity for synchronization code could be expressed in a high level form as a set of facts expressing which methods inspect an object and which modify it. Instead of having the programmer provide these facts, they can readily be derived from the `dependsOn` and `modifies` facts: [2]

```
TCOOLinspects(?class, ?selector, self) if
        dependsOn(?class, ?selector, ?variable).

TCOOLmodifies(?class, ?selector, self) if
        modifies(?class, ?selector, ?variable).
```

8.3.6 Deferred updates

In our experiment we have not yet dealt with deferred updates problem. This is however a more minor problem to tackle. A solution consisting of keeping track of whether a message was sent to an object from outside or as a self send should suffice. The notification can then be deferred to the end of the method invoked by the message sent by another object.

8.4 Evaluation

We have presented here several ways of dealing with the notification of views when decoupling views from models, each more advanced than the previous. We note that we have used both dynamic and static mechanisms. Notifying upon assignments is a very dynamic way of doing notification as notifications are really done when something changes. The other mechanisms used mostly reasoning about source code to try to determine whether a method *might* change an object. The use of if-tests and assignments assigning the value to a variable it already had may make that an object is not really changed by that method. Which is best depends on the situation. Some more research should go into this topic.

[2]We replaced the Java this with the Smalltalk self

The rules used to obtain the `dependsOn` and `modifies` facts essentially form a weaver, with the facts being the joinpoint structure the weaver provides. The rules reason about the source code to extract a joinpoint structure consisting of depends and modifies information. A more high leveled representation was then extracted through the use of the `notify` rules. The facts derived from these rules then essentially form notification joinpoints.

Views can be woven with models through the notification joinpoints by having the views express an interest in the properties that are notified at those joinpoints. This weaving can be brought back to the weaving provided by the Andrew weaver by defining an advice over message reception joinpoints. The notification joinpoints are related to reception joinpoints because the `notify` facts are expressed in terms of the properties that are changed after the execution of a method.

We have thus given another example of the application of De Volder's result in that LMP can be used to build user-defined weavers on top of a more basic weaver. The differences are however:

- We used active source code reasoning to determine a suitable joinpoint structure.
- The basic weaver is more high-leveled than the one used in TCOOL. In TCOOL the basic weaver is the source transforming mechanism of TyRuBa which expects a weaver built on top of it to do source code weaving. The basic weaver then only takes care of writing out the transformed program as Java source. In our example, the basic weaver is the Andrew weaver which provides executional joinpoints. The change notification weaver we built on top of it does its weaving by putting advices on these joinpoints, actual source transformation is left to the Andrew weaver.

The main problem we encountered in this experiment lies in the sheer size of the Smalltalk system. In order to make the derivation of the `modifies` facts from the source feasible we had to manually indicate the facts for some often used messages in Smalltalk such as `add:`, `addAll:` etc. Otherwise the rules for deriving the `modifies` facts would go recursively into these methods. This process would have probably eventually simply ran out of memory before ending.

In conclusion we can state the the notification aspect is an intriguing one to tackle but that more research is needed to do so effectively. At least if one wishes to use the method based on deriving the `modifies` facts from source. Using actual run time bookkeeping of changed instance variables in objects is also possible but inefficient.

Chapter 9

Conclusions

9.1 Summary

In this dissertation we set out to discuss the benefits and feasibility of using a logic meta programming language making use of a dynamic joinpoint structure to express the cross-cutting of concerns in order to achieve a better separation of concerns.

In chapter two we set the stage for the discussion by explaining what we want to achieve. The concept of separation of concerns was introduced. It was explained how cross-cutting concerns come about in today's software development practices, and how they affect code quality attributes by making separation of concerns difficult. We then described some approaches to dealing with such concerns. We paid particular attention to Aspect-Oriented Programming.

Chapters three, four and five were mostly spent on introducing our approach to Aspect-Oriented Programming. Chapter three presented logic meta languages. In chapter four the SOUL logic meta language and framework was extended with new predicates so as to form a pointcut language. In chapter 5 the composition mechanism of Andrew was presented which can be used to compose advices with normal Smalltalk code at joinpoints specified by a pointcut.

Chapter six explained some of the design decisions involved in the construction of the Andrew weaver and some details of its implementation. We specifically wanted to allow for the use of run time values in pointcuts, which requires execution of the Smalltalk program to obtain them. It is however not feasible to have the weaver operate entirely as an interpreter extension since this would require pointcut checking at every execution step. We showed how a simple pointcut analysis technique inspired by partial evaluation techniques can be used to "weed out" pointcuts during a compile time optimization phase. When actually executing the Smalltalk program, the weaver is only invoked at execution steps (joinpoints) which could match a pointcut. Run time checking of pointcuts is only done for those pointcuts that depend on dynamic values.

Chapter seven presented an evaluation criterium for pointcut languages which lies in their ability to make an aspect reusable. This requires some abstractional capabilities in the pointcut language, mostly in the form of pattern matching. Pattern matching is also essential for the description of cross-cutting to avoid the "enumeration problem": an aspect typically cross-cuts at some joinpoints which have something in common. So far simple wildcarding mechanisms have mostly been used to allow for pattern matching, but unification as used in logic languages is much more powerful. We also discussed the applicability of the meta constructs in our pointcut language to achieve loose coupling of aspects and modules.

Experiments illustrating the use of our pointcut language and composition mechanism were shown in chapters four, five and eight. In chapter four we extended the primitive joinpoint model provided by our weaver with new types of joinpoints. This consisted of using SOUL's meta reasoning capabilities to find initialization methods and exception handler blocks. In chapter five the telecom simulation example was given, mostly to show the use of aspins and as a validation of the similarity between our system and the AspectJ system it is based on. In chapter 8 a detailed discussion on how to use aspects to implement notification handling in MVC was given.

9.2 Conclusions

In the introduction we broke our original thesis statement into three core parts:

1. The use of a (Turing complete) logic language
2. The use of a meta language
3. The use of dynamic joinpoints, with reification of dynamic values in the pointcut language

We have shown that the use of a Turing complete language in combination with reification of dynamic values in the pointcut language allows to write more advanced domain specific pointcuts. We showed how this can be used to describe a pointcut such as "when the balance of a bank account would go into the red (below zero) because of a withdrawal". This required the use of a run time value to infer whether the balance would go into the red or not. The example showed that it is possible to express such a pointcut using our approach. We have shown how this approach can be made feasible through the use of partial checking of pointcuts at compile time in the aspect weaver.

We have shown how a meta language can be used to recognize patterns in a program and how this benefits the description of cross-cutting. A demonstration was given in recognizing the use of programming conventions in a flexible language such as Smalltalk. The particular examples we gave were recognizing the use of the programming convention for instance variable initialization and the one for using blocks as exception handlers. We were able to extend the primitive joinpoint model provided by the weaver with new user defined joinpoints: instance variable initialization joinpoints and exception handler execution joinpoints. The fact that these are user defined joinpoints gives rise to an open language, allowing easy addition, changing and inspection of joinpoint definitions.

We explained how the recognition of patterns also aids in obtaining a loose coupling between aspects and classes and their elements. This is possible because the logic language's powerful pattern matching and inference mechanisms allow to describe the pattern of classes, methods etc. an aspect cross-cuts in terms of joinpoints rather than a particular instantiation of that pattern in a program. Furthermore the use of a *logic* language leads to a clearer expression of *what* the pattern is rather than *how* it is recognized as would be the case when an imperative language is used.

Finally in chapter 8 we undertook an experiment showing how advanced reasoning in LMP can be used to detect the interdependencies between methods which occur through the changing and use of instance variables. This allowed us to separate the notification aspect from a model class and a view class.

9.3 Technical contributions

During our work on this dissertation the two most important technical contributions we made are the creation of an Aspect-Oriented Programming system for Smalltalk and a partial inference system for SOUL. While the Andrew system is still somewhat experimental it is usable to further explore some of the issues in doing AOP for Smalltalk. The Andrew system comes with a simple, but easy to use graphical programming interface. With some extra work it should be possible to perform some larger case studies. Our second contribution is the creation of a partial inference system for SOUL. This system is available as a meta interpreter, but also as a direct extension to the SOUL inference engine written in Smalltalk. The latter version is the one used in Andrew.

9.4 Remarks and Future work

There is still lots of room for extension of the topics discussed in this dissertation.

We already noted at the end of the chapter on our weaver's implementation that the partial evaluation technique used for optimization is too simple. The problem lies in the rechecking of *entire* pointcut queries at run-time instead of just the parts that depend on dynamic values. A great deal of work has already been performed in the area of partial evaluation techniques for Prolog like languages, but we would need to research this further in order to construct a more advanced static analysis tool for our weaver. The tool would need to perform *program specialization* which is at the heart of partial evaluation. Pointcut queries need to be rewritten so that only the parts that need to be checked at run time are left.

Speed is not only an issue for the run time weaving process, but also for the compile time optimization phase itself. Some of the more advanced predicates in the SOUL declarative framework can take a long time to infer. As any change to a program might require it to be rewoven with aspects this can be a serious problem during development. Dealing with this problem will probably prove to be very difficult.

It has been noted to us that the use of an explicit weaving step conflicts with the nature of Smalltalk. As can be seen from the screenshot shown in chapter five, a programmer has to explicitly invoke the weaver by pressing the "weave" button. At the time of writing De Alwis is working on an AOP system based on AspectJ for Smalltalk as well [9]. He is concentrating his research on the structures and support needed for maintaining Smalltalk's incremental nature, such that changes are applied immediately. Whenever a method is added it should be immediately woven with the aspects that affect it. In principle it would be enough to run the weaver's compile optimization phase but restricted to that method. The problem is of course that due to the use of meta programming, the newly added method may indirectly have an effect on the woven code of other methods as well. Consider the somewhat idiotic pointcut capturing "all reception joinpoints sent to objects of class `Bla` when class `Bleh` has more than 10 methods". This pointcut will never require the insertion of weaver calls in the methods of class `Bleh`, yet the addition of any method to that class might suddenly require weaver calls to be inserted in methods of class `Bla`. It seems very difficult to deal with this in any other way than as to simply apply weaving to all code at the addition of a method instead of just the added method. This is fairly unacceptable because of the overhead it imposes.

So far the number and size of experiments we have performed with our logic pointcut language is limited. It would be interesting to perform or have perform a case study in how it can be used in larger systems.

As was indicated at the end of chapter eight, we have not entirely tackled the notification aspect yet. More research is needed into this topic as well.

Appendix A

Partial checking

This appendix gives a more detailed explanation of partial checking. Partial checking was introduced in the main text in section 6.5. This discussion first explains why the standard inference mechanism of SOUL, and Prolog in general, is insufficient to deal with partial information and so why an extension to the SOUL inference engine was needed. The mechanism underlying partial checking is then briefly explained.

In the main text the following example pointcut was given which is also used in this appendix as an example:

```
assignment(?jp, [#test], ?, ?val), between([0], ?val, [20])
```

The question is now how we can use partial information on a joinpoint to determine whether a specific pointcut could match that joinpoint. This also brings up the question of what is to be the meaning of the primitive pointcut predicates when applied to a partial joinpoint. Consider a joinpoint for which we know that it is an assignment joinpoint, assigning to a variable named `test`. In the following part of a query this (partial) joinpoint is bound to the variable `?jp`, while the variables `?varName` and `?val` are not yet bound:

```
assignment(?jp, ?varName, ?, ?val)
```

Obviously, the predicate will bind `?varName` to the value `test`, but what binding will it provide for `?val`? To preserve the multi-way property of logic programs, it should in theory generate all possible values. This is clearly infeasible. The next best option would be to simply leave `?val` unbound. At first sight this seems to be a good option. Consider again the pointcut example given earlier, the one involving between, applied to the given partial joinpoint description. The `assignment` predicate will leave `?val` unbound, but the `between` will bind it to every number between 0 and 20. Thus the query essentially generates all the values that might be assigned at an assignment joinpoint to the variable `test` in order for the joinpoint to match the pointcut.

Leaving variables unbound in the primitive pointcut predicates when applied to partial joinpoint descriptions does not always lead to the desired behavior. Consider a variant on the pointcut involving between:

```
assignment(?jp, [#test], ?val), not(between([0],?val,[20]))
```

When using this query again with `?jp` bound to a joinpoint which is only known to be an assignment joinpoint assigning to variable `test`, this query would fail. However, when we extend the partial joinpoint description with the information that the value being assigned is 40, the query would succeed. This is undesirable because it would lead our weaver to dismiss joinpoints incorrectly.

Besides `not`, other meta predicates also cause problems. The `var` predicate for example is another obvious candidate for trouble.

The problem with leaving variables unbound is that in Prolog-like languages this does not signify "value unknown", but rather "give me the possible values". In order to deal with partial information, we had to extend the SOUL interpreter to allow variables to be marked as having an unknown value. Furthermore, queries no longer simply succeed or fail, but can also be undetermined. When a query is undetermined this means the success or failure of the query depends on an unknown value. The semantics of meta predicates was changed to be able to deal with flagged variables. We have taken the semantics of an unknown value to be that, when the value is known it will be a fully grounded value, which also gives the semantics for the `ground` and `var` predicates when applied to an unknown value. The `not` predicate then succeeds or fails as normal respectively when the query it negates fails or succeeds, when the subquery is undetermined the not will also be undetermined.

The semantics of unification is of course also affected by the unknown values mechanism. Again, unification no longer succeeds or fails, but can also be undetermined. When an unknown value is unified with another value, the unification is undetermined unless the other value is a variable which is not flagged as unknown. Furthermore, any variable involved in the unification with an unknown value also becomes unknown. An example unification and its result when `?x` is unknown and `?y` and `?z` are yet unbound:

```
equals(?x, test(?y, [7], bla(?z)))

[?x --> (unknown)
 ?y --> (unknown)
 ?z --> (unknown) ]
```

Finally, we describe the effect of the unknown values mechanism on SOUL's symbiosis mechanism. This is also fairly straightforward: when a Smalltalk term is used as a clause and is applied to any SOUL variable which is unknown, the term will not be executed by the interpreter. Rather, the execution is taken to be undetermined. When the Smalltalk term is used as a term and applied to something unknown, the value of the term is taken to be unknown.

We note that the unknown values mechanism introduces a reasoning based on ternary logic into SOUL. When combining clauses using the and operator (the comma), the rules for the reasoning are as follows: when either part of the and fails, the clause as a whole fails. When neither part fails but one is undetermined, the clause as a whole is undetermined. Only when both parts succeed is the clause as a whole a success. The normal short-cut evaluation rules are used.

Another way to describe the unknown values mechanism is that it allows the SOUL interpreter to try to find a reason to fail based on partial input. As described in chapter six, the reason for introducing the mechanism is to statically determine whether an executional joinpoint could possibly match a pointcut, based on information derived trivially from the statement the joinpoint represents the execution of. Using this information the SOUL interpreter does as much of a normal SLD resolution as is possible, trying to see if the query will fail. Some changes to the semantics of the mechanism are possible, so long as it preserves the property that the query will fail for a partial joinpoint description only if it will fail for any possible extension of the joinpoint description.

As a simple example of the application of the mechanism, we present the following three example queries. The `dynamicValue` predicate is the predicate used to declare a value to have an unknown value:

```
Rule test([5], ?val) if
  dynamicValue(?val)

Query test(?x, ?y), equals(?x, [5])
Query test(?x, ?y), equals(?x, [6])
Query test(?x, ?y), equals(?y, [7])
```

In the above example the first query succeeds, the second query fails and the last query is undetermined.

We note that the use of the SOUL DF somewhat blurs the distinction between static and dynamic information. After all, the goal of the framework is to derive non-trivial information from the source of a program. We note however that this information is usually structural and not behavioral.

Bibliography

[1] The AspectJ primer, subject/observer protocol example. http://aspectj.org/doc/primer/examples/observer/index.html. 77, 77

[2] The Java tutorial, a practical guide for programmers. http://java.sun.com/docs/books/tutorial/index.html. 18

[3] Subject-oriented programming. http://www.research.ibm.com/sop/. 14

[4] Subject-oriented programming and design patterns. http://www.research.ibm.com/sop/sopcpats.htm. 13, 13

[5] *International Workshop on Aspect-Oriented Programming at ECOOP*, 1999. 90, 91

[6] *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000. 92, 93

[7] Mehmet Aksit and Bedir Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. 14

[8] Mehmet Aksit, Bedir Tekinerdogan, and Lodewijk Bergmans. Achieving adaptability through separation and composition of concerns. 14

[9] Brian De Alwis. Apostle: Aspect programming in smalltalk. http://www.cs.ubc.ca/ bsd/apostle/. 85

[10] Isaac Asimov. *The Bicentennial Man and Other Stories*. Doubleday, 1976. 53

[11] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998. 5

[12] Lodewijk Bergmans and Mehmet Aksit. Composing multiple concerns using composition filters. 15

[13] Kai Böllert. Aspect-oriented programming, case study: System management application. Master's thesis, Fachhochschule Flensburg, 1998. 10, 68

[14] Kai Böllert. On weaving aspects. In *International Workshop on Aspect-Oriented Programming at ECOOP* [5]. 68

[15] Mohammed Nouraddine Bouraqadi-Saâdani. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects.* PhD thesis, Université de Nantes, 1999. 64, 64

[16] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. 68

[17] Johan Brichau, Wolfgang De Meuter, and Kris De Volder. Jumping aspects. 76

[18] Vassili Bykov. Exceptions by design: Ansi standard exception handling. http://www.smalltalkchronicles.net/Archives/Technical/technical.html, 1999. 49

[19] Lee Carver and William G. Grisworld. Sorting out concerns. 5, 8, 10

[20] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technical University of Ilmenau, 1998. 2, 14, 15

[21] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. 1, 6

[22] Johan Fabry. Full distribution transparency, without losing control - aop to the rescue. Master's thesis, Vrije Universiteit Brussel, 1999. Chapter 4. 5

[23] Robert E. Filman. Injecting ilities. In *Proceedings of the Aspect-Oriented Programming workshop at ICSE'98*, 1998. 9

[24] Robert E. Filman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns at OOPSLA 2000*, 2000. 72

[25] Peter Flach. *Simply Logical.* John Wiley & Sons, 1994. 31

[26] Pascal Fradet and Mario Südholt. An aspect language for robust programming. In *International Workshop on Aspect-Oriented Programming at ECOOP* [5]. 62

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable Object-Oriented software.* Addison-Wesley, 1995. 2, 12, 38, 75

[28] Adele Goldberg and Dave Robson. *Smalltalk-80: the language.* Addison-Wesley, 1983. 36

[29] Bill Griswold, Erik Hilsdale, Jim Hugunin, Mik Kersten, Gregor Kiczales, and Jeffrey Palm. Aspect-oriented programming with AspectJ. Tutorial included with the AspectJ distribution. 57

[30] T.J. Highley, Michael Lack, and Perry Myers. Aspect-oriented programming: a critical analysis of a new programming paradigm. 55

[31] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, MA, 1995. 1, 11

[32] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993. 65

[33] Mik Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 340–352. ACM, 1999. 10, 71

[34] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisworld. Getting started with AspectJ. 3, 19

[35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisworld. An overview of AspectJ. 3, 19, 21

[36] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European conference on Object-Oriented Programming*. Springer-Verlag, jun 1997. 6, 6, 11, 11

[37] P. Koopmans. *Sina/st: User's guide and reference manual*. TRESE project, Department of Computer Science, University of Twente. 14

[38] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, 2000. 72

[39] Cristina Lopes, Erik Hilsdale, Jim Hugunin, Mik Kersten, and Gregor Kiczales. Illustrations of crosscutting. In *International Workshop on Aspects and Dimensional Computing at ECOOP* [6]. 57

[40] Cristina Isabel Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, nov 1997. 17

[41] Cristina Videira Lopes. Recent developments in AspectJ. 19

[42] Cristina Videira Lopes. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox PARC, 1997. 17

[43] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987. 29, 64

[44] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case study for Aspect-Oriented Programming. Technical Report SPL97-009 P9710044, Xerox Palo Alto Research Center, 1997. 16

[45] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson research center, apr 1999. 9, 14

[46] Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972. 6

[47] Mohammed Nouraddine Bouraqadi Saâdani and Thomas Ledoux. How to weave? 63

[48] Andreas Speck, Elke Pulvermüller, and Mira Mezini. Reusability of concerns. In *International Workshop on Aspects and Dimensional Computing at ECOOP* [6]. 9

[49] Patrick Steyeart. *Open Design of Object-Oriented Languages*. PhD thesis, Vrije Universiteit Brussel, 1994. 35, 64

[50] Mario Südholt and Pascal Fradet. Aop: towards a generic framework using program transformation and analysis. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1998. 62

[51] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its applications to meta programming. Technical Report TR-126, Institute for New Generation Computer Technology, 1985. 65

[52] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998. 31, 39

[53] Kris De Volder. Aspect-oriented logic meta programming. In *Proceedings of the Second International Conference on Metalevel Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. 32, 39, 39

[54] Nancy M. Wilkinson. *Using CRC Cards: An Informal Approach to Object-Oriented Development*. SIGS Books and Multimedia, 1995. 9

[55] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001. 34, 35