May 30, 2007

Impact Analysis for AspectJ

A Critical Analysis and Tool-Based Approach to AOP

Dissertation

Maximilian Störzer

Eingereicht an der Fakultät für Informatik
und Mathematik der Universität Passau

Abstract

Aspect-Oriented Programming (AOP) has been promoted as a solution for modularization problems known as the *tyranny of the dominant decomposition* in literature. However, when analyzing AOP languages it can be doubted that uncontrolled AOP is indeed a silver bullet.

The contributions of the work presented in this thesis are twofold. First, we *critically analyze AOP language constructs* and their effects on program semantics to sensitize programmers and researchers to resulting problems. We further demonstrate that AOP—as available in AspectJ and similar languages—can easily result in less understandable, less evolvable, and thus error prone code—quite opposite to its claims.

Second, we examine how *tools relying on both static and dynamic program analysis* can help to detect problematical usage of aspect-oriented constructs. We propose to use *change impact analysis techniques* to both automatically determine the impact of aspects and to deal with AOP system evolution. We further introduce an analysis technique to detect potential semantical issues related to undefined advice precedence.

The thesis concludes with an overview of available open source AspectJ systems and an assessment of aspect-oriented programming considering both fundamentals of software engineering and the contents of this thesis.

Acknowledgements

My thanks goes to Prof. Dr. Gregor Snelting who accepted me as a PhD student and gave me the opportunity to follow my research interests. My thanks also goes to Prof. Dr. Barbara G Ryder (Rutgers University) and Dr. Frank Tip (IBM Research) who gave me the opportunity to participate in their Change Impact Analysis project as a visiting researcher for half a year and who provided funding during this time.

I'd also like to thank all students who worked with me and by doing so contributed to this thesis. In alphabetic order, Uli Eibauer, Katrin Fehlner, Andrea Fischer, Florian Forster, Jürgen Graf, Christian Koppen, Christoph Koch, Marc Pfeffer, Michael Pisula, Michael Rank, Stefan Schöffmann, Thorsten Schühlein and, last but not least, Robin Sterr. And of course I'd like to thank my colleague Daniel Wasserrab for his support while writing this thesis.

Finally my thanks goes to my parents, who financed my studies of computer science and to Carol, who was my anchor while writing this thesis. Without their support this thesis would not exist.

Table of Contents

List of Tables

Listings

List of Figures

1 An Introduction to Aspect-Oriented Programming

As this thesis is about aspect-oriented programming, a fair start indeed is to give the reader a first impression how the author actually understands this term and what is meant by an "aspect-oriented programming language".

This chapter starts with a short discussion of Separation of Concerns as a core principle of modern software engineering, before discussing the core ideas of aspect-orientation. In the following chapter we examine available case studies to summarize published experience with aspect-oriented software projects.

We critically examine this experience, and especially the impact of aspect-oriented techniques on software quality criteria, the so called *-ilities of software*, like comprehensibility, evolvability, etc. The chapter is concluded with an survey of related work also addressing some of the critique stated above.

1.1 Separation of Concerns

How do we write (good) software? Idealized the process of writing software has three major steps:

1. *decomposing* the problem into individual *concerns*,
2. *mapping* these concerns to appropriate modules and finally
3. *composing* these modules to a software system solving the initial problem.

In literature decomposition and composition are widely used terms whenever authors talk about decomposing *software* into manageable pieces. However, software itself is not decomposed (it does not exist yet). Programmers decompose problems. Thus this thesis explicitly uses the term *mapping* to describe one of the most interesting problems of language design and software engineering: how do we best implement the functionality a customer needs from *his* view, i.e. his *concerns* using the abstractions the chosen programming language provides.

Development of programming languages from this point of view can be seen as an ongoing effort to improve the possibilities for programmers to define such mappings by providing

appropriate language constructs and abstractions. A goal of software engineering—among others—is to develop techniques how to use (new) programming languages and tools to produce high quality software.

One of the most important terms in this context is *modularity*. When writing software, programmers follow a *divide and conquer* approach—decompose the problem into small manageable pieces (often called requirements or concerns), solve these (optimally by mapping a single concern to a single module), and then recompose the solutions (i.e. modules) to solve the whole problem. A baseline for this however is a general question: How should software modules in general look like? In his famous article "On the Criteria To Be Used in Decomposing Systems into Modules" [17] D.L. Parnas outlines that the expected benefits of modular programming are threefold:

- First, *development time will be shortened* as separate teams could implement different modules in parallel, mostly independent of each other,
- second the resulting software products are *more flexible* as it will be possible to make drastic changes to a module implementation without a need to change other modules and
- third, the resulting systems will be *more comprehensible*, as it is possible to understand a system one module at a time.

Today we derive four major software quality criteria from these expected benefits, the so-called *-ilities of software*:

Reusability: The ultimate dream of software engineers is to compose software systems out of standardized off-the-shelf components, comparable to other engineering disciplines. Thus software quality can be measured by the amount of modules which offer services reusable in a different context.

Scalability: It is a well-known fact that increasing the number of programmers working on a system does not proportionally decrease the time needed for completion, as programming involves considerable communication and coordination efforts. However, this management overhead depends on the modularity of a system. Well-modularized systems allow (relative) independent implementation of different modules and thus reduce communication overhead.

Maintain- or Evolvability: Empirical studies have shown that up to 80% and more of the total cost in a software system stem from the maintenance phase [14, 6].[1] If a concern changes, the amount of necessary changes depends on the locality of its implementation—well localized code can be more easily updated, compared to not localized code potentially resulting in invasive updates throughout the whole system.

Comprehensibility: Software is used to automate complex tasks—and as such is very complex itself. Comprehensibility addresses how easy it is to understand the implementation and the inner workings of software. Intuitively systems with well-defined modules implementing a single concern are easier to understand than systems with modularization deficiencies.

All these quality criteria finally depend on the ability of software engineers to keep the implementation of different concerns apart from each other. Well localized and encapsulated concern implementations are considerably easier to understand, reuse or change if necessary.

[1] While these figures are not surprising for the days of the software crisis, they are still valid today. Many companies still own mission critical legacy systems, where an increasing amount of money is necessary to keep these systems running.

While modularization and its benefits already were accepted concepts in the 70ies the *criteria* to by used to design good modules are still discussed today. In his article Parnas identifies the principles of *encapsulation* of core assets (or concerns) *likely to change* and the principle of *information hiding* as good criteria to define modules. Thus *modules* are characterized by their knowledge of a (single!) design decision which is hidden from all other modules. Module *interface* are chosen to reveal as little as possible about their internals.

Several years earlier Dijkstra discussed the benefits of layered architectures [5]. Although in this article Dijkstra describes the architecture of an operation system the ideas outlined there directly map to modules as well. Applied in this context modules should be built using the services of other lower-level modules which again hide their internal implementation details. The dependency structure of modules should not be circular—no module may use services of higher-level modules. Mutual dependent modules are highly coupled, and in general considered bad module design.

These principles examined by Parnas and Dijkstra nowadays are often referenced using the term *separation of concerns*. This term thus is usually contributed to Parnas and Dijkstra.

1.1.1 Impact on Programming

When observing the development of programming languages, one can observe that better support for separation of concerns is a driving force behind evolution of (high level) programming languages. As a first step hierarchical software design was used (stepwise refinement approach [21]), but in 1967 a new programming paradigm emerged with Simula: object-orientation. In the 80ies the principle started to spread with growing popularity of Smalltalk. In his paper Parnas gives examples of plausible concrete rules how to define modules. One of them is to move *data structures together with accessing and modifying procedures* into one module (today we call such modules *abstract data objects*). This principle—to encapsulate data together with the code working on it—is also a basic concept for object-oriented programming as we know it today (although classes in general implement *abstract data types*, i.e. additionally allow to instantiate several distinct exemplars of an abstract data object).

Although these concepts considerably improved the ways how software can be built, object-orientation is no silver bullet to create good modules. Today there is a growing consensus that modern high level programming languages—including but not limited to object-oriented languages—still lack support to cleanly modularize *any* kind of concern. In [3] Baclawski and Indurkhya state:

> "Concepts of the real world, which programs and databases attempt to model, do not come in neatly packaged hierarchies."

In [20] Tarr et al. also analyze this problem and observe that—although most programming languages support *some* mechanisms for decomposition and composition—in general they typically only support a single, *"dominant" dimension of separation* at a time. For example procedural languages use procedures, i.e. allow to decompose a system using *functional* criteria; object-oriented languages define modules based on the data; etc. Tarr et al. call this observation the *tyranny of the dominant decomposition*. In their opinion a simultaneous separation of overlapping concerns in multiple dimensions is necessary to improve the ability of programmers to separate concerns.

Aspect-Oriented Programming is one of the techniques trying to solve the problems steaming from this lack of modularization support by extending the possible modularization constructs in state of the art languages. In the following this thesis outlines these problems and describes how aspect-orientation tries to solve them.

1.1.2 Current Programming Paradigms and Crosscutting Concerns

The term aspect-oriented programming stems from [12]. Although this was not the first paper to address the tyranny of the dominant decomposition (for example refer to the Composition Filter approach [1, 2], the Hyper/J approach [20], Adaptive Programming [13] or Meta Object Protocols [10]), it has surely been one of the most influential papers.[2]

Among others Gregor Kiczales et al. analyzed modularization problems resulting in *scattered* and *tangled* code. Tangling addresses the observation that the implementation of multiple concerns[3] can overlap and end up with hardly understandable bloated code, when several of these conflicting concerns are implemented in a single module due to lack of proper modularization support[4]—i.e. the different concerns are not clearly separated from each other.

As an example the paper reports about experience gained from a simplified image processing system (among others). The *initial (functional) implementation* based on a set of hierarchically composed filters was easy to understand, but suffered from bad performance and a high memory footprint as a lot of intermediate images were created solely for the purpose to serve as input for the successive filter. A *manual optimization* of the system by collapsing several of these filters (by fusing loops) considerably reduced memory footprint and improved performance—however, at the cost of comprehensibility—the implementation of the performance concern now results in tangled hardly comprehensible code. Additionally code size exploded, as each optimized filter now was a unique method rather than reusing previously defined primitive filters.

The paper states that performance and comprehensibility as (meta) concerns cut across each other, meaning it is very hard to implement functional and performance concerns in separate (traditional) modules. The functional abstractions available in the used implementation language optimally match the functional decomposition of the system, thus the initial implementation is easy to understand. However, the performance optimization fused structurally similar loops in the implementation of successive filters (based on the data flow). For this kind of optimizations procedural languages have no adequate support. Instead these optimizations fuse loops across traditional module boundaries (the successive filters). Tangled code is the result, a consequence of the tyranny of the functional decomposition.

Besides performance optimization other concerns like synchronization, minimization of memory footprint or network traffic, or failure handling are named as typical examples of concerns where tangling occurs. As a commonality these concerns tend to interfere with the implementations of the main functional concerns implemented by a system—the concerns overlap. These kinds of concerns are often termed *non-functional* concerns; to express their overlapping or crosscutting nature for a given system the term *crosscutting concern* is used; this thesis will also use these terms in the following.

When examining the tangled implementations of crosscutting concerns a second problem becomes apparent—the implementation of these concerns is not modularized at all but *scattered* across several modules. So beside "polluting" functional modules with code not belonging here (tangling), there is also no module to localize these concerns. Evolution of the corresponding code in general requires global system modifications, so considerably reducing evolvability of this code. More abstractly crosscutting concerns considerably reduce software quality measured using the -ilities describes above.

The success of aspect-oriented techniques in part surely can be traced back to this problem analysis. Scattering and tangling are problems that most software engineers have to deal with in their everyday work. So aspect-orientation indeed *addresses a valid problem*. In the

[2] While writing this thesis CiteSeer (`http://citeseer.ist.psu.edu/`) listed more than 800 citations of this paper.

[3] The original article uses the term design goal, and in literature the terms requirements or concerns are used synonymously. We will use the term concern throughout this thesis.

[4] The original article uses the term component, but as aspect-orientation focuses of improving modularity of software, we use the term module throughout this thesis.

following we examine how aspect-orientation tries to solve it.

1.2 The Approach of Aspect-Oriented Programming

To break the tyranny of the dominant decomposition and so remove scattered and tangled code, aspect-oriented programming introduces *aspects* in contrast to traditional modules to provide an additional decomposition dimension. A concern can be implemented in a (traditional) *module*, if traditional programming constructs support a clear (i.e. well-localized and composable by the language) encapsulation of its implementation. Otherwise this concern is mapped to an *aspect*. Such concerns not modularizable using traditional modules are called *crosscutting concerns*.

A system defined by modules and aspects is constructed by passing both module and aspect definitions to an *aspect weaver*. The aspect weaver first analyzes the modules and aspects. For the performance example in [12] the base for the analysis is a data flow graph on which the aspect defines operations. In a subsequent phase this graph is optimized by eliminating data flow nodes and adjusting their code bodies according to the rules defined in the aspects. Finally code is generated from this optimized data flow graph.

More abstractly, the weaver first generates a *joinpoint model* from the module definitions and then applies the aspects to this model. *Joinpoints* are not necessarily explicit constructs but can be implicitly defined. They define synchronization points for aspects in a module.

To summarize, in [12] the problem of code scattering and tangling is analyzed. The tyranny of the dominant decomposition is identified as the reason for these problems (although this term stems from [20]). To solve this problem, aspects as a meta- or higher level programming construct are proposed. The aspect languages are targeted to explicitly express crosscutting concerns. The aspect weaver is then used to compose aspects and modules to produce a woven system, by analyzing the modules to derive the joinpoint model, apply the aspects to the joinpoints and finally generate the woven system.

It is interesting to note that all crosscutting concerns addressed in the initial paper *do not affect the input-output behavior of the application*. However, since the appearance of this initial paper various different systems claiming to be aspect-oriented and also very different applications for aspect-oriented techniques have been proposed where this in general does not hold. Second, aspects as found in current aspect languages are no longer restricted to a static joinpoint model. Indeed, application of aspects can again result in new joinpoints to become available. A detailed discussion of joinpoint and joinpoint model is deferred until Section 1.2.3.

1.2.1 Aspect-Orientation is Quantification and Obliviousness

Aspect-orientation had considerable impact on software engineering research, and lots of different systems and techniques emerged all claiming to be aspect-oriented. This of course led to the question of what aspect-orientation actually is: Macros? Reflective techniques? Meta-Programming? Although aspect-orientation is related to all these techniques, it is more powerful than macros, less powerful—but easier to understand and safer!—than meta-programming and sometimes (partly) implemented using reflective techniques. So what is it?

Filman and Friedman [7] further examined the idea of aspect-oriented programming and factored out two main concepts they considered essential for aspect-oriented systems: *quantification and obliviousness*. Quantification in this context describes the possibility for programmers to define behavior based on quantified statements over a program, e.g. “each time a method `void foo()` is called, update a cache”. Obliviousness states that the underlying program does not have to be prepared to allow definition of these additional behaviors.

To motivate their observations the authors examined (part of) the development of programming languages. The earliest languages were completely *unitary and local*, meaning that a statement had effect in precisely one place in the program (unitary) and that it is in general located proximate to its predecessor or successor statements (local). The paper states that the development of programming languages constantly moves away from these two properties. Introduction of subprograms (i.e. procedures) for example introduced constructs to factor out common code in libraries, which can be called elsewhere, thus breaking with locality of statements. Although a big step ahead, programmers are still cognizant of called behavior, and the call is still unitary.

Inheritance is identified as the next step, as it allows a limited form of quantification. If a cooperative base class programmer is assumed, behavior can be added to the base class and inherited by all subclasses, if these classes call the super implementation. Filman and Friedman finally state that true aspect-oriented systems allow systems to work with modules written by oblivious programmers, thus also removing the necessary cooperation needed by object-oriented inheritance. They state that

> "AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers."

Or stated otherwise, programmers should be able to make statements of the form "In program P whenever condition C holds, perform action A".

The ideas presented by Kiczales et al. fit this characterization of aspect-orientation. Base modules can be developed obliviously of the aspects used later on to adapt the system. The aspect itself is defined by describing conditions on the control or data flow of the original program. More abstractly, advice takes an action based on predicates defined on the joinpoint model, thus allowing quantified statements over the module program.

While quantification is still seen as a cornerstone of aspect-orientation, obliviousness has been criticized (for example in [18]). Case studies have shown that adding non-trivial behavior to a system using aspects in general requires a system which is aware of additional behavior defined by aspects. Persistence of system data is an example. While loading and storing of data can be aspectized, the system has to be aware where its data comes from, i.e. that some data *is* loaded from some background storage, as the source of data has major impact on system control flow.

The original paper of Kiczales and the work of Filman and Friedman describe the initial idea and two underlying characteristics of aspect-orientation, but they are still rather abstract. Meta-programming and reflective techniques also fit this definition. To get a better understanding of these ideas, this thesis will concentrate on aspect-oriented programming as implemented by the most popular general purpose aspect-oriented language AspectJ.[5]

1.2.2 An Overview of AspectJ

AspectJ as introduced in [11] is an aspect-oriented extension of Java. Compared to the first paper on aspect-oriented programming [12], AspectJ clarifies several of the terms by assigning a clear (but sometimes different) semantics.

AspectJ supports two different kinds of crosscutting, called *static* and *dynamic crosscutting*. Static crosscutting—which includes *inter type declarations* as most important construct—allows programmers to add methods and fields to existing classes via the aspect, so changing the (static) interface of existing classes. This is simply denoted by declaring the

[5] While the current language version is 1.5, the considerations in this thesis refer to the (direct!) predecessor language version 1.2., which is a subset of the 1.5 version. Thus the considerations in this thesis are still valid for AspectJ 1.5.

respective members in the aspect using qualified names, so specifying the target class as outlined in Listing 1.1. Here the aspect `InterTypeDemo` adds a field called `x` and a method `void setX(int)` to class `SomeClass`. As another important feature static crosscutting allows to

Listing 1.1: Inter Type Declarations in AspectJ

```
1 aspect InterTypeDemo {
2   int SomeClass.x = 5;
3   void SomeClass.setX(int x) {
4     this.x = x;
5   }
6 }
```

modify the inheritance hierarchy by reassigning super classes (as long as type constraints are maintained).

Dynamic crosscutting is based on the terms joinpoint and advice. While the terms joinpoint and joinpoint model were rather fuzzily defined in the original article on aspect-orientation, AspectJ defines a *joinpoints* as "certain well-defined points in the execution of a program". Joinpoints for AspectJ—simplified—can be considered as nodes in a simple runtime object call graph. Examples of such joinpoints are method call, method execution, field access, initialization events or error handlers.

AspectJ also introduces the term *pointcut* to specify a set of joinpoints. Pointcuts are specified using pointcut designators, a declarative expression selecting joinpoints available in the underlying joinpoint model. Note that pointcuts not only serve to select a set of joinpoints, but also can be used to *expose context* of a selected joinpoint. Behavior defined in aspects can access and modify this exposed context. The version of AspectJ this thesis is based on (1.2) supports a considerably different set of primitive pointcut designators as outlined in the original paper. This already shows that this part of the language (often called *pointcut language*), although stabilizing, is still controversially discussed. An example of a pointcut definition is shown in Listing 1.2. The pointcut named `callsToFoo` selects all joinpoints corresponding to invocations of a method called `foo` with a single `int` parameter using the `call` pointcut designator. The parameter is made accessible using the `args` keyword.

Listing 1.2: Pointcut Definition in AspectJ

```
1 aspect PointcutAndAdvice {
2   pointcut callsToFoo(int i):
3     call(void foo(int) && args(i);
4   ...
5 }
```

The third important cornerstone of AspectJ is *advice*, a method-like construct bound to a pointcut definition. Each time a joinpoint selected by the associated pointcut is reached, advice is executed either *before*, *after* or *around* (instead) of the joinpoint. The advice body is written using traditional Java code and can access the context of the joinpoint made available by the pointcut. Listing 1.3 shows a piece of `before`-advice bound to the pointcut defined above. Each time a method matching the signature above is called, the piece of advice will be executed *before* the method is invoked, i.e. the value of the integer parameter will be printed to standard out.

Aspects finally are a new class-like construct comprising all aspect-specific elements. Aspects can also contain definitions of traditional class members. Per default, aspects are

Listing 1.3: Advice Definition in AspectJ

```
1 aspect PointcutAndAdvice {
2   pointcut callsToFoo(int i):
3     call(void foo(int)  args(i);
4   before(int i): classToFoo(i) {
5     System.out.println("Method foo "
6       + "called with parameter : " + i);
7   }
8 }
```

singletons. However it is also possible to create multiple instances of an aspect by using appropriate modifiers. For now this short introduction to AspectJ should suffice, for details the reader is referred to the AspectJ manual.[6]

When comparing the ideas presented in [12] with the current release of AspectJ, several important differences can be observed. [15] gives an interesting overview of the roots of AspectJ and how the language came into being. The paper also outlines the main differences and commonalities compared to the early aspect-oriented systems which were used as examples in [12].

First of all, aspect-orientation as outlined in [12] had a rather domain specific flavor. AspectJ in contrast is a general purpose aspect language and as such applicable in a broader context. However, there was also a price for this—AspectJ is more low-level compared to the early domain specific aspect-oriented systems.

Second, for the example systems presented in [12] aspects and modules were strictly separated. More precisely the modules were not able to reference aspects. For AspectJ this is different, as the singleton aspect instance is accessible from the module code. Note that if aspects are indeed accessed in the module code this introduces a high coupling of aspects and base modules.

An important clarification was the definition and elaboration of the joinpoint model. The term joinpoint was only fuzzily used in the original AOP paper. Earlier systems had a similar but more restricted join point model, but AspectJ generalized this term supporting a broader applicability of the language.

1.2.3 Aspect-Orientation—An abstract view

While AspectJ is the most popular aspect-oriented language today there are several comparable approaches which all basically use the same abstract programming model. This section elaborates on the basic concepts underlying all these approaches. Although this thesis concentrates on AspectJ, the problem analysis and concepts introduced here are valid for many other languages using the concepts introduced below. This abstract model of aspect-orientation has in part been previously published in [19].

Joinpoint Model

Most (or even all) aspect-oriented languages and systems add new constructs to an underlying *base language*. These additional constructs allow to select "well-defined points during the execution of a program" [11] called *joinpoints*. We will use the term *joinpoint model* to refer to the set of all available joinpoints in a given system.

[6] http://www.eclipse.org/aspectj/

The granularity and kind of the available joinpoints greatly differ from system to system, also depending on the underlying base language. As this thesis is not about classification of AO Systems or joinpoint models we will only describe an important difference observable when studying the different joinpoint models described in the literature: *static* and *dynamic* joinpoints.

Static joinpoints are very natural for programmers, as these joinpoints are directly visible in the program code. Examples are method calls, field assignments or even a single operator. Static joinpoints can be described without knowledge of runtime values.

Dynamic joinpoints in contrast depend on runtime values. Examples are method calls where target or calling objects *have a specific runtime type* or field assignments where a *condition on the new runtime values are met*. These joinpoints can only be described by referring to runtime values, a pure "static" description—based on code entities only—is not possible.

An important observation to make however is that often (for AspectJ always) a dynamic joinpoint can be mapped to an underlying static joinpoint. This underlying static joinpoint has been called *Joinpoint Shadow* in [9, 16].

A simple intuition to compare dynamic and static joinpoints might be the following. Consider the trace of a program execution logging each method entry and exit. The trace shows all methods executed during a program run, as shown in Figure 1.1. All calls with the *same*

```
1  class TraceDemo {
2    public static void main(String[] args) {
3      TraceDemo demo = new TraceDemo();
4      for (int i=0; i<10; i++) {
5        demo.run(); // joinpoint call(* run())
6      }
7    }
8    void run() {
9      // do something
10   }
11 }
```

```
→ main
  → run
  ← run
  → run
  ← run
  ...
  → run
  ← run
← main
```

Figure 1.1: Source Code and resulting Program Trace.

underlying lexical code position form a single static joinpoint. But each single call to `run`—respectively each line in the trace—is a distinct dynamic joinpoint. In the example, method `run` is called ten times, but alway from the same call site, i.e. the call site is the joinpoint shadow.

Although there may be systems where a similar mapping from dynamic joinpoints to joinpoint shadows is not possible, such a mapping exists and is valid for AspectJ and AspectJ-like languages. As the problems and methods described in this thesis are relevant for this class of languages, we will accept this interpretation of dynamic joinpoints in the following.

Joinpoint Selection

The joinpoint model defines all those points in a program (or its execution) where an aspect programmer may influence the system. As in general one is not interested in changing system behavior at all available joinpoints, aspect-oriented systems in general offer ways to select a subset of the joinpoints defined by the joinpoint model. We refer to this mechanism using the term *joinpoint selection*.

Today there are very different approaches to select joinpoints, from simple enumeration of lexical positions in program code up to Turing complete selection languages which allow to formulate a meta-program. For the latter running these meta-programs on the source code

produces a set of joinpoints as result. An example for this strategy is [8]. Other approaches suggested to augment the code with semantic tags which can be used later to bind new functionality to [4]. The new version of AspectJ (1.5) also adopts this strategy.

The strategy of AspectJ and related languages is to offer a set of predefined operators called *pointcut designators*, which can also be combined using logic operators ($\wedge, \vee, \neg$). AspectJ's `call`, `get/set` or `this` pointcut designators are examples. Each pointcut designator allows to pick out a special kind of joinpoint, its parameters further restrict the selection.

As these pointcut designators often explicitly reference *names* in the code base, they have been criticized as introducing a high coupling between aspect and base system. As a reaction to this critique, languages introduced wild card mechanisms allowing to exploit naming conventions. Additionally they introduces *abstract aspects*, where the joinpoint selection could be postponed to the implementation of a concrete (coupled) sub-aspect, thus allowing to reuse functionality defined in the abstract aspect.

Joinpoint selection languages are still controversially discussed today. The idea is to offer *semantic joinpoint selection*, and to avoid referencing named or syntactic elements. However, truly semantical selection languages are not yet available and might be hard to achieve in general (see Section 5.1.2 for details).

Naturally the granularity and nature of the joinpoint model and the underlying base language considerably influence the constructs available in selection languages. For the following we will use the term *pointcut* not only in the context of AspectJ but more widely to refer to any expression in a joinpoint selection language. Evaluating a pointcut results in a set of joinpoints. For a dynamic joinpoint model however this evaluation is only possible at runtime, as here per definition runtime values are necessary.

Joinpoint Adaptation

Finally if a pointcut selects a set of joinpoints, an aspect-oriented language has to provide a way to specify the actions to be taken at these joinpoints. Therefore, a *joinpoint adaptation mechanism* has to be provided which allows to formulate these actions and when—before, after or instead of the selected joinpoints—these actions have to be taken. A joinpoint with an attached action will be called an *adapted joinpoint* in the remaining of this thesis.

As one often wants to access the data of the base system or even call some of its functionality, the actions to be taken can be written using the base language itself, although in this case some additional constructs to specify how a joinpoint should be augmented have to be provided. To be able to model advice execution and behavior of the base system in a single model AspectJ follows this approach.

Kinds of Crosscutting Concerns

When we examine crosscutting concerns, we observe that there are two different kinds of them. Some concerns can be easily implemented with a single pointcut and a single piece of advice, and nevertheless affects a large set of joinpoints. Other aspects have to define very specific pointcuts and separate pieces of advice for only few (sometimes only a single) joinpoints. We call the first set of concerns *homogeneous (crosscutting) concerns* and the latter one *heterogeneous (crosscutting) concerns*.

Homogeneous concerns usually adapt joinpoints with behavior which does not depend on the specific joinpoint context. Consequently a single pointcut and a single piece of advice suffices to implement them. The resulting implementation thus tends to be very elegant and are usually superior compared to non-aspectized implementation.

For heterogeneous aspects this is different, as here applied behavior depends on the joinpoint context. While it might be possible to implement an adaptive piece of advice in some cases, for others only implementing different joinpoint-specific pieces of advice is a practical

solution. This easily results in a bloated aspect implementation, which is often inferior compared to the object-oriented implementation. Note that the latter can also be a deficiency of existing aspect-oriented languages.

1.3 Summary

In this chapter we described separation of concerns as a driving factor in the development of programming languages. We discussed that currently available programming paradigms all suffer from modularization deficiencies called the tyranny of the dominant decomposition in literature.

We then described how aspect-oriented programming attacks these deficiencies, discussed AspectJ as the most popular aspect-oriented language available today and also introduced a more abstract view on aspect-orientation at the end of this chapter.

The following chapter will review published case studies with AspectJ, summarize reported problems and discuss published approaches to attack these problems.

Bibliography

[1] AKSIT, M., AND TEKINERDOGAN, B. Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects using Composition Filters, 1998.

[2] AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. Abstracting Object Interactions Using Composition Filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming* (1994), R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds., vol. 791, Springer-Verlag, pp. 152–184.

[3] BACLAWSKI, K., AND INDURKHYA, B. The Notion of Inheritance in Object-Oriented Programming. *Commun. ACM 37*, 9 (1994), 118–119.

[4] BONÉR, J. What are the key issues for commercial AOP use: how does ApectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development* (New York, NY, USA, 2004), ACM Press, pp. 5–6.

[5] DIJKSTRA, E. W. The structure of the 'THE'-multiprogramming system. *Commun. ACM 11*, 5 (1968), 341–346.

[6] ERLIKH, L. Leveraging legacy system dollars for e-business. *IT Professional 2*, 3 (2000), 17–23.

[7] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. Tech. rep., RIACS, 2000.

[8] GYBELS, K., AND BRICHAU, J. Arranging Language Features for more robust Pattern-based Crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 60–69.

[9] HILSDALE, E., AND HUGUNIN, J. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development* (New York, NY, USA, 2004), ACM Press, pp. 26–35.

[10] KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. G. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[11] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An Overview of AspectJ. *Lecture Notes in Computer Science (LNCS) 2072* (2001), 327–355.

[12] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[13] LIEBERHERR, K. J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[14] LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. Characteristics of Application Software Maintenance. *Commun. ACM 21*, 6 (1978), 466–471.

[15] LOPES, C. V. *Aspect-Oriented Software Development, AOP: A Historical Perspective (What's in a Name?)*. Addison Wesley, 2004, ch. 5, pp. 97–122.

[16] MASUHARA, H., KICZALES, G., AND DUTCHYN, C. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Compiler Construction: 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003* (January 2003), vol. 2622 of *Lecture Notes in Computer Science (LNCS)*, pp. 46–60.

[17] PARNAS, D. L. On the Criteria to be used in Decomposing Systems into Modules. *Commun. ACM 15*, 12 (1972), 1053–1058.

[18] RASHID, A., AND CHITCHYAN, R. Persistence as an Aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 120–129.

[19] STOERZER, M., AND HANENBERG, S. A Classification of Pointcut Language Constructs. Software-Engeneering Properties of Languages and Aspect Technologies (SPLAT). Workshop at AOSD (2005), 2005.

[20] TARR, P., OSSHER, H., HARRISON, W., AND STANLEY M. SUTTON, J. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 107–119.

[21] WIRTH, N. Program Development by Stepwise Refinement. *Commun. ACM 14*, 4 (1971), 221–227.

2

AOP as "Silver Bullet"? A Review of Case Studies

Research papers dealing with aspect-oriented techniques today—including some of the author's papers—often start with a characterization of AOP as a "new promising programming technique (or even paradigm!) solving modularization problems experienced when using conventional programming languages" (or at least something close to it).[1] Let us discuss this characterization in more detail.

(i) First of all, as AOP was introduced in 1997, it is now 9 years old and no longer *new*.

(ii) While maybe still promising it might be time to critically examine available case studies (which we did, see Section 2.1). Unfortunately there are only *few of them*, and as far as we know *no long term case studies*.

(iii) When analyzing current AO languages and the results reported in the above mentioned case studies, some *problems of AOP*—compared to "conventional" programming languages—become apparent. We will discuss these problems in Section 2.2 and finally review research papers addressing these problems in Section 2.3.

2.1 Examining available Case Studies

Aspect-oriented techniques became very popular in the software engineering research community. However whether these ideas prove useful in practice is still under discussion. Only few case studies of evaluating aspect-orientation in research projects and basically no reports from industry strength projects are available. Studying available case studies nevertheless gives interesting insights as to where and how aspect-orientation is superior compared to classical object-oriented designs. But these case studies also highlight some of the problems current aspect-oriented languages (or more precisely AspectJ) suffer from. In the following some publicly available case studies are summarized and discussed to factor out open problems of aspect-oriented approaches comparable to AspectJ.

[1] Due to space limitations we refrain from giving an (exhaustive) list of references.

2.1.1 Aspectizing Distribution

In [8] Kersten and Murphy report about the implementation of the web based learning environment *Atlas* built on Java Server Pages and AspectJ, which resulted in a system with around 10 kLoC of AspectJ code. Although based on a rather old version of AspectJ (0.4, which is in part outdated), some principal results are still valid and interesting to study.

To provide scalability on the software side, different *network setups* (from single server to distinct web server, several parallel application servers and a database server) had to be supported. Instead of implementing a complex class hierarchy, the Atlas base system was implemented obliviously of the different distribution setups. These setups were then implemented as aspects. The paper reports that this modularization resulted in both cleanly modularized distribution and network setups as well as an easily readable base code. During development, additional *development aspects* have been used to trace calls in the system and log profiling data.

Beside this success story the paper also states that to improve comprehensibility and evolvability, it is important to *carefully design aspects*. An important tool in this context proposed by the paper is the "knows-about" relation between aspects and classes. An aspect/class knows of a class/aspect, if it explicitly references it or depends on its services. Based on this relation, four cases are distinguished:

Closed association: neither aspect nor class know of each other. This is the most loosely coupled form of association, and for example occurs for a tracing aspect. The aspect is in general easily reusable, as it is truly orthogonal to the system it is applied to.

(Aspect-Directional): the class knows of the aspect, but not vice-versa. The aspect might be reusable. This form of association is possible but not very common for AspectJ.

Class-Directional: the aspect knows of the class, but not vice-versa. The class can be reused.

Open association: Both aspect and class know each other. In this case, aspect and class are tightly coupled, with all resulting negative effects on understandability and reusability of the code.

During the development of Atlas, *open associations* often occurred in the beginning but were soon identified as problematic and replaced by class-directional associations. These observations are in the spirit of layered systems as outlined by Dijkstra [6]. *Closed associations* would be the preferable ideal case, but are often not achievable.

Although the paper states that in the case of *class-directional association* both class and aspect are easier to understand, this implicitly assumes that the aspect does not unexpectedly change the semantics of the class. For the distribution aspects for example the semantics of an invoked service is the same no matter if the service is local or executed on a different machine. Thus the produced result matches the expectations of the programmer reading the original code without distribution support.

The paper also discusses problems related to undisciplined use of both *inter type declarations and advice*. As the system grew, this resulted in code which was hard to understand as it was no longer trivial to figure out how all the different building blocks of the system fit together. Additionally, changes in the base class often resulted in several subsequent changes in the aspect code. As a result the authors restricted the form of aspect-class interfaces used in the system.

The described problems and development policies used to counter them show that undisciplined use of aspects can result in reduced comprehensibility and maintainability of the resulting system.

2.1.2 Aspectizing Transactions

In [10] Kienzle and Guerraoui report about their experience of using AspectJ to aspectize transactional behavior in the OPTIMA framework. Their report is a rather critical case study. The authors concentrated on an evaluation of the obliviousness property and in detail examined three different layers of obliviousness in the possible application of aspects.

First, they examined if an *application oblivious* of transactional behavior could be made transactional using an aspect. Their conclusion is that this is clearly not the case. For justification they outlined the synchronization mismatch in Java and for transactions. In Java, mutual exclusion of critical operations is used to assert correctness of parallel threads while transaction isolation (as one of the ACID principles [13]) requires that all operations of a transaction are serialized. This mismatch is only solvable if all cooperating threads are executed within the same transaction. Automatic determination of the relevant thread however is not feasible. Additionally, an application might contain irreversible actions (I/O operations for example) where a rollback is not possible per definition, thus preventing ex-post addition of transactional behavior.

Second, they studied if *transactional interfaces* could be localized and implemented using an aspect. While technically possible, their solution was problematic for failure reporting, as now exceptions implicitly are coupled with a rollback on the one hand and—due to Java's checked exceptions semantics—newly introduced transaction exceptions had to extend `RuntimeException`. As a consequence, handling these exceptions can be missed by programmers.

Finally they also examined if the *setup of the transaction framework* can be aspectized. The paper introduces an elegant solution to define recovery manager, concurrency control and semantically compatible operations using aspects, but also states that this implementation is not without cost. For programmers it is important that this aspectized configuration in general depends on the semantics of the underlying application. Note that the syntactical decoupling of the configuration does not imply a semantical decoupling. Thus, if the underlying application evolves, care has to be taken to avoid invalidation of the aspectized configuration. For this evolution problem they suggest that development tools could considerably support programmers by displaying relevant tightly coupled aspects.

Although the authors admit that their case study is too limited to draw any conclusion in general, two interesting things can be noted:

1. The authors showed that an assumed crosscutting concern like transactional behavior indeed can be an essential part of core system semantics. Thus "separation" of this concern might be syntactically possible, but the resulting aspect will always be *semantically tightly coupled* with the underlying base modules.

2. They argued that *(base) system evolution* is a relevant problem, as in that case aspects might be (silently) invalidated. They suggest that tools are necessary to alert programmers of tightly coupled aspects.

2.1.3 Error Handling

In [11] Lippert and Lopez examined exception usage and handling in an interactive business-application framework called JWAM. They observed that exception handling code in general is tangled with the core functionality of a module and that this code additionally is highly redundant: only five different exceptions were caught in over two thirds of all exception handlers and only 14 different standard reactions (like "log and ignore") were implemented.

Aspects can avoid redundant code in this context and thus allow easier evolution or replacement of the error handling strategies. It is also possible to define some default exception handlers and apply them to different systems. As a result of the aspect-oriented refactoring

in this case study the code size could be considerably reduced, as many default error handlers could be localized and redundant code was removed.

As qualitative results the paper reports that the code quality improved considerably. As error handlers were localized in aspects, changing an error handling policy now is very easy, compared to the invasive code modifications without aspects. The aspectized error handlers also allow to start with some default error handling strategy and later incrementally refine error handling. Default error handling aspects can be reused.

As this case study was performed with AspectJ 0.4, some limitations of the language were also addressed in this paper. Some of these limitations are no longer valid for current versions of AspectJ. It has to be stated however, that the refactoring concentrated on some *very common default error handlers and policies*, like contract enforcement and "log and ignore" error handling.

As this case study is rather old, the reported results are at least in part outdated. AspectJ developed to address some major issues documented by this case study. However, it would be interesting to examine aspectizability of more complex and/or specialized error handling strategies, which require more context information of either the error or the error handling location. In this case, error handling develops to a heterogeneous concern, as specific error handling does not allow to formulate quantified statements when to apply it—it is more likely applied at a *single* position.

2.1.4 Distribution and Persistence

In [18] Soares et. al. examined refactoring of a Java Health Care system using AspectJ. This case study explicitly refactored an existing Java application. The goal of the study was to improve the layered system architecture by removing tangled and scattered code due to crosscutting concerns and thus achieve better adaptability of the system for different distribution scenarios and persistence mechanisms.

The study concentrated on three main concerns: distribution, persistence (including transaction control) and error handling. To implement distribution, the authors heavily use AspectJ's static crosscutting features in addition to advice. Reusability of aspects is achieved by using abstract aspects.

While their solution decouples distribution, persistence and transaction control using aspects, the authors complain that the version of AspectJ they are using prohibited some generality and so required them to write a lot of structurally identical code. This loss of generality is identified as a software maintenance issue, as interface extensions of objects to distribute require code adoptions on the aspect side. Code generation or simple extensions of AspectJ are suggested to leverage maintenance related problems.

A second observation of the authors is that using aspects does not free developers from careful design. In the case study, persistence and distribution were not completely independent from each other. Both aspects in particular require synchronization of objects. Aspect interference (both semantical and syntactical) thus is identified as a major issue in this case study.

2.1.5 Persistence as an Aspect

In [16] Rashid and Chitchyan implemented an aspect-oriented version of a bibliography system as a real world assessment of *persistence as an aspect*. In their study they examined in detail whether

- it is possible to aspectize persistence in general,
- this aspectized implementation is reusable and

- the base application can remain oblivious
  of the persistence aspects.

Their resulting aspect-oriented persistence framework shows that although *aspectizing persistence is possible in a highly reusable manner*, it is not possible that the application remains completely obliviously of these persistence aspects.

Especially retrieval of objects has to be explicitly accounted for by the application, as external data sources have to be considered. Deletion is a similar case, partly due to the Java garbage collection feature, which removes explicit object deletion from the language. This strengthens the thesis that base functionality *cannot be implemented oblivious* of some crosscutting concerns as it is semantically coupled to it.

It is interesting to note that their aspects in order to be reusable highly rely on *reflective techniques*. Drawbacks of this approach are a performance penalty and that their framework is only usable for reflective systems. A consequence of the latter is also that the approach relies on strict encapsulation of persistent object state through getter and setter methods.

The paper also states that *aspect interaction* in AspectJ can be problematic, especially if the numbers of aspects grow. Examples are tracing, caching and the persistence aspect, all affecting the same joinpoints. For this interaction, manual ordering using the "`dominates`" (`declare precedence`) construct in AspectJ seems not to be sufficient, especially as significant support for the detection and resolution of interactions is still missing.

Although this review of case studies might not be exhaustive, it nevertheless allows to identify some common problems of AOP researchers met when performing these case studies. We will discuss these problems in detail in the following subsections.

2.2 A Critique of AOP

Although there is consensus that on the one hand base code modularity is improved by removing tangled code and localizing crosscutting concerns thus easing their evolution, aspect-orientation has several deficiencies which will be discussed in the following. Although the methods of achieving good modularizations and the supporting languages and programming paradigms changed since Parnas' famous paper [15], the quality criteria derived from it are still valid today. These software quality criteria will now be used to examine whether aspect-orientation indeed *in general* improves quality of software.

Scalability: Does aspect-orientation allow programmers to independently develop distinct modules, including aspects?

Maintain-/Evolvability: Does aspect-orientation improve evolvability of code? Two facets have to be considered in this context—evolution of crosscutting concerns (aka aspects) and evolution of core (functional) concerns.

Comprehensibility: Does aspect-orientation promote comprehensibility of modules? Again traditional modules and aspects and especially their composed effects have to be considered.

Reusability: Does aspect-orientation promote reuse of modules, including crosscutting concerns?

From the above summary of the case studies we have seen that aspect-orientation can also have counter-productive effects, at least in its current form.

Atlas: The Atlas case study gives interesting insights in the development process with aspects. Here the considerations of the *known-by relation* (a dependency relation) and the restriction of the usage of aspect-oriented constructs due to the reduction of understandability have to be stated.

The authors explicitly state that aspects should *only augment* the control flow (this includes that advice should not throw exceptions[2]) and respect the contracts of augmented code. It is also stated that tool support can be very helpful when developing with aspect-oriented techniques. Note that in this case study a new system was developed from scratch with aspects as modules in mind, thereby potentially influencing the structure of the underlying application—thus the application is not 'oblivious' per definition.

Transaction Control: In the study of aspectizing transactional behavior, the *difficulties of obliviously aspectizing crosscutting concerns* become apparent. It can be argued that for some concerns, although syntactically crosscutting, the semantics is an integral part of the base behavior. Thus syntactical separation might be (partly) possible, but a (semantic) coupling remains; in the case study the transactional code could be extracted to an aspect, but resulted in non-intuitive artifacts of interface separation; especially exception handling has to be mentioned in this context.

More importantly, the *semantic coupling results in non-obvious aspect-base dependences*. Thus evolution of base classes can easily break aspects. The authors argue that supporting tools might considerably lighten these problems by displaying relevant aspects.

Persistence and Distribution: In this case study, AspectJ has been criticized to lack constructs to write aspects general enough to avoid structurally duplicated code (especially in the context of inter type declarations). While this could be a language specific problem, the other reported observations are important problems of AOP in general.

AOP has been recommended as a mechanism to deal with a questionable design. This case study seems to contradict this statement, as the importance of careful design is explicitly emphasized. Lack of careful design especially of aspects *and their interactions* is highlighted as an important problem observed during the refactoring of the Health Watcher System.

The study shows that adding behavior by using aspects can be a very complex task. It explicitly reports about interactions and dependences of distribution and persistence aspects (via object synchronization). Even for these few aspects (applied to different layers of the system!) understanding these interactions is far from trivial.

Persistence as an Aspect: The creation of a reusable aspect-oriented persistence framework further strengthens the *restricted view on obliviousness*. Here retrieval and deletion of objects explicitly have to be handled by the application, thus again resulting in an application aware of a persistence mechanism. However, the implementation of this mechanism can be aspectized.

Although this case study does not report about coupling and evolution problems, another interesting aspect of aspect-orientation is discussed here: *interaction of aspects*. Currently there is only limited support to detect or analyze such interactions.

In the following we will examine the problems of aspect-orientation underlying the problems reported in the case studies in details and discuss them in the context of Parnas' module criteria.

2.2.1 Scalability

Parnas argued that modules should allow parallel development of different modules with little need for communication. Aspect-orientation improves parallel development, as requirements

[2]Note that AspectJ 0.4 had no `around` advice.

resulting in crosscutting concerns now can be localized in an aspect. This localization in different files improves parallel development of crosscutting concerns (tangling hinders parallel development). Examples seen in the discussion of the case studies are error handling, distribution, transactions or persistence. Before, invasive changes of all affected modules have been necessary, potentially resulting in conflicting edits when other teams changed the affected files as well.

However, there are also two hidden drawbacks here concerning correctness. First, the semantics of a given module is no longer apparent from neither base or aspect code alone—only the combination of both describes the resulting semantics. However, aspects and base are *syntactically decoupled* but non-obviously *semantically coupled*. Thus for correctness reasons, base and aspect programmers cannot be oblivious of each other, as also shown in the case studies. In fact, it seems important that aspect and base programmers know of each other and also communicate.

Assume different aspects are developed independently from each other, but affect the same base module. It is not guaranteed that the effect of applying several aspects to a base module is as expected, as aspects might conflict. Detection and correction of these conflicts however requires global system knowledge, thus hindering parallel system development and reducing understandability, maintainability and evolvability of the system.

From these observations we derive general problems of aspect-orientation which will be addressed in this thesis:

P1—Semantical Aspect-Base Coupling: Only aspects and base code together define system semantics. As aspects and base code are syntactically decoupled programmers have to be aware of existing dependencies.

P2—Aspect-Aspect Interaction: Aspects potentially interfere, but interference is hard to detect and resolve.

These properties of aspect-oriented systems can compromise system correctness and considerably harden parallel development. Necessary communication overhead is not removed, but only hidden.

2.2.2 Evolvability

Comparable to parallel development localization of crosscutting concerns in aspects considerably eases changing crosscutting concerns. Thus for crosscutting concerns indeed evolution now is much easier, as an invasive change of many modules in the system is no longer necessary.

However, for current aspect-oriented languages aspects and base in general are—at least semantically if not syntactically—coupled. This coupling in turn results in evolution problems, as also outlined in the Atlas case study: evolving base code can easily break aspect semantics. Thus evolvability of crosscutting concerns is improved, but at the cost of having to carefully check effects of base code edits on aspects. For the transactions case study the authors stated that tool support would be helpful to alert programmers of all affecting aspects and would also allow to check if aspects still work as expected after (base code) edits. Summarized, this can be stated as the following problem:

P3—Evolvability and Aspect-Base Coupling: Aspects potentially rely on (internals of) base modules. Evolution of the base code can alter and break aspect semantics.

2.2.3 Comprehensibility

In [4] Clifton and Leavens define modular reasoning as "the process of understanding a system one module at a time". The ability to modularly reason about software strongly supports

comprehensibility. A language supports modular reasoning, if "the actions of a module M can be understood based solely on the code contained in M along with the signatures and behavior of any modules referenced by M".

If we examine the most popular aspect-oriented language AspectJ based on this definition, we can observe that AspectJ in contrast to Java *does not support modular reasoning*. This can be easily justified by quickly reviewing the possible impact of aspect-oriented language constructs available in AspectJ.

Inter Type Declarations can add new methods (as well as fields) to classes potentially overriding methods from super classes and thus potentially resulting in changed lookup behavior for the modified class *and its subclasses*.

Advice can access—and also modify—accessible joinpoint context (e.g. method parameters), or more directly change the control flow by throwing exceptions or (in the case of `around`-advice) by not calling `proceed()`.

The important thing to note is that neither members added by aspects nor applied advice is directly referenced by the target module. Thus it is not possible to understand the semantics of a given module without knowledge about any applied advice or the introduced method.

This assumes that we still define a module as a class, without applying aspects. However, if we add all applying aspects to our notion of module, there is the problem that such modules—in the case of AspectJ—are only implicitly defined. The programmer needs to know which aspects apply to know the module boundaries.

During development of Atlas, the programmers ran into exactly these problems when they were using introductions and advice undisciplinedly—understanding the system became a hard task. Their solution was to only use a very restricted form of aspects by only augmenting but not adapting aspect semantics.

To summarize aspects as well as base code can be understood more easily, however the semantics of their combination is considerably harder to comprehend:

P4—System Comprehensibility: The semantics of the woven system only becomes comprehensible with global system knowledge. In particular a programmer of each module has to be aware of any applied aspect.

2.2.4 Reuse

For completeness reasons reuse is also addressed. Aspect-orientation improves reuse compared to traditional approaches. On the one hand tangled code from crosscutting concerns is removed from the base modules allowing for easier reuse of these modules in a different context. As shown by the persistence case study the same is true for crosscutting concerns as well. It is now possible to implement such concerns in separate modules which can also be reused, although reuse can be hard for application specific aspects, as they are usually tightly coupled to the base code they apply to. Currently libraries of standard crosscutting concerns are not yet available, but this might change in the future. For example one can imagine to provide a library in which abstract standard aspects like Tracing, Persistence, Caching, etc. are bundled. To use these aspects a programmer only has to provide a concrete sub-aspect where pointcuts are defined to connect the functionality defined in the abstract library aspect with the particular base system.

2.2.5 Discussion

It has been argued that crosscutting concerns are considerably clearer and easier to understand and evolve if localized in an aspect. The same has been claimed for the base code as the tangled code implementing the crosscutting concerns has been removed.

However, once a problem occurs, i.e. once the woven system does not behave as expected, programmers have to *examine the code to find out where the problem comes from*. Although an aspect might cleanly encapsulate a crosscutting concern, during debugging or in the maintenance phase aspects might considerably reduce comprehensibility and thus hinder system evolution as programmers now have to *read non-composed code to deduce the semantics of composed code*. This can be impossible without knowledge of applying the semantics changing aspects, and thus now the problems described above are highly relevant. Consequently the problems concerning scalability, system evolution and comprehensibility have to be addressed to make aspect-orientation a safe technique to use.

This thesis will discuss each of the problems in detail in subsequent chapters and also introduce a tool-based approach to deal with them. Tools can help to automatically extract information about relevant applied aspects to give programmers the necessary module references to understand semantics of base code with applying aspects, without putting the burden of requiring global system knowledge on them. The AspectJ development environment, the AspectJ Development Tools (ajdt) [2], basically on-the-fly provide programmers with information about applying aspects.

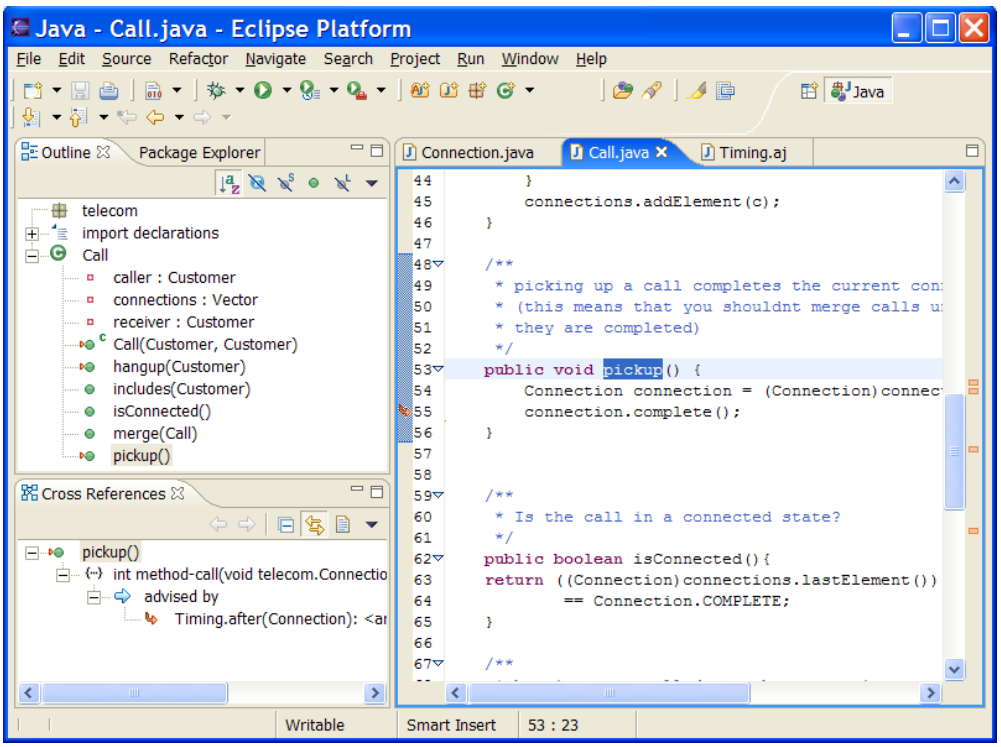


Figure 2.1: AspectJ Development Tools in Eclipse

The screen shot shown in Figure 2.1 shows the outline view available in ajdt. As can be seen for each base module, affecting advice or any other relevant aspect construct can be seen and traced back to the relevant aspect code. However, this information is merely syntactic, ajdt does not offer any semantic information about applying advice, i.e. whether advice changes the base module semantics or not.

In this thesis we close this "semantical gap" by examining how techniques of program analysis can be used to derive useful semantic information from the code to support programmers in understanding and evolving aspect-oriented systems. The approach described here

explicitly refrains from restricting aspect-oriented languages, but provides additional information, in the spirit of (but extending) the information available in ajdt. Some of the techniques proposed here might be useful to integrate in a compiler (comparable to the `Xlint` options of the AspectJ compiler), others are clearly designed for integration in a development environment. If not explicitly stated otherwise, all described techniques have also been implemented as a set of Eclipse plug-ins forming the AOPA[3] analysis suite.

The problems described so far have been recognized by the research community, and several approaches have been suggested to counter them, however most of them at the language level, i.e. researchers try to improve the languages to avoid problems rather than deal with the problems for existing code. Most approaches either try to re-introduce missing information for the programmer by augmenting programs (e.g. by defining interfaces explicitly naming aspect-base relations) or restrict the expressive power of aspect language to only allow 'safe' aspect operations.

All restricting approaches have been criticized as unnecessarily removing expressive power from the language so reducing applicability of these constructs to encapsulate crosscutting concerns. As dynamic aspect-oriented systems (i.e. systems where runtime weaving is possible) are a major research topic today it seems hard to convince programmers to give up some expressive power for "theoretical reasons". The next section in detail discusses related work from other researchers addressing the deficiencies of aspect-orientation.

2.3 Deficiency Analysis of AOP in Literature

By now the AOP community has recognized some of the problems discussed in the previous chapter. In this section we review and summarize related work addressing the outlined problems of aspect-oriented techniques. Most of this work is addressing the *lack of modular reasoning* and the fact that aspect-orientation often violates the *information hiding principle* due to the crosscutting nature of aspects and the lack of clearly defined modules.

2.3.1 Modular Reasoning

In [4] Clifton and Leavens define modular reasoning as "the process of understanding a system one module at a time". A language supports modular reasoning, if "the actions of a module M can be understood based solely on the code contained in M along with the signatures and behavior of any modules referenced by M". This definition has been used by other researchers and will also be used in this thesis.

Spectators, Assistants, Obliviousness and Modular Reasoning

Clifton and Leavens examined the impact of aspect-orientation on modular reasoning. They observe that Java supports modular reasoning but *AspectJ does not* as advice changing the semantics of a given class is not explicitly referenced by this class. Nevertheless, the semantics of the resulting system changes if compiled together with the aspect. Thus the programmer has to examine the complete system including all potentially applying aspects in order to understand a single module. This is clearly in conflict with modular reasoning.

To deal with this problem, they propose to classify aspects as spectators and assistants, where spectators preserve the semantics of the original code, while assistants may change it. As a module is not comprehensible without knowing about assistants, they add a mandatory language construct to explicitly state that a module accepts a specific assistant, thus adding an explicit reference of the aspect to the modified module. To deal with large numbers of modules affected by an assistant, these explicit references may be defined in an external aspect map.

[3] Aspect-Oriented Program Analysis

Thus modular reasoning is re-established. Research recently also proposed a way to check whether an aspect is a spectator or assistant by applying escape analysis to advice [17], thus allowing to construct a compiler including the necessary checks.

Unfortunately this approach has some important drawbacks:

- First, currently an implemented system is not available and getting acceptance for a feature restriction (mandatory assistant acceptance) for an existing language (i.e. AspectJ) in general is hard to achieve.
- Second, the approach requires that programmers maintain the aspect map. Maintenance of this file again can be tedious, and modular reasoning requires checking this file for applying advice (which is only implicitly referenced). Thus tools are necessary to support aspect map maintenance and display.
- Third, the proposed aspect classification as spectator or assistant is not decidable in general, thus a compiler has to assume assistants if in doubt thus potentially overestimating the set of assistants and rendering aspect map maintenance more tedious. As the discussion in their paper shows, even a manual classification is not always clear.
- Finally, explicit acceptance of aspects is in conflict with the obliviousness of aspect-orientation as stated by Filman and Friedman [7]. In follow up work [5], this conflict is discussed by stating that obliviousness and modular reasoning in general are in tension.

In followup work [5] Clifton and Leavens discuss commonalities between dynamic dispatch[4] in object-oriented languages and advice application. As Filman and Friedman pointed out, dynamic dispatch also is a kind of obliviousness, as the actually executed method is statically unknown. However, for object-orientation the concept of behavioral sub-typing [12], i.e. the demand that an overriding method respects the contracts of the overridden method, restores modular reasoning. Informally overriding methods should not have a "surprising" behavior for the programmer, more formally the contracts of the original method should be respected by any overriding method.

This concept in part can be transported to aspect-orientation, if aspects are considered which only add behavior but leave the behavior of the original code unchanged, similarly to behavioral sub-typing. However, sometimes advice is explicitly used to adapt behavior of a given module. Here the analogy to overriding methods is lost. For object-oriented programs adaptation of behavior is usually implemented using wrapper classes, not sub-typing. Wrappers however are not an oblivious concept. Clifton and Leavens use this analogy to justify that it might be necessary to sacrifice some obliviousness for adapting advice to gain modular reasoning.

In [9] Kiczales and Mezini argue that, despite of the above, modular reasoning is *improved* by aspect-oriented programming, if programmers are willing to accept a changed notion of module interface. The paper suggests *aspect-aware interfaces* for modules. These interfaces are similar to the interfaces of Open Modules described in Subsection 2.3.2, but in contrast they are *derived on demand* once the complete system configuration is known, thereby capturing any necessary internal pointcuts and adding them to the export list.

As these interfaces are based on global system knowledge, they again re-introduce the missing information for programmers to allow modular reasoning by associating each traditional method in the interface with affecting advice, as outlined in Figure 2.1 (taken from [9]).

Using a hierarchy of geometrical shapes as an example, the paper argues that with this information available, a crosscutting concern like “display update signaling” is made explicit, thus easing modular reasoning and avoiding global analysis of the implementation otherwise

[4] Dynamic dispatch is used as a synonym of late binding in object-oriented programming, where the choice which method is actually executed is deferred until the actual runtime type is known.

Listing 2.1: An Aspect-Aware Interface for a class Shape

```
1 Shape {
2    void moveBy(int, int) : UpdateSignaling--after returning
3                            UpdateSignaling.move();
4    ...
5 }
```

necessary to uncover the underlying invariants (one update per set of related changes) of this concern.

While for the given example the discussion is conclusive, there are some important limitations also outlined in the paper. First, the proposed aspect-aware interfaces are only described for execution joinpoints and `after`-advice. The authors outline a generalization of aspect-aware interfaces as future work. However, for other kinds of joinpoints and advice three important general problems have to be dealt with:

Annotation Granularity: The suggested interface extension is on a per method basis, matching the granularity of execution joinpoints. The papers discussed that a similar annotation on a per class basis would also be possible and correct, but not satisfactory as programmers had to examine all—i.e. also unaffected—methods to get an actual overview of aspect effects.

However, if other kinds of joinpoints are considered, like for example `get/set` or `call` joinpoints, the matching granularity is the *statement level*. As interfaces have no matching construct at this level of granularity, a necessary annotation always has to deal with this *granularity mismatch*.

Scalability: Aspect-aware interfaces might become very unreadable, if *several pieces of advice* are associated with a single method. This might well be the case, if one considers classical examples like Authorization, Tracing or some other Observer concern. Thus an additional filtering is needed nevertheless.

Other Types of Advice: The information provided in aspect-aware interfaces indeed helps programmers to reason about `after`- and `before`-advice. However a core property for this kind of advice is that the original code in general (i.e. if advice does not throw an exception) is executed.

This property is not valid for `around`-advice. As a consequence, the programmer in detail has to examine associated `around`-advice to learn about the actual behavior of his program as `around`-advice can completely invalidate the semantics of a given base method.

The information provided in such interfaces can considerably help programmers to estimate the effects of aspects on a given base module. It is interesting to note that construction of the proposed aspect-aware interfaces needs *global system knowledge*. Thus these interfaces have to be constructed as a pre-processing step to enable modular reasoning.

2.3.2 Modules in Aspect-Orientation

A *(software) module* is usually defined as a functionally or logically closed building block of a software system with a clearly defined interface which hides its internal implementation details. Modules ideally are understandable by only considering interfaces of used modules (i.e. modules allow modular reasoning), have a high internal cohesion, but are loosely coupled.

When considering aspect-oriented systems, it is relatively hard to define what a module actually should be. The first choice—to keep the understanding that each class (and similarly each aspect) form a separate module—has two important problems:

1. For most AO languages, classes do no longer hide their internals, as aspects can access even joinpoints in private methods, thus breaking the *information hiding principle*; i.e. aspects effectively bypass class interfaces.

2. As aspects access internals of classes, they are tightly coupled with the classes they access. This is clearly not desirable for separate modules.

As a consequence this simple straight-forward notion of modules has been questioned.

The following discusses papers which examine this question in more detail and also define a new notion of module for aspect-oriented programming. Although this thesis does not directly address this question, the problem analysis in these papers is important to understand the problems current AO languages (including AspectJ) suffer from.

Open Modules

The problem related to the use of AOP with respect to independent evolution of modules and the preservation of necessary invariants is examined by Aldrich et al. in [1]. Two core research questions addressed in the paper are formulated as follows. How can developers specify interfaces for a library or module that permits as many uses of advice as possible, while still

- allowing the module to be changed in meaningful ways and without affecting clients (including aspects!), and

- ensuring correctness properties of the implementation.

To achieve these goals, the authors propose a new module concept called *Open Modules* restricting advice in general but allowing developers to consciously override these safe settings. More precisely Open Modules extend traditional modules by also allowing to export joinpoints. For these explicitly exported joinpoints the module maintainer guarantees to maintain their semantics. Advice thus can safely be used when extending the explicitly exported pointcuts or any interaction of non-module and module functions from the interface.

To prove that the informally introduced Open Modules concept satisfies the claims made above, the paper defines a small step operational semantics for an aspect-oriented functional core language called *TinyAspect* and proves the above based on this semantics. Semantic equivalence rules are defined which can be used to verify that modifications of a module don't break clients or invalidate client correctness. However, the model permits programmers to ignore the Open Module rules and access internal (i.e. not exported) module events as well, however thereby losing the guarantees provided otherwise.

As the underlying language is functional, side effects of aspects are not considered. Further on the pointcut language is limited as only `call`-joinpoints are captured and dynamic pointcuts are out of the scope of the paper. Finally, Open Modules rely on clients to augment interfaces with exported joinpoints. This again is in conflict with the obliviousness property stated for aspect-oriented approaches as outlined in [7].

Adding Open Modules to AspectJ

In [14], Ongkingco et.al. built on Aldrich's idea of Open Modules [1] and implemented a matching module concept for AspectJ, based on the alternative *abc* compiler [3].

Their contributions include an extension of Aldrich's notion of Open Modules to the complete AspectJ pointcut language. They also provide a way to define a sound composition of

modules which also results in a clearly defined aspect precedence order—which is an important problem in plain AspectJ.

Their notion of modules is an optional extension of AspectJ, and allows to explicitly define the joinpoints exported by a module. Modules are strictly hierarchically composed, their aggregations structure is a tree. Each module can either *constrain* or *open* access to contained submodules. Each class and aspect are part of at most one module. As modules are arranged in a tree, and aspect member declaration order defines aspect precedence within the module, a global total precedence order is defined by construction.

To implement this new module concept for AspectJ, the *abc* team added a single new primitive pointcut—called `vis`—to check if a traditionally matched joinpoint is visible to the matching aspect. `Vis` is evaluated by checking against the module declarations. Each module can explicitly export relevant joinpoints or define *friend aspects* which have full access to all module joinpoints. Thus a fine-grained joinpoint access control is possible.

Although this module concept for AspectJ is a big step ahead, the programmer still has to be aware of the chosen Aspect precedence, and handle this precedence with care. For conflicting aspects in different modules, aspect precedence is defined, but not obvious to the programmer. The methods presented in Chapter 6 can be used to leverage this issue and alert programmers when precedence order for aspects is relevant for a given application.

On the Criteria to be Used in Decomposing a System into Aspects

The analysis and critiques addressed above are also confirmed by a study of Sullivan et al. [19]. In a case study they compare and evaluate the design and implementation of a standard object-oriented implementation with an oblivious aspect-oriented implementation and an aspect-oriented design based on design rules, which express how base code should be organized in order to allow aspectized implementation of crosscutting concerns.

An empirical assessment of their implementations showed that the classical aspect-oriented approach is superior compared to the object-oriented approach but suffers from several severe problems. Writing pointcuts for an oblivious code base can be hard as commonalities are not necessarily captured by fitting naming conventions resulting in bloated highly coupled pointcut definitions. Pointcuts potentially have to expose internal implementation details of a module (and thus are tightly coupled to it). Even worse, necessary joinpoints might not be accessible at all.

More abstractly speaking, the *aspects conflict with the information hiding approach* taken by traditional modules and thus suffer from all the problems related to the resulting coupling with base module internals. In contrast to the approach advocated by Kiczales and Mezini to calculate aspect-aware interfaces by the actual usage of joinpoints through aspects Sullivan et al. suggest defining design rules as a kind of predefined interface based on the potentially changing crosscutting concerns. Their design rules follow an information hiding approach in the sense of Parnas, as each design rule is derived from a potentially changing crosscutting concern. However, these rules are only informally defined (comparable to design patterns).

Their assessment of the design rule approach based on a case study shows promising results avoiding some of the problems related to obliviousness at the cost of imposing rules on the base programmers how to write their code. However, unanticipated concerns naturally do not anticipate on the benefits of such predefined interfaces and their approach rather targets the design level in contrast to the code level. As their approach attacks the problem on the design level, we see their work orthogonal to ours.

2.3.3 Summary

This chapter gave an overview of available case studies of aspect-oriented systems. From these case studies four main problems of current aspect-oriented approaches comparable to

the popular aspect-oriented language AspectJ were derived:

P1: Aspect-Base Coupling

P2: Aspect Interaction

P3: System Evolution

P4: Base Comprehensibility

In the following this thesis will introduce tool-based techniques to lighten these problems.

Bibliography

[1] ALDRICH, J. Open Modules: Modular Reasoning about Advice. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2005), vol. 3586 of *Lecture Notes in Computer Science (LNCS)*, Springer, pp. 144–168.

[2] ANDY CLEMENT, A. C., AND KERSTEN, M. Aspect-Oriented Programming with AJDT. In *Proceedings of AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003* (July 2003).

[3] AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (New York, NY, USA, 2005), ACM Press, pp. 87–98.

[4] CLIFTON, C., AND LEAVENS, G. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning, 2002.

[5] CLIFTON, C., AND LEAVENS, G. T. Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning. Tech. rep., Iowa State University, October 2002.

[6] DIJKSTRA, E. W. The structure of the 'THE'-multiprogramming system. *Commun. ACM 11*, 5 (1968), 341–346.

[7] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. Tech. rep., RIACS, 2000.

[8] KERSTEN, M., AND MURPHY, G. C. Atlas: a Case Study in building a web-based Learning Environment using Aspect-Oriented Programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1999), ACM Press, pp. 340–352.

[9] KICZALES, G., AND MEZINI, M. Aspect-Oriented Programming and Modular Reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 49–58.

[10] KIENZLE, J., AND GUERRAOUI, R. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming* (London, UK, 2002), Springer-Verlag, pp. 37–61.

[11] LIPPERT, M., AND LOPES, C. V. A Study on Exception Detecton and Handling using Aspect-Oriented Programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ACM Press, pp. 418–427.

[12] LISKOV, B. H., AND WING, J. M. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[13] NAVATHE, S. B., AND ELMASRI, R. A. *Fundamentals of Database Systems with Cd-Rom and Book*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] ONGKINGCO, N., AVGUSTINOV, P., TIBBLE, J., HENDREN, L., DE MOOR, O., AND SITTAMPALAM, G. Adding Open Modules to AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development* (New York, NY, USA, 2006), ACM Press, pp. 39–50.

[15] PARNAS, D. L. On the Criteria to be used in Decomposing Systems into Modules. *Commun. ACM 15*, 12 (1972), 1053–1058.

[16] RASHID, A., AND CHITCHYAN, R. Persistence as an Aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 120–129.

[17] RINARD, M., SALCIANU, A., AND BUGRARA, S. A Classification System and Analysis for Aspect-Oriented Programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (New York, NY, USA, 2004), ACM Press, pp. 147–158.

[18] SOARES, S., LAUREANO, E., AND BORBA, P. Implementing Distribution and Persistence Aspects with AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM Press, pp. 174–190.

[19] SULLIVAN, K., GRISWOLD, W. G., SONG, Y., CAI, Y., SHONLE, M., TEWARI, N., AND RAJAN, H. Information Hiding Interfaces for Aspect-Oriented Design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM Press, pp. 166–175.

3 Static Crosscutting

AspectJ supports two different crosscutting mechanisms, static and dynamic crosscutting. While dynamic crosscutting (pointcuts and advice) influences runtime behavior of a system, *static crosscutting modifies the static system structure*. In this chapter this thesis will examine how static crosscutting can be used and what effects it can have on system semantics.

As joinpoints in AspectJ can be mapped to a statement in the source code (joinpoint shadow), dynamic crosscutting follows the modified structure resulting from evaluating static crosscutting constructs. Thus analysis of static crosscutting is a prerequisite to yield correct results for any (static) analysis of dynamic crosscutting constructs. This chapter will examine algorithms for a fundamental analysis of these effects, which will be used in the remainder of this thesis. An earlier version of this work has been published in [22, 23].

3.1 Background

Inter Type Declarations (as a part of Static Crosscutting) as present in AspectJ are a "feature" stemming from the limitation of Java to define additional members (i.e. methods and fields) for a class *outside of the file containing the class definition*. There is a one-to-one mapping between public classes and files in Java, i.e. class definitions cannot be spread over several files. For C++ this is (and always was) possible.

In [8] the ability to non-locally add additional members to existing classes has been termed *Open Classes*.[1] There, open classes are defined as classes

> "to which new methods can be added without editing the class directly. An open class allows clients to customize their interface to the needs of the clients application. Unlike customization through subclasses, in-place extension of classes does not require existing code referencing the class to be changed to use the subclass instead."

The open class mechanism in AspectJ has no replacing semantics as can be found in Hyper/J [24]. It is only possible to add additional members (i.e. non-conflicting methods and fields) to existing classes.

[1] Although [8] cites [14] as the origin of the term.

It is a common misconception that adding new non-conflicting members cannot affect the semantics of an existing program. For object-oriented programs this is not the case as we will see in this chapter. In short, adding a new method can *override, overload or hide* a method previously inherited from a superclass and thus lead to execution of a different method as before. Similarly newly added fields may hide fields from supertypes also potentially leading to unexpected behavior.

This chapter will examine the potential impact of non-local member additions to an existing class in the following, using AspectJ as example. An interference criterion is presented to state whether an existing program is affected by externally added code or not. Additionally this thesis demonstrates how the generated information can be used to help programmers in identifying the reasons for unexpected system behavior due to inter type declarations.

3.2 Static Crosscutting in AspectJ

Static Crosscutting in AspectJ consists of three relatively different features:

- Inter Type Declarations,
- Hierarchy Modifications and
- Exception Softening.

Their commonality is that they all affect the static structure of the system. This section in short describes these features, for a detailed discussion refer to the AspectJ manual or [13]. An interesting application of static crosscutting in AspectJ is a simulation of a limited form of multiple inheritance. This application is discussed briefly in this section.

3.2.1 Inter Type Declarations

Inter Type Declarations actually implement open classes as defined in [8]. They can be useful to break several limitations of Java. Often advice needs some data to be kept with each instance of an adapted class. In that case it can be useful to declare new attributes or methods in the context of the target class. The aspect can then use these attributes and methods for its own calculations. This is close to benefits of Open Classes as outlined in [8]—the only difference is that in general advice is the client for the adapted interface.

Programmers define inter type members in aspects similarly to regular Java class members. The only difference is that inter type members are defined using qualified names to specify the target class they are introduced to.

Example 3.2.1 *The code below for example introduces method `f()` and field `x` to a class `C`. These members are then also available for other (plain Java) classes.*

```
1 aspect A {
2   public int C.x;
3   public void C.f() {
4     // Java Code
5   }
6 }
```

Method `f()` can be written as if defined in class `C`, i.e. it can access all members of `C`. Additionally members of the defining aspect can be accessed in the body of inter type methods. Note that AspectJ's inter type declarations also allow to define interfaces as targets of inter type declarations.

Inter type declarations of different aspects can conflict with each other, if members with identical signatures should be introduced to the same target class from several different aspects. In such cases *aspect precedence* determines which member is finally introduced—only the members introduced by the dominating aspect survive in this case. For aspects not explicitly ordered by precedence statements conflicting inter type declarations are considered a compiler error.

3.2.2 Hierarchy Modifications

The `declare parents ...extends` clause allows to redefine the inheritance relation for classes within certain constraints. Informally, type correctness has to be maintained when changing the superclass (i.e. all previously legal method or field accesses must still be legal). As a consequence a class can only be declared to extend a (subclass of a) sibling class. For details refer to the AspectJ manuals.

Changing the supertype has two important effects on system semantics. First, the *method lookup may change*, i.e. calls in the system now potentially execute a different method. Second, this modification also affects the *type of objects* instantiated from classes in a modified hierarchy. This is relevant as expressions using Java's `instanceof` construct potentially change their result.

Besides changing the superclass, AspectJ also allows to declare that a class implements several interfaces using the `declare parents: ...implements` clause. Note that this will result in compiler errors if the class interface does not implement all methods declared in the interface. However, together with inter type declarations to interfaces, this allows an interesting application as outlined in Section 3.2.4. While declaring interfaces to be implemented cannot change method lookups, it also changes the type of implementing classes (and their subclasses).

3.2.3 Exception Softening

Besides the above constructs AspectJ allows exception softening, i.e. to change Java's checked exceptions to unchecked runtime exceptions. This is often necessary if advice throws a checked exception not declared in the context of an adapted joinpoint. Here Java's catch-or-specify policy for exceptions would result in a compiler error without exception softening.

Example 3.2.2 *The code below will transform any `IOException` thrown by a call to method `f()` into a runtime exception of type `SoftenedException`.*

```
1 aspect A {
2   declare soft : IOException : call(* f());
3 }
```

Exception softening is always bound to a pointcut definition, to allow programmers to restrict its effects to the advice context. This is necessary as exception softening can change system semantics—*the softened exceptions now* are *runtime exceptions and thus potentially caught by different catch blocks.*

The analysis techniques examined in this chapter focus on lookup changes. Changes to `instanceof` predicates or exception handling behavior both are a result of type changes, and are only briefly discussed. A precise analysis of these effects is not decidable and not in the scope of this thesis.

3.2.4 Inter Type Declarations to Interfaces

While using inter type declarations to support aspects is relatively straight forward, the use of interface inter type declarations to simulate a limited form of multiple inheritance is worth a more detailed discussion.

Simulation of a limited form of multiple inheritance is possible as in AspectJ the targets of inter type declarations can also be interfaces. Together with Java's multiple interface inheritance, inheriting *code* from multiple parents can now be achieved.

AspectJ allows to modify the type hierarchy by using the `declare parents ...implements` construct. This feature is typically used if a single target class for inter type declarations cannot be identified. In this case, an empty *marker interface* is declared to be implemented by all relevant target types. The necessary members are then introduced to this interface using inter type declarations and are thus available in all relevant types. This technique has been called the *marker interface idiom* in [11]. It is especially useful to create reusable aspects, where the set of relevant target classes is not known in advance. The marker interfaces idiom is a solution in this case, as the user can mark up the relevant types using this mechanism and at the same time provide all necessary auxiliary data fields and methods.

Remark: *Note that instead of the marker interface idiom a per-object association of aspects is an interesting alternative. Per default aspects are singletons. However, it is also possible to instantiate aspects on a per-object basis (using AspectJ's* `perthis` *and* `pertarget` *constructs). In his book "AspectJ in Action" [13] Ramnivas Laddad shows that in general inter type declarations can be replaced with aspects instantiated per instance. Instead of introducing members to a class the members are simply declared local to the aspect. While introduced members might be easier to understand—these members effectively become part of the target class—per-object instantiated aspects can be a more elegant solution, especially if a common supertype to introduce members to is not known. A second benefit of only using per instance aspects is a cleaner language design, as static crosscutting constructs are then superfluous. For AspectJ, the choice between inter type declarations and* `perthis` *instantiation is a choice between simplicity and elegance [13].*

Multiple Inheritance

Multiple inheritance is a feature which can be used to model classes combining several *different concepts*. From a software engineering perspective, one would prefer to model each of these concepts separately. Additionally a mechanism to combine relevant concepts is needed. Multiple inheritance offers this combination mechanism. In the context of an object-oriented programming language, a class can inherit and thus combine behaviors of *several* superclasses.

Example 3.2.3 *A common example is the hierarchy presented in Figure 3.1. Here a class* `Person` *has two subclasses:* `Student` *and* `SportsMan`*. Both classes define some specific behavior, the* `Student` *class defines a method* `study()`*, the class* `SportsMan` *a method* `doSports()`*. However, there are also students studying sports. For languages providing multiple inheritance this is easy to model. A new class is introduced which inherits from both* `Student` *and* `SportsMan`*—the class* `SportsStudent`*.*

However, some object-oriented languages (including Java) do not offer multiple inheritance. Multiple inheritance has often been criticized to be too complex, especially if *structurally equivalent code* (i.e. methods with the same signature or fields with the same name/-type) is *inherited from several parents*.

Example 3.2.4 *Assume that both class* `Person` *and class* `Student` *have an own implementation of a method* `payTaxes()`*. Both methods are accessible for class* `SportsStudent`*. Which method however should be/is called?*

```
1  class Person {
2    int ssn;
3    ...
4  }
5  class SportsMan extends Person {
6    doSports();
7    ...
8  }
9  class Student extends Person {
10   study();
11   ...
12 }
13 class SportsStudent
14   extends Student, SportsMan {
15   ...
16 }
```

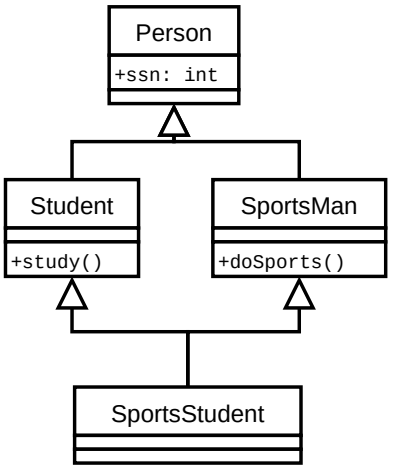


Figure 3.1: Example for Multiple Inheritance—SportsStudents

To resolve this issue the implementation used in the subclass is chosen among the available implementations according to *dominance rules* [12]. For multiple inheritance the lookup at runtime using the dominance rules is a comparably complex task, resulting in less understandable code as the actually executed method is not always obvious. This is one of the reasons why multiple inheritance has been criticized and not incorporated in Java, although there are relevant applications for it.

Example 3.2.5 *An example of a beneficial application of multiple inheritance are the event handler classes in Java Swing. For event handlers, the Java API in general offers an interface, for example* `MouseListener`*, but additionally abstract classes implementing this interface, for example* `MouseAdapter`*.*

The reason why the API defines an abstract class and an interface is that several of the interfaces define more than one method (`MouseListener` *defines five), although for a particular application often only a single one of these methods is needed. To reduce the effort to implement a fitting event handler, the adapter classes exist, which define an (empty) default implementation for each method, i.e. to implement a listener only relevant methods have to be overridden by the implementing subclass. Subclassing the adapters however is only possible, if the new event handler class has no other superclass to inherit code from. In that case, only the interface can be used as Java lacks multiple inheritance.*

Open Classes as introduced with AspectJ now provide a way to re-introduce some of the benefits of multiple inheritance to Java using interface inter type declarations. With this feature it is possible to provide *default implementations*. Such widely usable methods can considerably reduce necessary work for classes implementing these interfaces.

Example 3.2.6 *For example it is possible to introduce the empty default listener methods to the listener interfaces thus eliminating the need for the adapter classes in the Swing event handler API.*

Modeling Alternatives

When implementing the `SportsStudent` example in Java, multiple inheritance is no option as Java only supports single inheritance for classes. It would be technically possible to add

the behavior of either SportsMan or Student to a common superclass (i.e. Person). From a software engineering point of view this is not satisfactory as now Person has methods study() and/or doSports(). These methods however do not specify behavior relevant for *any* person. A similar argument applies if a programmer either adds study() to SportsMan or doSports() to Student (thus linearizing the inheritance hierarchy). However this would remove either the pure Student or SportsMan class. Thus an inheritance based solution is no option either.

The usual way to deal with this problem is *delegation*. Each Student-object owns a SportsMan-object and delegates calls to doSports() to it, thus sharing the uniquely defined behavior. However, delegation in general suffers from another problem called *object schizophrenia*. The associated SportsMan-object is also a legal Person and thus has a unique identity (the Social Security Number). On the other hand it is only valid in conjunction with the Student object—with its own distinct identity. It seems tempting to avoid this problem by implementing SportsMan independent from the Person hierarchy. However, this will also sacrifice the possibility to model a SportsMan who is no student.

Simulating Multiple Inheritance with Aspects

Java allows multiple inheritance only for interfaces. As a result, an implementation of Java can never by ambiguous, as only the superclass can provide an implementation, even if several superinterfaces also define a method with identical signature. This changes when AspectJ inter type declarations on interfaces are used. Now again several valid implementations can be available. For AspectJ's interface inter type declarations such ambiguities are resolved statically at compile time. Therefore a dominating method is chosen using the following rules:

1. A matching implementation inherited from the superclass always dominates all other implementations.

2. Among implementations inherited from multiple superinterfaces, C++ dominance rules (cmp. [12]) are used to statically choose an implementation. This code is *copied* into the implementing classes.

3. In case of a non-resolvable or ambiguous lookup of an inherited interface method, AspectJ reports a compiler error.

Note that at runtime no complex lookup or dominance rules are used. The chosen implementation is either the one of the superclass or an implementation textually copied into the implementing class. In both cases standard Java method lookup is used at runtime to choose the method to be executed.

Due to the modeling problems briefly outlined above, some programmers requested to add multiple inheritance to Java. Inter type declarations now offer a similar feature useful to simulate a limited but relatively safe form of multiple inheritance. Aspects allow to define default implementations for interface methods. As interfaces in plain Java do not define behavior (only method signatures are allowed) Java allows classes to implement several interfaces. In combination with aspects and definition of default implementations this can be used to "inherit" behavior from multiple superclasses and interfaces.

Example 3.2.7 *It is possible to define an interface* SportsMan *and an aspect defining a default implementation of* doSports(), *and then let* SportsStudent *inherit from* Student *and* implement *the interface* SportsMan *thus also inheriting the implementation of* doSports(). *However, similar to the delegation approach, it is not possible to instantiate a* SportsMan. *However can be solved by defining an empty subclass of* Person *also implementing the* SportsMan *interface.*

```
1  interface I {
2    void x();
3    void y();
4  }
5  class A {
6    void n(A a) {
7      System.out.print("A.n()");
8    }
9  }
10 class B extends A {
11   int i;
12   void m() {
13     System.out.print("B.m()");
14   }
15 }
16 class C extends B {
17   public void x() {
18     System.out.print("C.x()");
19   }
20 }
```

```
21 class D extends B {
22   public void y() {
23     System.out.print("D.y()");
24   }
25   public void x() {
26     System.out.print("D.x()");
27   }
28 }
29 class E extends C { }
30
31 class F extends D {
32   void n(A a) {
33     System.out.print("F.n()");
34   }
35 }
36 class G extends B {
37   void n(A a) {
38     System.out.print("G.n()");
39   }
40 }
```

Figure 3.2: Original Class Hierarchy

Note that default implementations have to be clearly distinguished from "real" multiple inheritance. For multiple inheritance the method lookup is performed at runtime, i.e. this is a dynamic lookup. For default implementations, the aspect weaver decides at compile time which method is added to the resulting target class. The dominance rules for AspectJ therefore prefer a matching method defined in the superclass if available and otherwise use the dominating default implementation from one of the superinterfaces. The definition of this method is then textually copied to the target class implementing the interface.

3.2.5 Running Example

In the remainder of this chapter the simple class hierarchy defined in Figure 3.2 will be used as an example to demonstrate the effects of inter type declarations.

In the following this chapter in detail describes the problems due to inter type declarations to classes and due to hierarchy modifications, and presents an algorithm to detect these effects. While the results of this analysis can be directly used in a tool to help programmers reduce flaws in a software system, they are also important input information used by more elaborate analysis techniques of aspect-oriented software.

3.3 Theoretical Foundations

To capture the effects of aspects, analysis of static crosscutting is a necessary first step. What we want to know is *weather* an aspect modifies system semantics using static crosscutting or not. And if the aspect modifies system semantics, we want to know the *exact aspect influence*, i.e. *how* the aspect affects the semantics.

We base our analysis on previous work by Snelting and Tip. In [21] Snelting and Tip analyze the effects of composing class hierarchies to a new combined hierarchy for a set of existing clients of the two original hierarchies. This work was done in the context of *multidimensional separation of concerns* (Hyper/J)[24], where different concerns are implemented in

different hierarchies which are then combined to the final system using a set of programmer-defined composition rules. We will first briefly summarize their work and then show how this work can be applied to static crosscutting in aspect-oriented programming.

Snelting and Tip define class hierarchy as a set of classes together with an inheritance relation ($\mathscr{H} = (\mathscr{C}, \trianglelefteq)$). We write $C \trianglelefteq D$, if C is a subtype of D (or $C \triangleleft D$ if $C \neq D$). For hierarchies $\mathscr{H}_1$ and $\mathscr{H}_2$, their composition $\mathscr{H}_1 \oplus \mathscr{H}_2$ is defined as the union[2] of all classes in both hierarchies, where a class in the combined hierarchy contains the union of all members in corresponding classes. The two inheritance relations are order-embedded into each other.

During composition two different cases of interference can occur: static and dynamic interference. *Static interference* denotes conflicts in composition, e.g. if corresponding classes in the two hierarchies both contain a non-abstract method with identical signature. For such conflicts either composition is forbidden (*basic composition*) or an explicit (programmer defined) precedence relation $\ll$ is used to resolve these conflicts (*overriding composition*). Following the composition definition in [21] the resulting composed hierarchy is type correct.[3]

Dynamic interference denotes potential semantical changes at runtime. However, as the term dynamic is misleading, the term *binding interference* is preferred in this thesis. To reveal such cases, [21] defines a sufficient interference criterion stating when analyzed clients (or even any client) are affected by binding interference. If this is the case, a call-graph based impact analysis similar to [19, 20] can be used to identify affected program parts of clients.

3.3.1 The Snelting/Tip Criterion and Statically Bound Members

Static noninterference does *not* imply binding noninterference, as overriding method definitions can be added to the resulting composed hierarchy thus changing the semantics of the composed class hierarchy. Snelting and Tip prove that behavioral equivalence[4] is guaranteed, if for all potentially executed calls in the program the static lookup is preserved. Formally [21] defines the interference criterion as follows:

> A composition $\mathscr{H}_1 \oplus \mathscr{H}_2$ meets the noninterference criterion if for all $p \in \text{ObjectRefs}(\mathscr{H}_1)$, for all method calls $p.m(...) \in \mathscr{H}_1$, and for all $o \in \text{PointsTo}(\mathscr{H}_1 \oplus \mathscr{H}_2) \cap \text{Objects}(\mathscr{H}_1)$ we have that $\text{StaticLookup}(\mathscr{H}_1, T, m)$[5]$= \text{StaticLookup}(\mathscr{H}_1 \oplus \mathscr{H}_2, T', m)$ where $T = \text{TypeOf}(\mathscr{H}_1, o)$, and $T' = \text{TypeOf}(\mathscr{H}_1 \oplus \mathscr{H}_2, o)$.

In this definition *ObjectRefs* is the set of all object references in a given hierarchy, *Objects* is the set of creation sites and *PointsTo* formalizes the results of a pointer analysis associating all references from *ObjectRefs* with the subset of creation sites from *Object* these references may point to. *StaticLookup* finally formalizes the lookup mechanism for member access, based on the hierarchy, the member accessed and the type of the underlying object as denoted by *TypeOf*.

Their criterion however is only valid in the absence of *statically bound members and overloading*. As fields as well as static methods are bound statically in Java (and as well in AspectJ), their criterion is *not sufficient* in this context.

Example 3.3.1 *To show the failure of the Snelting/Tip criterion, consider Figure 3.3. Observe that execution of method* `X.f()` *will call* `X.zap()` *in the context of hierarchy* $\mathscr{H}_1$*, as* `b.x`

[2] For simplicity corresponding classes are identified by the same name, methods/fields by the same signature. More complex mechanisms are possible as well.

[3] i.e. member access is well-defined and for each assignment $a = x$ the type of a is a supertype of x.

[4] Behavioral equivalence of $\mathscr{H}$ and $\mathscr{H}'$ is defined as an identical executions sequence (sequence of executed statements) for the same client and inputs in both $\mathscr{H}$ and $\mathscr{H}'$.

[5] The function StaticLookup formalizes the name resolution process as found in most object-oriented languages and also in Java. Note that this also captures data member access.

```
1  class X {
2    void f() {
3      B b = new C();
4      if (b.x < 0) {
5        zip();
6      } else {
7        zap();
8      }
9    }
10   ...
11 }
```

```
12 class A {
13   int x = 1;
14 }
15
16 class B extends A {
17   int x = -1;
18 }
19
20 class C extends B {
21   int x = 1;
22 }
```

Figure 3.3: Example for a failure of the Snelting/Tip criterion: The figure shows an example hierarchy, the underlined code is only present in the edited version, i.e. hierarchy $\mathscr{H}_2$ is identical to $\mathscr{H}_1 \oplus \mathscr{H}_2$ and includes the underlined line.

resolves to `A.x` *which is initialized with value* 1. *However, this is different in the context of* $\mathscr{H}_1 \oplus \mathscr{H}_2$, *which adds field* `B.x` *(indicated by underlining the field definition in Figure 3.3). In the combined hierarchy the field access* `b.x` *resolves to the new field* `B.x` *with value* -1. *As a result,* `X.zip()` *is called in the combined hierarchy, and no longer* `X.zap()` *as in* $\mathscr{H}_1$. *So the semantics of clients calling* `X.f()` *are different in both systems (depending of definitions of* `X.zip()` *and* `X.zap()`, *of course). However, as there the semantical difference is only due to field access but not due to a method call, the original criterion is trivially fulfilled.*

If we extend the criterion to additionally check field access, the results are the following: Observe that ObjectRefs($\mathscr{H}_1$*) = {b}, PointsTo(*$\mathscr{H}_1$, `b`*)* $\cap$ *Objects(*$\mathscr{H}_1$*) = { c }, T = TypeOf(*$\mathscr{H}_1$, `c`*) = TypeOf(*$\mathscr{H}_1 \oplus \mathscr{H}_2$, `c`*) = T' =* `C` *and thus StaticLookup(*$\mathscr{H}_1$, T, `x`*) = StaticLookup(*$\mathscr{H}_1 \oplus \mathscr{H}_2$, T', `x`*) =* `C.x`, *i.e. the criterion is still fulfilled. Note that an analogous problem exists for static method calls.*

The problem here is that the criterion handles statically bound lookups equivalently to dynamically bound lookups. However, this can miss lookup changes as demonstrated in the example. The criterion only examines the static lookup of the *instantiated* `C` object. For this object however the lookup *is* preserved. The lookup change for the field access of `b` is not checked, as no `B` object is instantiated, thus the criterion states behavioral noninterference although the semantics of the system obviously changed by the hierarchy composition.

Apart from static members the criterion also fails if a language has *reflective capabilities*, like the `instanceof` construct in Java. The combination of inheritance relations of two hierarchies in general yields a different inheritance relation than either of the original hierarchy has, thus potentially affecting results of `instanceof` predicates. However, we will disregard reflective features in the analysis presented here.

3.3.2 Extending Snelting/Tip for Fields and Static Methods

Examining the proof of the interference criterion in [21] reveals the problems: the criterion only examines execution of *dynamically bound* (virtual) methods, but does not handle statically bound (i.e. non-virtual) methods. Formally [21] defines an execution sequence of a program (and related terms) as follows:

> 1. An *execution sequence* of a hierarchy $\mathscr{H}$ is the (finite or infinite) sequence of statements $E(\mathscr{H}, K, I, \sigma_0) = S_1, S_2, S_3, \ldots$ which results from executing the client code K of $\mathscr{H}$ with input I in initial state σ_0. [...]

2. The statement sequence S_1, S_2, S_3, ... consisting only of member accesses (data member access or method calls) is denoted $M(\mathcal{H}, K, I, \sigma_0) = S_{\eta_1}, S_{\eta_2}, S_{\eta_3}, \ldots$ where $S_{\eta_i} = S_j \in E(\mathcal{H}, K, I, \sigma_0)$. The corresponding sequence of invoked target methods [including data members] is denoted $T(\mathcal{H}, K, I, \sigma_0) = t_{\eta_1}, t_{\eta_3}, t_{\eta_3}, \ldots$ where each t_{η_i} is the method [or data member] invoked [or accessed] at runtime.

For clarity in the definition of $T(\mathcal{H}, K, I, \sigma_0)$ explicit references to data member access have been added, although they are already captured by the original definition.

Snelting and Tip prove in their work that identical call target sequences $T(\mathcal{H}, K, I, \sigma_0)$ and $T(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$ for all *method calls* in the system imply identical statement executions sequences. While the proof that identical target sequences imply identical execution sequences is correct, their interference criterion does *not* imply identical target sequences in the presence of statically bound members and is thus incomplete. To solve this problem, we extend their criterion as follows:

Definition 3.3.1 (Extended Noninterference Criterion) *A composition $\mathcal{H}_1 \oplus \mathcal{H}_2$ meets the noninterference criterion if for all $p \in ObjectRefs(\mathcal{H}_1)$, for all method calls p.m(...) in $\mathcal{H}_1$, and for all $o \in PointsTo(\mathcal{H}_1 \oplus \mathcal{H}_2) \cap Objects(\mathcal{H}_1)$ we have that StaticLookup($\mathcal{H}_1$, T, m) = StaticLookup($\mathcal{H}_1 \oplus \mathcal{H}_2$, T', m) where*

- *T = TypeOf($\mathcal{H}_1$, p), T' = TypeOf($\mathcal{H}_1 \oplus \mathcal{H}_2$, p), if m is bound statically, and*
- *T = TypeOf($\mathcal{H}_1$, o), T' = TypeOf($\mathcal{H}_1 \oplus \mathcal{H}_2$, o), iff m is bound dynamically.*

Note the extension to check the StaticLookup for the *static* type of the reference variable. This extension asserts that the lookup resolves to the same target for both static methods and data members. In the following we will refer to this extended criterion as the Snelting/Tip noninterference criterion.

In this thesis this noninterference criterion is used in the context of AspectJ's static crosscutting to analyze impact of non-static inter type declarations. Therefore the following section will show that inter type declarations can be seen as hierarchy compositions, and thus that the interference criterion is applicable in the context of AspectJ. We also propose an algorithm to calculate lookup changes due to static crosscutting.

3.4 Impact of Default Implementations

While inter type declarations give the programmer some additional modeling power, they are not without cost. We will first discuss potential side effects of interface inter type declarations. Usage of this feature can result in *incompatible inherited default implementations* potentially introducing flaws into a program. Usually the compiler checks if a class actually implements all methods of an implemented interface. For interfaces with default implementations this is different. The compiler no longer issues an error message if a class implements an interface but does not (re)define all default implementations, as now an implementation is available.

While the scenario seems to be related to standard inheritance of methods, it is not exactly the same as the programmer of the target class is not necessarily aware of the interface let alone the default implementations. Additionally classes in an inheritance hierarchy model semantically related entities, while for unrelated classes implementing the same interface this is not necessarily the case. As a consequence an interface default implementation may be less fitting for an implementing class compared to methods inherited from superclasses. Thus it is necessary to check if a default method has to be overridden—but this fact is not reported by the compiler any more. The aspect programmer is aware of default implementations used,

however he might be no expert for the target classes, and thus unable to decide if the default implementation is valid for a given class.

A simple analysis of interface inter type declarations can provide the necessary information and alert programmers of used default implementations. This allows aspect and base programmers to reconcile their implementations to avoid such cases. Given a class hierarchy and an aspect A, an analysis could be performed in three steps:

1. The set of interfaces for which aspect A provides default implementations has to be determined by scanning A's inter type declarations. Let $\mathscr{I}_{def}$ be the set of these interfaces. For $\mathtt{I} \in \mathscr{I}_{def}$ let *methods*(I) be the set of methods for which default implementations are provided.

2. The set of classes implementing an interface $\mathtt{I} \in \mathscr{I}_{def}$ has to be identified. Let $\mathscr{C}_{\mathscr{I}_{def}}$ be the set of these classes.

3. The set of classes $\mathscr{C}_{di}$ which do not provide an implementation of all interface methods (i.e. which use the default implementations) has to be determined. Let *methods*(C) be the set of all methods defined in Class C. Then

$$\mathscr{C}_{di} = \{\mathtt{C} \in \mathscr{C}_{\mathscr{I}_{def}} \mid \exists m \in \mathit{methods}(\mathtt{I}) : \mathtt{StaticLookup}(C,m) = \mathtt{I}\}$$

   Here $\mathtt{StaticLookup}(C,m) = \mathtt{I}$ indicates that the lookup resolves to the introduced default method in I.

   It is sufficient to check whether all methods in *methods*(I) are implemented as other missing methods are detected by the Java compiler. Note that any subclass of an affected class is influenced as well, unless it implements the necessary method and thus overrides the default implementation.

   If a class implementing an interface has been checked, for subclasses inheriting default implementations the situation is similar to traditional method inheritance. The assumption here is that a method which is meaningful in the context of the superclass will be meaningful in the context of the subclass.

Base and aspect programmer together can then examine affected classes to check whether the default implementation given by the interface is appropriate.

Example 3.4.1 *As an example consider aspect* M *given by the program below, which is applied to the running example shown in Figure 3.2. The aspect declares that classes* C *and* D *implement interface* I *and introduces a default implementation of method* y() *to the interface. The resulting hierarchy is shown in Figure 3.4 on page 44.*[6]

```
1 aspect M {
2   declare parents: C implements I;
3   declare parents: D implements I;
4
5   public void I.y() {
6     System.out.print("I.y()");
7   }
8 }
```

Note that classes C *and* E*—maybe unexpectedly—use the implementation given by* I.y(). *This fact is reported by the proposed analysis.*

[6] We use an non-standard extension of UML, as currently no standard notation to model aspects is available.

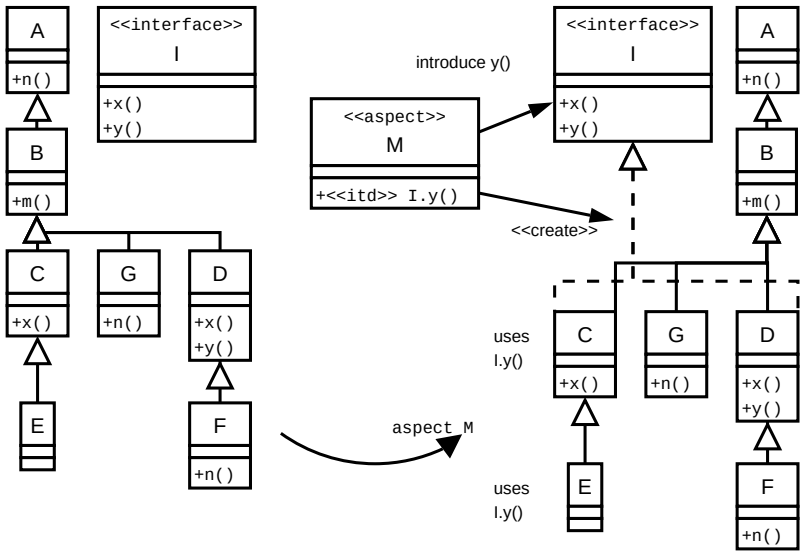


Figure 3.4: Using default implementations.

As a default implementation can only become available if no implementation is provided either by the implementing class itself or a superclass, no existing code can call this new method. Thus inter type declarations to interfaces cannot change the lookup for existing clients, neither statically nor dynamically. As we will see in the discussion below, interface inter type declarations thus cannot alter the lookup semantics of AspectJ programs. Note however that adding implemented interfaces widens the type of objects instantiated from the implementing class.

3.5 Impact of Class Inter Type Declaration

In contrast to interface inter type declaration, class inter type declarations are more complex as program semantics may change without modifying any class directly. This is called *binding interference* (derived from [21]) as outlined above. Resulting effects are described in detail in this section, but in short inter type declarations can change system semantics due to

- hiding of fields and static methods
- overloading existing methods with a more specific method and finally
- overriding methods originally inherited from a superclass.

Note that this analysis relies on the theoretical foundations of [21], which we will adapt for aspect-orientation.

3.5.1 Field Inter Type Declarations

Recall that fields are statically typed in Java. Unfortunately[7] fields defined in a superclass are hidden by a subclass field with the same name in Java. Field hiding can result in very unintuitive and thus hard to understand code, especially if inter type declarations are involved.

[7] "Unfortunately" as field hiding in general results in code which is hard to understand and we are not aware of a convincing application of this feature.

```
1  class C1 { int i = 0; }
2  class C2 extends C1 {
3    void f() {
4      C2 c2 = new C2();
5      c2.i = 5;
6      g(c2);
7    }
```

```
8    void g(C1 c1) {
9      System.out.print(c1.i);
10   }
11 }
12 aspect A {
13   int C2.i = 0;
14 }
```

Figure 3.5: Changed Field Lookup due to field hiding by an inter type declaration.

Example 3.5.1 *An example is shown in Figure 3.5. The field* `C2.i` *is added to* `C2` *by an inter type declaration in aspect* `A`*. Due to this addition the* `print` *command prints* 0 *and not* 5 *as one might expect—and as is also the case without the aspect. As* `C2.i` hides `C1.i` *and the object created at line 4 is statically typed as a* `C2` *object in* `f()` *(line 5) but as a* `C1`*-object in* `g()` *(line 9, right), the write access of* `i` *in* `f()` *actually accesses* `C2.i`*, in contrast to the non-aspectized program version where the lookup resolves to* `C1.i`*.*

Field hiding can produce subtle flaws in plain Java, and inter type declarations add additional complexity. Due to the non-locality of inter type declarations, the new field `C2.i` is not visible when examining the original class hierarchy. Currently there is no semantic support for programmers to detect that introduced fields hide an existing field and thus potentially change system semantics.

As inter type declarations have been introduced as a way to add additional state to classes in order to support aspects, it seems that name clashes in general might not be on purpose but rather accidental. An analysis should detect all such cases and warn the programmer of potentially changed program semantics.

Field hiding is comparable to method hiding for static methods in Java. As the problems and analysis of static method hiding are analogous to field hiding, this thesis will not discuss them explicitly; instead static methods are included in the discussion of (statically bound) field access. Hiding effects of static methods are detected by the same analysis algorithm.

3.5.2 Overriding Method Inter Type Declarations

Introducing members to classes can result in dynamic lookup changes if the introduced method is not static and redefines a method of a superclass.

Example 3.5.2 *Consider the example in Listing 3.1. This aspect introduces a method* `f()` *to class* `C2`*, with an identical signature as* `C1.f()`*. Any (virtual) call on a* `C2`*-object, e.g. from class* `X`*, now results in calling* `C2.f()` *and not in* `C1.f()` *as before. Consequently the semantics of a call to* `f()` *has possibly changed for any object of class* `C2` *(or any subclass thereof which does not define an own implementation of* `f()`*) without direct modification of these classes.*

If the introduced method `C2.f()` overrides `C1.f()` with respect to behavioral sub-typing [15], an (unknown) client of (a subclass of) `C2` may still work as expected. However, neither Java nor AspectJ guarantee this kind of method redefinition. The described problem is a special case of the *fragile base class problem* [17]—subclasses change behavior because of changes in the superclass (like i.e. adding a new, overriding method). Although tracking down bugs introduced by changing a base class is difficult, the problem is even worse with aspect languages as modifications of the base class are not visible if the code is viewed in isolation (i.e. without the applied aspect). To track bugs emerging from dynamic interference, impact analysis of aspect application should reveal method calls whose *dynamic lookup has changed*.

Listing 3.1: Inter Type Declarations and Overriding

```
1  class C1 {
2    void f() {}
3  }
4  aspect A {
5    void C2.f() {}
6  }
7  class C2 extends C1 {}
8
9  class X {
10   static void main(String[] args) {
11     new C2().f();
12   }
13 }
```

3.5.3 Overloading Method Inter Type Declarations

Another feature to take into account is overloading of methods. AspectJ allows to define/introduce methods with the same name as existing methods but different parameter lists, i.e. AspectJ allows overloading by method introduction. Unfortunately overloading also interferes with inheritance, and the rules by which the method lookup works in presence of overloading are rather complex. While field hiding and overriding are relatively easy to understand, overloading requires a more detailed consideration of the actual language semantics.

As AspectJ builds on Java, the Java overloading rules are recapitulated in the following. According to the Java language specification [10], Java first calculates the set of all *applicable and accessible methods*. The Java language specification defines 'accessible' as follows:

> Whether a method declaration is accessible (§6.6) at a method invocation depends on the access modifier (public, none, protected, or private) in the method declaration and on where the method invocation appears.

Accessible methods informally are all methods defined by the receiver or inherited from superclasses if access is possible under consideration of Java's access specifiers.

A method $T_D.m(F_1,\ldots,F_n)$ is defined to be *applicable* to a call $C.m(A_1,\ldots,A_n)$ if (quoted from [10], 15.12.2.1):

- The number of parameters in the method declaration equals the number of argument expressions in the method invocation.
- The type of each actual argument can be converted by method invocation conversion (§5.3) to the type of the corresponding parameter. Method invocation conversion is the same as assignment conversion (§5.2), except that constants of type int are never implicitly narrowed to byte, short, or char.

Informally method invocation conversion allows implicit upcasts, but no downcasts.

It is possible that more than one method implementation is both applicable and accessible. In this case the *maximally specific method* is chosen from the set of applicable methods. The Java language definition defines 'maximally specific' as follows:

> Let m be a name and suppose that there are two declarations of methods named m, each having n parameters. Suppose that one declaration appears within a class or interface T and that the types of the parameters are $T_1,\ldots,T_n$; suppose moreover

Listing 3.2: Inter Type Declarations and Overloading.

```
1  class C1 { }
2  class C2 extends C1 { }
3  aspect A {
4    void X.g(C2 c2) { }
5  }
6  public class X {
7    static void main(String[] args) {
8      g(new C2());
9    }
10   void g(C1 c1) { }
11 }
```

> that the other declaration appears within a class or interface U and that the types of the parameters are $U_1, \ldots, U_n$. Then the first member method is more specific than [the] other if and only if T_j can be converted to U_j by method invocation conversion, for all j from 1 to n.[8]
>
> A method is said to be *maximally specific* for a method invocation if it is applicable and accessible and there is no other applicable and accessible method that is more specific.

It is possible that no unique maximally specific method can be determined. In this case the call is *ambiguous* and considered a compilation error. Otherwise the unique maximally specific method is chosen. Note that resolution of overloading is a compile time issue.

The important observation to be made is that inter type declarations can change the set of applicable methods. They can even introduce a new most specific method and thus result in the execution of a different method as before so changing system semantics.

Example 3.5.3 *Consider the example in Listing 3.2. In this example aspect* `A` *introduces method* `g(C2)` *to class* `X`*. Now for the call* `g(new C2())` *two methods are applicable:* `X.g(C1)` *and* `X.g(C2)`*. As* `C2` *can be converted to* `C1` *but not vice versa, the introduced method is more specific and thus chosen. Analogous to hiding effects discussed before it can be argued that such semantical system modifications are likely due to unexpected name clashes and thus should be reported to the programmer.*

3.5.4 Noninterference Criterion for Inter Type Declarations

To detect semantical changes in the hierarchy, the noninterference criterion of Definition 3.3.1 is applied to aspects by defining a mapping from inter type declarations to hierarchy composition. As a result, the extended noninterference criterion can be applied to aspect introduction for inter type declarations as well.

Note that even with this extension this criterion is only a *necessary but not sufficient criterion*, as we do not consider the effects of *type changes*. However, recall the purpose of the analysis techniques presented here is to aid programmers in finding flaws in their programs. Thus completeness is—although desirable—not a necessary criterion for the techniques presented here. Even partial results can considerably help programmers using aspect-oriented techniques as these results are derived automatically and help to understand aspect effects.

[8] Note that this changes in an addendum to the Java Language specification. Before, the type of the caller was also relevant.

In contrast to Hyper/J, AspectJ is more restrictive in the possible class modifications using static crosscutting. Modification of system behavior is mainly achieved by using dynamic crosscutting techniques (advice). The approach presented here will reduce inter type declarations to hierarchy composition. Let therefore a hierarchy $\mathscr{H}$ be defined as in [21]:

Definition 3.5.1 (Class Hierarchy) *A class hierarchy $\mathscr{H}$ is a set of classes and an inheritance relation: $\mathscr{H} = (\mathscr{C}, \trianglelefteq)$. A class $\mathtt{C} \in \mathscr{H}$ has a name and contains a set of members. According to this definition, members$(\mathtt{C})$ does not contain inherited members that are declared in superclasses of $\mathtt{C}$. To indicate the members of class $\mathtt{C}$ defined in hierarchy $\mathscr{H}$ we write members$_{\mathscr{H}}(\mathtt{C})$; $\mathtt{C}_{\mathscr{H}}$ references definition of class $\mathtt{C}$ in hierarchy $\mathscr{H}$.*

Any AspectJ inter type declaration can be viewed as a hierarchy composition by defining a new hierarchy induced by an aspect $\mathtt{A}$.

Definition 3.5.2 (Aspect Induced Hierarchy) *Let $\mathscr{H} = (\mathscr{C}, \trianglelefteq)$ be a hierarchy, and $\mathtt{A}$ an aspect applied to it ($\mathtt{A} \notin \mathscr{C}$). Let $\mathscr{I}$ be the set of inter type declarations declared in this aspect. Elements of $\mathscr{I}$ have the form $(\mathtt{C},\mathtt{m})$. $\mathtt{C} \in \mathscr{C}$ indicates the class to which the new member $\mathtt{m}$ should be added. Then the aspect-induced hierarchy $\mathscr{H}' = (\mathscr{C}', \trianglelefteq')$ is constructively defined as follows:*

1. *$\forall \mathtt{C} \in \mathscr{H}$ create a new empty class named $\mathtt{C}$, add it to $\mathscr{C}'$*
2. *$\forall(\mathtt{C},\mathtt{m}) \in \mathscr{I}$ add member $\mathtt{m}$ to the corresponding class $\mathtt{C} \in \mathscr{C}'$ created in (1)*
3. *$(\trianglelefteq') = (\trianglelefteq)$ (same inheritance relations as in $\mathscr{H}$)*
4. *Add aspect $\mathtt{A}$ to $\mathscr{H}'$, but remove all inter type members.*

Informally, the resulting hierarchy contains no members from the base hierarchy but all introduced members and mirrors the original inheritance relations. Empty classes are possible.

As name clashes or *static interference* are considered an error by the AspectJ compiler *ajc*

$$\forall \mathtt{C} \in \mathscr{C}' : \forall \mathtt{m} \in \mathit{members}_{\mathscr{H}'}(\mathtt{C}) : \quad \mathtt{C} \in \mathscr{C} \wedge \mathtt{m} \notin \mathit{members}_{\mathscr{H}}(\mathtt{C})$$

always holds for syntactically correct AspectJ programs. AspectJ[9] does not allow overriding inter type declarations in the sense of Hyper/J, i.e. that the programmer can specify precedence rules for conflicting methods from different hierarchies. So only *basic compositions* where no name clashes occur have to be considered.

The hierarchy induced by an aspect need not be syntactically correct as methods introduced by the aspect might reference methods not present in $\mathscr{H}'$ but only in $\mathscr{H}$. All these dangling references are bound after combination of the resulting hierarchies if the original AspectJ program was correct.[10]

The hierarchy $\mathscr{H}'$ induced by the inter type declarations of an aspect $\mathtt{A}$ will now be composed with the hierarchy of the base system $\mathscr{H}$ by using a hierarchy composition operator $\oplus_s$. As described in [21] for arbitrary hierarchies, the inheritance relations of both hierarchies can be contradictory, e.g. if $(\mathtt{B},\mathtt{C}) \in \trianglelefteq$ and $(\mathtt{C},\mathtt{B}) \in \trianglelefteq'$. This however is impossible if a hierarchy induced by an aspect should be combined with the base hierarchy, as the resulting inheritance relation is always conflict free (here, even identical). Thus no collapsing of cycles is necessary and the general combination operator of [21] can simplified as follows.

[9] Referenced Version is 1.2

[10] For unqualified references to a Java member *m* of aspect `A` referenced by introduced methods, it is necessary to add the aspect instance as qualifier `A.aspectOf().`*m*, which accesses the relevant instance in $\mathscr{H}'$.

Definition 3.5.3 (Simplified Hierarchy Composition $\oplus_s$) *We define a simplified hierarchy composition operator $\oplus_s$ as follows. Let $\mathscr{H}_1 = (\mathscr{C}_1, \trianglelefteq_1)$ and $\mathscr{H}_2 = (\mathscr{C}_2, \trianglelefteq_2)$ be two class hierarchies with conflict free inheritance relations $\trianglelefteq_1$, $\trianglelefteq_2$ and no static interference. Then $\mathscr{H}_1 \oplus_s \mathscr{H}_2 = (\mathscr{C}, \trianglelefteq)$ is defined as follows*[11]*:*

1. $\mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2$ *(note that classes are identified by name)*
2. $(\trianglelefteq) = transRed(\trianglelefteq_1 \cup \trianglelefteq_2)$
3. $\forall \mathtt{C} \in \mathscr{C}_{\mathscr{H}} : members_{\mathscr{H}}(\mathtt{C}) = members_{\mathscr{H}_1}(\mathtt{C}) \cup members_{\mathscr{H}_2}(\mathtt{C})$,

where transRed is a function calculating the transitive reduction of the resulting relation.

Note that per definition the resulting combined hierarchy never contains multiple inheritance—as is possible if arbitrary hierarchies are combined—as for aspect-induced hierarchies $\trianglelefteq_1 = \trianglelefteq_2$.

Composing a hierarchy $\mathscr{H}$ and an aspect-induced hierarchy $\mathscr{H}'$ using operator $\oplus_s$ has the same effects as the inter type declarations in the corresponding aspect: Both operations simply add the introduced members to the respective classes of the resulting hierarchy.

Following the analysis of [21], it is now possible to apply the extended noninterference criterion to AspectJ introduction as well, which informally states that in order to maintain the semantics of existing clients of a base hierarchy $\mathscr{H}$, in the combined hierarchy $\mathscr{H} \oplus_s \mathscr{H}'$ all field accesses and method calls must resolve to the same member as in $\mathscr{H}$.

As the above definition of aspect induced hierarchies reduces inter type declarations to hierarchy composition, correctness of the interference criterion for inter type declarations is given by the proof presented in [21], if we show that the criterion implies an identical target sequence $T(\mathscr{H}, K, I, \sigma_0)$ and $T(\mathscr{H} \oplus_s \mathscr{H}', K, I, \sigma_0)$.

Proposition 3.5.1 *If the extended noninterference criterion holds, then*

$$T(\mathscr{H}, K, I, \sigma_0) = T(\mathscr{H} \oplus_s \mathscr{H}', K, I, \sigma_0).$$

Proof: The noninterference criterion demands that the static lookup for field accesses, static method calls and calls of virtual members remains unchanged by hierarchy composition. Each member access in Java is one of the three mentioned above. As the hierarchy composition operator $\oplus_s$ does not change or remove any code, any change in the target sequence can only be due to changed lookups and thus violates the criterion. □

3.6 Hierarchy Modifications

Besides introduction, AspectJ allows to modify the structure of inheritance hierarchies, with the intention to move classes (together with all their subclasses) 'down' the inheritance hierarchy, so that the resulting program is still type correct (i.e. each member access is resolvable and for each assignment $a = x$, the type of a is a supertype of x)[12].

3.6.1 Impact of Changing the Inheritance Hierarchy

At first glance any client using classes with a modified inheritance hierarchy should still work as any member access is still resolvable, i.e. type correctness is maintained. However, similarly to class inter type declarations, there are two problems. The aspect in Listing 3.3 results in the hierarchy modifications shown in Figure 3.6. Let `d` be an instance of class `D`.

[11] In the following $\oplus$ will refer to $\oplus_s$.

[12] It is not possible to move classes 'up' in the inheritance hierarchy (AspectJ accepts this declaration without effect).

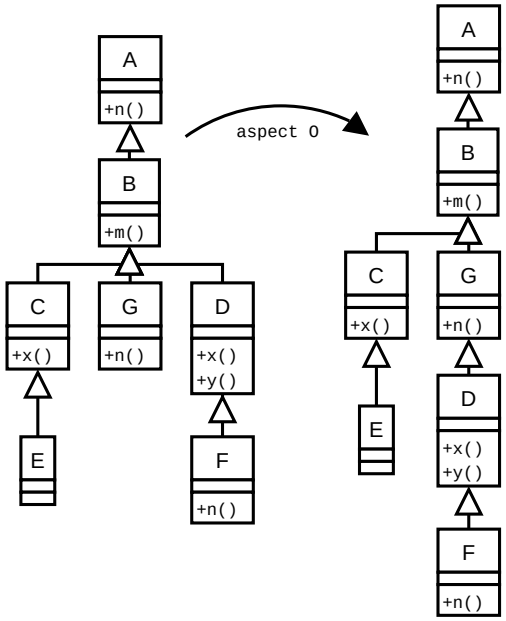


Figure 3.6: Effects of hierarchy modification due to application of aspect `O`. Moving classes `D` (and also `F`) changes the lookup for `D.n()`. Lookup for other methods remains unchanged, however the type of `D` and `F` also changed.

Type Changes: Class `D` is moved down the inheritance hierarchy by aspect `O`. Any predicate `d instanceof G` now changed value—from *false* to *true*. More generally, the *type* of class `D` has changed. This allows additional up-casts (`(G)d`), which resulted in a `ClassCastException` before. These exceptions might have been caught and so control flow might have changed.

Binding Interference: A change in the structure of an inheritance hierarchy possibly changes the method actually executed by a virtual call due to binding interference. Figure 3.6 gives an example of this situation with method call `n()` on a `D` object: Without aspect application, `A.n()` is called; with `O` applied, the virtual call resolves to `G.n()`.

Listing 3.3: Changing the Inheritance Hierarchy

```
1 aspect O {
2   declare parents: D extends G;
3 }
```

Example 3.6.1 *The impact of changes in the inheritance relation is demonstrated in Figure 3.6. The changes presented in this example are due to application of the simple aspect `O` to the running example shown in Figure 3.2 on page 39. Note that changing the inheritance hierarchy results in a changed lookup for calls to method `n()` for `D`-objects. Before, `A.n()` has been called, with applied aspect the call resolves to `G.n()`.*

3.6.2 Hierarchy Modification as Hierarchy Composition

We can again reduce modification of the inheritance hierarchy to a hierarchy combination. In this case, the hierarchy induced by `declare parents ...extends` statements contains an empty class for any class in the base hierarchy and an inheritance relation $\trianglelefteq'$ modified by the aspect statement as follows.

Definition 3.6.1 (Induced Hierarchy) *Let $\mathscr{H} = (\mathscr{C}, \trianglelefteq)$ be a hierarchy an aspect* `A` *is applied to. For each statement* `declare parents: A extends B` $\Rightarrow (A,B) \in D$. *Then* $\trianglelefteq' = (\trianglelefteq \cup D)$. *The hierarchy defined by* `A` *is* $\mathscr{H}' = (\mathscr{C}', \trianglelefteq')$, *where* $\mathscr{C}' = \mathscr{C}$, $\forall \mathtt{C} \in \mathscr{C}' : members(\mathtt{C}) = \emptyset$.

As hierarchy modifications in AspectJ are restricted—it is only allowed to declare that a class now is a subclass of a sibling[13] in the inheritance hierarchy (or a sibling's subclass)—the following always holds:

- $(\trianglelefteq) \subseteq (\trianglelefteq')$
- $(\mathtt{D},\mathtt{C}) \in (\trianglelefteq') \Rightarrow (\mathtt{C},\mathtt{D}) \notin (\trianglelefteq')$ (no conflicts in $\trianglelefteq'$)

With these properties, the simplified hierarchy combination operator $\oplus_s$ can again be applied as no special handling of conflicts is necessary. The resulting hierarchy is given by $\mathscr{H}' = (\mathscr{C}', transRed(\trianglelefteq'))$.

3.6.3 Induced Hierarchy for Static Crosscutting

For completeness reasons we finally define the induced hierarchy for combining a given hierarchy $\mathscr{H}$ with a set of aspects $\mathscr{A}$.

Definition 3.6.2 (Aspect Induced Hierarchy) *Let $\mathscr{H} = (\mathscr{C}, \trianglelefteq)$ be a hierarchy, and $\mathscr{A}$ an set of aspects applied to it ($\mathscr{A} \cap \mathscr{C} = \emptyset$). Let $\mathscr{I}_{\mathtt{A}}$ be the set of inter type declarations declared by an aspect $\mathtt{A} \in \mathscr{A}$. Then $\mathscr{I} = \bigsqcup_{\mathtt{A} \in \mathscr{A}} \mathscr{I}_{\mathtt{A}}$ is the set of all inter type declarations. Elements of $\mathscr{I}$ have the form $(\mathtt{C},\mathtt{m})$. $\mathtt{C} \in \mathscr{C}$ indicates the class where the new member* `m` *should be added. For each statement* `declare parents: C2 extends C1` $\in \mathtt{A} \Rightarrow (\mathtt{C2},\mathtt{C1}) \in D_{\mathtt{A}}$. *Then $\mathscr{D} = \bigsqcup_{\mathtt{A} \in \mathscr{A}} D_{\mathtt{A}}$ is the set of all hierarchy modifications.*

Then the aspect-induced hierarchy $\mathscr{H}' = (\mathscr{C}', \trianglelefteq')$ is constructively defined as follows:

1. $\forall \mathtt{C} \in \mathscr{H}$ *create a new empty class named* `C`, *add it to $\mathscr{C}'$*
2. $\forall (\mathtt{C},\mathtt{m}) \in \mathscr{I}$ *add member* `m` *to the corresponding class $\mathtt{C} \in \mathscr{C}'$ created in (1)*
3. $\trianglelefteq' = transRed(\trianglelefteq \cup \mathscr{D})$
4. $\mathscr{C}' = \mathscr{C} \cup \mathscr{A}$. *Add all aspects $\mathtt{A} \in \mathscr{A}$ to $\mathscr{H}'$, but remove all inter type members from them.*

Note that the above definitions uses symbol $\bigsqcup$ instead of standard set union, as we want to explicitly express the handling of conflicting static directives.

Definition 3.6.3 (Handling of Conflicts) *Let A_1 and A_2 be two aspects and $\mathscr{I}_{A_1}$ and $\mathscr{I}_{A_2}$ the sets of inter type definitions derived from them. Then*

$$\mathscr{I}_{A_1} \sqcup \mathscr{I}_{A_2} = \begin{cases} \mathscr{I}_{A_1} \cup \mathscr{I}_{A_2}, & \text{if } \mathscr{I}_{A_1} \cap \mathscr{I}_{A_2} = \emptyset, \\ (\mathscr{I}_{A_i} - \mathscr{I}_{A_j}) \cup \mathscr{I}_{A_j}, & \text{if } A_j \text{ has higher precedence than } A_i, \\ \bot & \text{else} \end{cases}$$

For the resulting hierarchy modifications D_{A_1} and D_{A_2}, $D_{A_1} \sqcup D_{A_2}$ is defined analogous.

[13] If u, v are siblings $\Rightarrow (u,v) \notin (\trianglelefteq^*) \wedge (v,u) \notin (\trianglelefteq^*) \wedge \exists w \in \mathscr{C} : (u,w) \in (\trianglelefteq^*) \wedge (v,w) \in (\trianglelefteq^*)$, $(\trianglelefteq^*)$ indicates the transitive closure of $(\trianglelefteq)$.

The above definition explicitly captures AspectJ's overriding semantics in case of identical inter type declarations or hierarchy modifications in two aspects, as long as aspect precedence is declared. In case of undeclared precedence, the AspectJ compiler reports a compiler error, indicated by returning $\perp$.

3.7 Impact of Type Changes

To prove that the semantics of a client of a given hierarchy is preserved by static crosscutting, the interference criterion of [21] is a necessary but *not sufficient* condition. If a language contains statements for run time type identification (*RTTI*), control flow might change although the above noninterference criterion is met. Java contains such statements with the predicate `instanceof`, which allows to make control flow dependent of the type of an object.

To guarantee that behavior of a client is preserved, all `instanceof` statements have to evaluate to the same value independently of aspect application. To calculate the value of such expressions at compile time, the runtime type of each reference involved in an `instanceof` predicate has to be known. Approximations with points-to analysis are possible, but precise points-to analysis is undecidable. Thus in general only a *superset* of the objects (and their types!) a reference may point to can be calculated.

Preservation of behavior can only be guaranteed iff points-to sets of references involved in an `instanceof`-statement before and after the hierarchy modification evaluate to the *same single type*—a very rigid requirement. In general, when using static analysis, many predicates will evaluate to type-sets with a cardinality greater than one. In this case, conservative approximation requires to assume that the behavior of the client has changed. This however will presumably result in many false positives, reducing the value of the whole (expensive!) analysis.

To check the impact of changes to any client of the modified hierarchy the noninterference criterion can be applied if RTTI is excluded. Finding those method calls with changed lookup is relatively easy: Only calls to methods (re)defined in a class between (and including) the new and the former superclass can be influenced, if those methods are not redefined by the affected class itself. We will describe an algorithm explaining detection of changed lookups in detail in Section 3.8.

3.8 Finding Changed Lookups

In this section we present a set of algorithms which analyze the inheritance relation and derive all changed lookups due to

- field/method hiding,
- overloading,
- overriding and
- hierarchy modifications

resulting from aspect application.

As static crosscutting, as implemented with AspectJ's inter type declarations, "only" modifies the static system structure, its effects are mostly (neglecting type changes) amenable to static analysis. This changes once dynamic crosscutting (i.e. advice) is considered.

3.8.1 Deriving Lookup Changes

We finally explain how lookup changes can be derived to check if our noninterference criterion is met. We will first discuss how the different cases mentioned above are handled in detail, before we finally introduce a combined analysis to capture all lookup changes.

Lookup Changes due to Hiding

Let (T,m) be an inter type declaration of a statically bound member (i.e. either a field or a static method), where m identifies the member and T the target class. Let $subs(\mathscr{H},T)$ be a list of direct subclasses of T in hierarchy $\mathscr{H}$.

Definition 3.8.1 (Lookup Changes) *A lookup change is given by a triple (m,C,T), where m indicates the referenced member, C is the callee, i.e. the static (for statically bound) or dynamic (for dynamically bound) type used to reference the member, and $T.m$ is the actually accessed member.*

We then calculate lookup changes by calling `lookupChanges`($\mathscr{H}$, $\mathscr{H} \oplus_s \mathscr{H}'$, (T,m), [T], []), where function `lookupChanges` is defined as follows:

```
lookupChanges(H, H ⊕s H', (T,m), [], lc) = lc
lookupChanges(H, H ⊕s H', (T,m), C:Cs, lc) =
  case StaticLookup(H,T,m) = ⊥ then lc ; new member
  otherwise
    (let case StaticLookup(H ⊕s H',C,m) = T then
           ; lookup changed to introduced member
           Cs' = Cs@subs(H ⊕s H',C) ∧ lc' = lc@[(m,C,T)]
         otherwise
           Cs' = Cs ∧ lc' = lc ; lookup unchanged
     in lookupChanges(H, H ⊕s H', (T,m), Cs', lc'))
```

Intuitively, function `lookupChanges` traverses the combined class hierarchy $\mathscr{H} \oplus_s \mathscr{H}'$ in a top-down manner starting at T, and in each class checks if `StaticLookup`$(\mathscr{H} \oplus_s \mathscr{H}',C,m) = T$. If this is the case, a respective lookup change is created, and analysis proceeds with the subclasses of T. Otherwise analysis terminates for this branch of the inheritance tree.

Note that function `lookupChanges` calculates *all* possible lookup changes. However, if noninterference should be checked for a certain client $\mathscr{K}$, then we can match the resulting set of lookup changes with member accesses actually present in the code. As we are currently analyzing *statically* bound members, the necessary information—i.e. the static type for each statically bound member access—is directly available from a typed abstract syntax tree.

Lookup Changes due to Overriding

Note that function `lookupChanges` also captures all potential lookup changes for dynamic methods. However, calculating the relevant lookup changes with respect to a particular client $\mathscr{K}$ is considerably harder than for static members, as the dynamic type for a method call is statically not known and has to be approximated for example by using pointer analysis. We give an example demonstrating lookup calculation for both statically and dynamically bound members.

Example 3.8.1 (Hiding and Overriding) *Consider Example 3.2 on page 39 where we will apply aspect P (shown below) and then use the above algorithm to calculate relevant lookup changes.*

```
aspect P {
  int D.i;
  void B.n(A a) {}
}
```

From the above aspect, we derive two inter type declarations, (D, i) and (B, n(A)). We thus calculate `lookupChanges(`$\mathscr{H}$`, `$\mathscr{H} \oplus_s \mathscr{H}'$`, (D, i), [D], [])` *and* `lookupChanges(`$\mathscr{H}$`, `$\mathscr{H} \oplus_s \mathscr{H}'$`, (B, n(A)), [B], [])`*, which yields the following calculations:*

```
lookupChanges(H, H ⊕s H', (D, i), [D], []) =
lookupChanges(H, H ⊕s H', (D, i), [F], [(i, D, D)]) =
lookupChanges(H, H ⊕s H', (D, i), [], [(i, D, D), (i, F, D)]) =
   [(i, D, D), (i, F, D)]

lookupChanges(H, H ⊕s H', (B, n(A)), [B], []) =
lookupChanges(H, H ⊕s H', (B, n(A)), [C, D, G], [(n(A), B, B)]) =
lookupChanges(H, H ⊕s H', (B, n(A)), [D, G, E],
   [(n(A), B, B), (n(A), C, B)]) =
lookupChanges(H, H ⊕s H', (B, n(A)), [G, E, F],
   [(n(A), B, B), (n(A), C, B), (n(A), D, B)]) =
lookupChanges(H, H ⊕s H', (B, n(A)), [E, F],
   [(n(A), B, B), (n(A), C, B), (n(A), D, B)]) =
lookupChanges(H, H ⊕s H', (B, n(A)), [F],
   [(n(A), B, B), (n(A), C, B), (n(A), D, B), (n(A), E, B)]) =
lookupChanges(H, H ⊕s H', (B, n(A)), [],
   [(n(A), B, B), (n(A), C, B), (n(A), D, B), (n(A), E, B)]) =
  [(n(A), B, B), (n(A), C, B), (n(A), D, B), (n(A), E, B)]
```

Lookup Changes due to Overloading

To calculate if a newly introduced method overrides an existing method, function `lookupChanges` is not sufficient, as we have to compute if the new method is more specific than an existing method with the same name.

Therefore, we assume availability of a function `applicable(`$\mathscr{H}$`,`T`,`m`)` which returns the set of all applicable and accessible methods for a call to m on a T object (or for class T in case of static methods) in hierarchy $\mathscr{H}$ where the static type of all parameters matches the formal parameter types of m. Let $paras(m)$ denote the list of parameters for a method m. The set of methods overloaded by an inter type declaration (T,m) is then calculated by calling `getOverloaded(`$\mathscr{H} \oplus_s \mathscr{H}'$`, `$(T,m)$`, [`$T$`], [])`, which is defined below.

```
; generate a lookup change if new method is most specific
lessSpecific(H ⊕s H',C,m,[]) = [(m,C,StaticLookup(H ⊕s H',C,m)]
lessSpecific(H ⊕s H',C,m,a:as) =
  let paras(m) = (M1,...Mn) and paras(a) = (A1,...Ak) in
  case n = k and ∀i : Ai ⊵ Mi and ∃j : Aj ▷ Mj then
    lessSpecific(H ⊕s H',C,m,as) ; current a is less specific
  otherwise [] ; more specific method

getOverloaded(H ⊕s H', (T,m), [], lc) = lc
getOverloaded(H ⊕s H', (T,m), C:Cs, lc) =
  case StaticLookup(H,C,m) = ⊥ then [] ; new member
  otherwise
    let as = applicable(H ⊕s H', C, m)
    and l = lessSpecific(H ⊕s H', C, m, as) in
    getOverloaded(H ⊕s H', (T,m), Cs@subs(H ⊕s H',C), lc@l)
```

Note that, as function `lessSpecific` is only called with applicable methods (i.e. $a \in$ `applicable`$(\mathscr{H} \oplus_s \mathscr{H}', T, m)$) for each parameter A_i in function `lessSpecific` $A_i \trianglerighteq M_i$ always holds. Otherwise we immediately get an ambiguous lookup resulting in a non-compilable system.[14] Stated in other words, function `getOverloaded` determines all methods applicable to a call with the signature of the newly introduced method (for new members there are none and thus no lookup changes) and uses function `lessSpecific` to check if the new method is indeed most specific. In that case a lookup change is generated. Function `getOverloaded` proceeds with all subclasses, which can be affected by the new method as well.

Note that this does not directly describe all lookup changes, as a specific client can include even more specific method calls. The set of lookup changes we calculate however gives programmers an estimate to asses such changes. A call in C is affected by an inter type declaration (T,m) if m is applicable to it and a lookup change (m,C,T) exists.

Example 3.8.2 (Overloading) *To demonstrate the algorithm to calculate lookup changes due to overloading, we apply aspect* `Q` *shown below to Example 3.2 on page 39.*

```
aspect Q {
  void B.n(B);
}
```

As method `n(B)` *is more specific than* `n(A)` *(*`B` *is a subclass of* `A`*), we expect lookup changes. Applying function* `getOverloaded` *yields the following calculation:*

```
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [B], []) =
  (lessSpecific(ℋ ⊕s ℋ', B, n(B), [n(A)]) = [(n(B), B, B)])
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [C,D,G], [(n(B), B, B)]) =
  (lessSpecific(ℋ ⊕s ℋ', C, n(B), [n(A)]) = [(n(B), C, B)])
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [D,G,E],
  [(n(B), B, B), (n(B), C, B)]) =
  (lessSpecific(ℋ ⊕s ℋ', D, n(B), [n(A)]) = [(n(B), D, B)])
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [G,E,F], [(n(B), B, B),
  (n(B), C, B), (n(B), D, B)]) =
  (lessSpecific(ℋ ⊕s ℋ', G, n(B), [n(A)]) = [(n(B), G, B)])
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [E,F], [(n(B), B, B),
  (n(B), C, B), (n(B), D, B), (n(B), G, B)]) =
  (lessSpecific(ℋ ⊕s ℋ', E, n(B), [n(A)], [F] = [(n(B), E, B)])
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [F], [(n(B), B, B),
  (n(B), C, B), (n(B), D, B), (n(B), G, B), (n(B), E, B)]) =
  (lessSpecific(ℋ ⊕s ℋ', F, n(B), [n(A)]) = [(n(B), F, B)])
getOverloaded(ℋ ⊕s ℋ', (B, n(B)), [F], [(n(B), B, B), (n(B), C, B),
  (n(B), D, B), (n(B), G, B), (n(B), E, B), (n(B), F, B)]) =
[ (n(B), B, B), (n(B), C, B), (n(B), D, B),
  (n(B), G, B), (n(B), E, B), (n(B), F, B) ]
```

Lookup Changes due to Hierarchy Modifications

We finally describe how lookup changes due to hierarchy modifications are captured. To capture all methods where lookup can potentially change, we first define the set of known methods.

Definition 3.8.2 (Known Members) *The set of known members for a class C is defined as*

$$\texttt{knownMembers}(\mathscr{H}, C) = \{sig(m) : \exists C' \trianglerighteq C, m \in C'\},$$

[14] Note that we only analyze compilable systems; otherwise analysis is often hard or even not possible in general.

where $\texttt{sig}(m)$ *indicates the signature of a member m, i.e. for methods the method signature and for fields field name and type.*

We then determine the members which have to be checked.

Definition 3.8.3 (Lookup Change Candidates) *The set of members with potentially changed lookups* `toCheck` *is defined as*

$$\texttt{toCheck}(\mathscr{H},C) = \{sig(m) \in \texttt{knownMembers}(\mathscr{H},C) : m \notin C\}$$

In other words each known member where C does not define its own version is potentially affected by lookup changes. Let (C,S,T) be a hierarchy change defined in aspect A indicating that the superclass of class C changes from S to T in the following. To derive changed lookups, we have to consider two effects:

Changes to C: Class C itself is affected by the following set of lookup changes:

$$\{(m,C,\texttt{StaticLookup}(\mathscr{H} \oplus_s \mathscr{H}',C,m)) | sig(m) \in \texttt{toCheck}(C) : \\ \texttt{StaticLookup}(\mathscr{H} \oplus_s \mathscr{H}',C,m) = T' : S \rhd T' \trianglerighteq T\}.$$

Changes to subclasses C' ($C' \lhd C$): For each subclass $C' \lhd C$ we also have to check whether C' is affected. However, here it is sufficient to check if lookup changes of the direct superclass are also affecting the subclass (as the subclass might override or hide one of the affected members).

Let $super(\mathscr{H},C)$ denote the supertype of class C in hierarchy $\mathscr{H}$. We then calculate lookup changes by using the following functions:

---

knownMembers($\mathscr{H}$, C) =
  case $super(\mathscr{H},C) \neq \bot$ **then** { $sig(m) : m \in C$ }
  otherwise { $sig(m) : m \in C$ } $\cup$ knownMembers($\mathscr{H}$, $super(\mathscr{H},C)$)

toCheck($\mathscr{H}$, C, $\emptyset$, cs) = cs
toCheck($\mathscr{H}$, C, ms, cs) =
  let m $\in$ ms **and** ms' = ms - {m} **in**
  case $\exists n \in C : sig(n) = sig(m)$ **then** toCheck($\mathscr{H}$, C, ms', cs)
  otherwise toCheck($\mathscr{H}$, C, ms', cs $\cup$ m)

The two above definitions describe how `knownMembers` and `toCheck` can be calculated by a bottom-up traversal of the inheritance hierarchy and successive filtering.

We first define a function to calculate the impact on a specific class C' affected by the hierarchy modification by calling `getHierarchyChangesTarget(`$\mathscr{H} \oplus_s \mathscr{H}'$`, `$(C,S,T)$`, `$C'$`, `$\texttt{toCheck}_{C'}$`, [])`, where $C \trianglerighteq C'$ and $\texttt{toCheck}_{C'}$ is the set of method to be checked and function `getHierarchyChangesTarget` is defined below.

---

getHierarchyChangesTarget($\mathscr{H} \oplus_s \mathscr{H}'$, (C,S,T), C', $\emptyset$, lc) = lc
getHierarchyChangesTarget($\mathscr{H} \oplus_s \mathscr{H}'$, (C,S,T), C', ms, lc) =
  let m $\in$ ms **and** ms' = ms - { m } **in**
  case StaticLookup($\mathscr{H} \oplus_s \mathscr{H}',C',m) = T' : S \rhd T' \trianglerighteq T$ **then**
    getHierarchyChangesTarget($\mathscr{H} \oplus_s \mathscr{H}'$, (C,S,T), C', ms', m:lc)
  otherwise
    getHierarchyChangesTarget($\mathscr{H} \oplus_s \mathscr{H}'$, (C,S,T), C', ms', lc)

Intuitively function `getHierarchyChangesTarget` checks if the lookup for a method we have to check evaluates to a new superclass. If this is the case a respective lookup change is generated.

With this information, we can finally calculate all lookup changes due to the hierarchy modification (C,S,T) by calling `hierarchyChanges(`$\mathscr{H} \oplus_s \mathscr{H}'$`, (`$C,S,T$`), [(`$C$`, HTC)], [])`, where HTC = `toCheck(`$\mathscr{H}$`, `C`, knownMembers(`$\mathscr{H}$`, `C`), `$\emptyset$`)` and function `hierarchyChanges` is defined as follows.[15]

```
hierarchyChanges(ℋ ⊕s ℋ', (C,S,T), [], lc) = lc
hierarchyChanges(ℋ ⊕s ℋ', (C,S,T), (C', ms):ns, lc) = lc
  case ms = [] then hierarchyChanges(ℋ ⊕s ℋ', (C,S,T), ns, lc)
  otherwise
    let ms'= ns@pair(subs(ℋ ⊕s ℋ',C'), lc')
    and lc'= getHierarchyChangesTarget(ℋ ⊕s ℋ', (C,S,T), C', ms', lc)
    in hierarchyChanges(ℋ ⊕s ℋ', (C,S,T), ns, changes(C', lc')@lc
```

Intuitively function `hierarchyChanges` traverses the class hierarchy in a top-down manner starting at the class directly affected by the hierarchy modification and checks if the lookup for relevant methods evaluates to a new superclass in the combined hierarchy by using function `getHierarchyChangesTarget`.

Example 3.8.3 (Hierarchy Changes) *We demonstrate the calculation for the running example (Figure 3.2 on page 39 and aspect O (Figure 3.6 on page 50)).*

```
knownMembers(ℋ, D) =
  {x, y} ∪ knownMembers(ℋ, B) =
  {x, y} ∪ {m} ∪ knownMembers(ℋ, A) =
  {x, y, m} ∪ {n} = {x, y, m, n}

toCheck(ℋ, D, {x, y, m, n}, ∅) =
  {m, n} ; D itself defines x and y

hierarchyChanges(ℋ ⊕s ℋ', (D,B,G), [(D, {m, n})], []) =
  (lc'= getHierarchyChangesTarget(ℋ ⊕s ℋ', (D,B,G), D, {m, n}, []) =
          ( StaticLookup(ℋ ⊕s ℋ', D, m) = B )
       getHierarchyChangesTarget(ℋ ⊕s ℋ', (D,B,G), D, {n}, []) =
          ( StaticLookup(ℋ ⊕s ℋ', D, n) = G; B \rhd G ⊵ G )
       getHierarchyChangesTarget(ℋ ⊕s ℋ', (D,B,G), D, ∅, [n]) = [n])
hierarchyChanges(ℋ ⊕s ℋ', (D,B,G), [(F,{n})], [(n, D, G)]) =
  (lc'= getHierarchyChangesTarget(ℋ ⊕s ℋ', (D,B,G), F, {n}, []) =
          ( StaticLookup(ℋ ⊕s ℋ', F, n) = F )
       getHierarchyChangesTarget(ℋ ⊕s ℋ', (D,B,G), F, ∅, []) = [])
hierarchyChanges(ℋ ⊕s ℋ', (D,B,G), [], [(n, D, G)]) = [(n, D, G)]
```

Combined Algorithm

While we described how lookup changes due to hiding, overriding, overloading and hierarchy modifications can all be captured in isolation, capturing all changes from the combined effects resulting from an aspect is not completely trivial. However, as we carefully designed the above algorithms, we can simply collect all the above lookup changes to calculate all relevant lookups:

[15] The definition uses a simple helper functions `pair` and `changes` defined as `pair([], x) = []; pair(l:ls, x) = [(l, x)]:pair(ls, x)` and `changes(C, []) = []; changes(C, m:ms) = (m, C, StaticLookup(`$\mathscr{H} \oplus_s \mathscr{H}'$`, C, m)):changes(C, ms)`

- First, function `lookupChanges` collects *all* lookup changes due to hidden or overridden members which now resolve to the newly introduced member in the context of the *combined hierarchy*. This especially includes lookup changes to the newly introduced member of moved subclasses. Thus side effects of hierarchy modifications are captured as well.

- Second, function `getOverloaded` captures all lookup changes due to an inter type declaration (T,m). However, hierarchy modifications can now also result in overloaded calls as potentially more specific methods in the newly added superclasses are now available.

  We capture these cases as follows. For a hierarchy modification (C,S,T), we collect all methods signatures for methods defined in classes T' with $S \rhd T' \trianglerighteq T$. Let M be this set of method signatures. From this set of methods we then remove all method signatures known in S and also all method signatures of methods defined in C, i.e. the relevant set of methods R is defined as $R = M - \texttt{knownMembers}(\mathscr{H},S) - \{sig(m) : m \in C\}$.

  We now interpret each method $m \in R$ as a (virtual) inter type declaration and calculate relevant lookup changes by calling `getOverloaded(`$\mathscr{H} \oplus_s \mathscr{H}'$`, (`$T'$`, `$m$`), [`$C$`], [])`, so capturing all lookup changes due to overloading resulting from hierarchy modifications.

- Finally function `hierarchyChanges` collects all changes resolving to members in additional superclasses in the *combined hierarchy*.

Section 3.9 on page 59 gives an example of this combined analysis algorithm.

3.8.2 Client Specific versus General Analysis

The described analysis algorithms derive *all* lookup changes, for any type defined in the hierarchy. However, the noninterference criterion is formulated based on *lookup changes for individual call sites*. If we demand that no lookup changes at all may occur, the noninterference criterion is trivially fulfilled. This approach however has the disadvantage that we might report interference although for a specific client of interest the lookup change is not relevant. This is the case if the affected call is never executed, e.g. because the class affected by a changed lookup is never instantiated. However this disadvantage is less important in the context of AspectJ, as inter type declarations are not primarily a mechanism to *modify* system semantics—this is the purpose of advice—but rather to *augment* existing classes. In this context, changed lookups in general are likely to be unwanted side effects.

If an increased precision with respect to a specific client is needed, the set of derived lookups can be filtered by directly matching static member access with derived lookup changes and using the results of a pointer analysis, as Snelting and Tip suggested, for dynamic lookup changes. The latter would correspond to a straightforward implementation of the Snelting/Tip criterion.

Note that a pointer analysis however is not the only way to increase precision with respect to a specific client. Using type analysis techniques like CHA [9], RTA [4] or XTA [25] a whole range of analysis techniques exists which all increase precision by reducing possible targets for (virtual) call sites.

All client specific analysis techniques however produce results which are only valid for the specific client which has been analyzed, but not in general. To avoid semantical differences for *any* client, including yet unknown clients, no lookup changes *at all* may occur. First experimental results on the limited available AspectJ code have shown that lookup changes due to inter type declarations scarcely occur (see case studies Section 4.5.7, derived lookup changes). This supports the assumption that such lookup changes only occur unexpectedly,

although it is too early to draw general conclusion, as we can only report experience from a very limited available AspectJ code base.

3.9 An Example Analysis

To see how the proposed algorithms work, our analysis is applied to an example using all static modification features of AspectJ. Therefore we expand the running example presented in Figure 3.2 (page 39) with a client and apply aspect `Complex` shown in Listing 3.4 to this system. This aspect combines several of the aspects which have been discussed in the previous sections, i.e. we have to deal with a combined analysis of inter type member declarations, default interface methods and hierarchy modifications.

3.9.1 The System to Analyze

As a starting point, the class hierarchy defined by program shown in Figure 3.2 (page 39) is given, together with aspect `Complex`, which combines the effects of former aspects. It introduces new methods `n(A)` and `n(B)` to class `B`, adds a field `int i` in class `D`, changes the inheritance relation (`declare parents:D extends G`) and declares that classes `C` and `D` implement interface `I`. Finally method `y()` is introduced to interface `I`. Additionally, the aspect defines an own `main`-method which is necessary to test the results of interface declaration. Effects of aspect application are a changed inheritance relation as well as changed lookups for some methods due to addition of new methods.

The classes of this example are quite simple: All methods only print their name and the class they are defined in, but this setting is already sufficient to show how the aspect affects the existing system. Figure 3.7 presents the output of the system. The figure contains three sections. The output of the original system without application of the aspect is shown in sub-figure (a). The effects of binding interference are visible in sub-figure (b), which shows the output of the original `main(..)`-method with aspect `Complex` applied to the system. The dispatch has changed for calls to `n(A)` for classes `B`, `C`, `E`, and `D`. The first three classes are directly affected by the introduction of `n(A)` to `B`, class `D` by the change of the hierarchy. The introduced overloading method `n(B)` even affects all classes except `A`.

All effects of the aspect are visible in sub-figure (c), where the effects of the `declare parents: ...implements I` statements become visible. No 'old' base system code uses this effects as in the original hierarchy `C` and `D` did not implement `I`. So, for `C`, `D` and all their subclasses, methods `x()` and `y()` can only be called by the aspect. For class `C` only an implementation of `x()` is provided, for `y()` the default implementation of `I` is used—as is visible in the output.

3.9.2 Applying the Proposed Analysis

The analysis revealing classes only using the default implementation of an interface, like e.g. `E` does, is quite simple and not explicitly considered. The example concentrates on changes in lookup. Aspect `Complex` defines three inter type declarations (omitting the interface inter type declaration), `(B, n(B))`, `(B, n(A))` and `(D, i)`.

Hiding and Overriding

For each inter type declaration, we calculate function `lookupChanges`. However, the calculation for `lookupChanges(`$\mathscr{H}$`, `$\mathscr{H} \oplus_s \mathscr{H}'$`, (D, i), [D], [])` = [(i, D, D), (i, F, D)] is identical to the one shown in Example 3.8.1, so we will not repeat this calculation. Due to the above mentioned hierarchy modification the calculation for `lookupChanges(`$\mathscr{H}$`, `$\mathscr{H} \oplus_s \mathscr{H}'$`, (B, n(A)), [D], [])` however has changed:

Listing 3.4: Combined Aspect `Complex` applied to the running example of Figure 3.2. `main`-Methods have been added to demonstrate impact analysis.

```
1  class Main {
2    public static void main(String[] args) {
3      A pa=new A();
4      B pb=new B();
5      System.out.print("\nA: "); A a=new A(); a.n(pa); a.n(pb);
6      System.out.print("\nB: "); B b=new B(); b.n(pa); b.n(pb); b.m();
7      System.out.print("\nC: "); C c=new C(); c.n(pa); c.n(pb); c.m();
8      System.out.print("\nD: "); D d=new D(); d.n(pa); d.n(pb); d.m();
9      System.out.print("\nE: "); E e=new E(); e.n(pa); e.n(pb); e.m();
10     System.out.print("\nF: "); F f=new F(); f.n(pa); f.n(pb); f.m();
11     System.out.print("\nG: "); G g=new G(); g.n(pa); g.n(pb); g.m();
12     System.out.println();
13   }
14 }
15
16 aspect Complex {
17   // declare parent extends / implements
18   declare parents: D extends G;
19   declare parents: C implements I;
20   declare parents: D implements I;
21
22   // inter type declarations
23   public void I.y() {
24         System.out.print("I.y()\t");
25   }
26   void B.n(A a) {
27         System.out.print("B.n(A)\t");
28   }
29   void B.n(B a) {
30         System.out.print("B.n(B)\t");
31   }
32   int D.i;
33
34   public static void main(String[] args) {
35     A pa=new A();
36     B pb=new B();
37     System.out.print("\nA: "); A a=new A(); a.n(pa); a.n(pb);
38     System.out.print("\nB: "); B b=new B(); b.n(pa); b.n(pb); b.m();
39     System.out.print("\nC: ");
40     C c=new C(); c.n(pa); c.n(pb); c.m(); c.x(); c.y();
41     System.out.print("\nD: ");
42     D d=new D(); d.n(pa); d.n(pb); d.m(); d.x(); d.y();
43     System.out.print("\nE: ");
44     E e=new E(); e.n(pa); e.n(pb); e.m(); e.x(); e.y();
45     System.out.print("\nF: ");
46     F f=new F(); f.n(pa); f.n(pb); f.m(); f.x(); f.y();
47     System.out.print("\nG: "); G g=new G(); g.n(pa); g.n(pb); g.m();
48     System.out.println();
49   }
50 }
```

```
javac demo.java
java Main
A: A.n(A)   A.n(A)
B: A.n(A)   A.n(A)   B.m()
C: A.n(A)   A.n(A)   B.m()
D: A.n(A)   A.n(A)   B.m()
E: A.n(A)   A.n(A)   B.m()
F: F.n(A)   F.n(A)   B.m()
G: G.n(A)   G.n(A)   B.m()
```

(a) : Original System Output

```
ajc demo.java demo.aj
java Main
A: A.n(A)   A.n(A)
B: B.n(A)   B.n(B)   B.m()
C: B.n(A)   B.n(B)   B.m()
D: G.n(A)   B.n(B)   B.m()
E: B.n(A)   B.n(B)   B.m()
F: F.n(A)   B.n(B)   B.m()
G: G.n(A)   B.n(B)   B.m()
```

(b) : Output of `Main.main(..)` with aspect `Complex` applied.

```
ajc demo.java demo.aj
java Complex
A: A.n(A)   A.n(A)
B: B.n(A)   B.n(B)   B.m()
C: B.n(A)   B.n(B)   B.m()   C.x()   I.y()
D: G.n(A)   B.n(B)   B.m()   D.x()   D.y()
E: B.n(A)   B.n(B)   B.m()   C.x()   I.y()
F: F.n(A)   B.n(B)   B.m()   D.x()   D.y()
G: G.n(A)   B.n(B)   B.m()
```

(c) : Output of executing `Complex.main(..)`.

Figure 3.7: Example: Produced output. Section (a) shows the output of running `Main` without application of aspect `Complex`, section (b) the output of running `Main` with aspect `Complex` applied, and finally section (c) the output of running the `main()`-method of aspect `Complex`.

```
lookupChanges(ℋ, ℋ ⊕s ℋ', (B,n(A)), [B], []) =
lookupChanges(ℋ, ℋ ⊕s ℋ', (B,n(A)), [C,G], [(n(A),B,B)]) =
lookupChanges(ℋ, ℋ ⊕s ℋ', (B,n(A)), [G,E],
  [(n(A),B,B), (n(A),C,B)]) =
lookupChanges(ℋ, ℋ ⊕s ℋ', (B,n(A)), [E],
  [(n(A),B,B), (n(A),C,B)]) =
lookupChanges(ℋ, ℋ ⊕s ℋ', (B,n(A)), [],
  [(n(A),B,B), (n(A),C,B), (n(A),E,B)]) =
[(n(A),B,B), (n(A),C,B), (n(A),E,B)]
```

Finally, `lookupChanges(`$\mathscr{H}$`, `$\mathscr{H} \oplus_s \mathscr{H}'$`, (B, n(B)), [D], []) = []` as method `n(B)` is a new member.

Reason	Lookup Changes
Hiding	`(i,D,D), (i,F,D)`
Overriding	`(n(A),B,B), (n(A),C,B), (n(A),E,B)`
Overloading	`(n(B),B,B), (n(B),C,B), (n(B),D,B),` `(n(B),G,B), (n(B),E,B), (n(B),F,B)`
Hierarchy	`(n(A),D,G)`

Table 3.1: Summary of Lookup Changes.

Overloading

For each introduced method, we also have to check if the new method is more specific than existing methods as in that case the lookup also changes. Therefore we calculate `getOverloaded(`$\mathscr{H} \oplus_s \mathscr{H}'$`, (B, n(A)), [B], []) = []` and `getOverloaded(`$\mathscr{H} \oplus_s \mathscr{H}'$`, (B, n(B)), [B], []) = [ (n(B), B, B), (n(B), C, B), (n(B), D, B), (n(B), G, B), (n(B), E, B), (n(B), F, B) ]`. The first calculation yields no lookup changes, as `n(A)` is not more specific than any existing methods; the second calculation yields the same results as for Example 3.8.2 (although the calculation is different) as the observant reader may verify.

To capture overloading effects due to the hierarchy modification, we now also have to check if a newly accessible method in a new superclass—in this case only `G`—is more specific. `G` however only defines `n(A)` which is also present in the old superclass `B` and thus not relevant. In case a relevant method is defined, we treat this member like a virtual inter type declaration but only examine its effect on the new subtree (i.e. here class `D` and subclasses).

Hierarchy Modification

We finally calculate effects of hierarchy modifications by calculating `hierarchyChanges(`$\mathscr{H} \oplus_s \mathscr{H}'$`, (D,B,G), [(D, m(), n(A))], []) = [(n(A), D, G)]` as we saw in Example 3.8.3. Table 3.1 summarizes the total set of lookup changes.

3.10 Conclusion and Related Work

We finally summarize our impact analysis of static crosscutting effects and give an overview of related work on the subject.

3.10.1 Summary

In this chapter we showed how static crosscutting, or more precisely inter type declarations and hierarchy modifications, can modify system semantics. In general addition of new members can result in lookup changes, either due to hiding, overriding or overloading. We called this effect binding interference.

We extended and used the noninterference criterion of Snelting and Tip [21] to show that static crosscutting is semantics preserving, if no changed lookups due to static crosscutting occur and RTTI is excluded. We also describe how we calculate changed lookups by analyzing the original hierarchy and applied aspects.

The main purpose of the results presented here—while maybe informative for programmers by themselves—is to serve as input for more elaborate analysis techniques like the one describes in Chapter 4, where the analysis presented here will be used as a building block.

3.10.2 Related Work

To improve separation of concerns, several other approaches besides aspect oriented programming have been suggested. Especially relevant for the approach presented here are [18] and [6, 5].

In [18] Ossher and Tarr proposed multi-dimensional separation of concerns, leading to a separate implementation of different features and a composition of the resulting hierarchies according to user defined composition rules. Semantics of these compositions are a research topic addressed in [21]. Inter type declarations are an implementation of Open Modules [1]; this work has been discussed in Chapter 2.3.

In [6, 5] Batory et. al. present *Feature-Oriented Programming (FOP)* as a mechanism to synthesize programs based on features. A feature comprises a set of collaborating classes. Each feature can be refined by other features. This refinement approach–depending on the concrete implementation–merges classes similarly to the hierarchy combination approach presented by Hyper/J, but with a fixed set of rules. A possible refinement technique is mixin-based inheritance [7]. In [16], an approach to refacture legacy systems using FOP is presented. In[3], Appel and Batory examine the relation of FOP and AOP.

Merely syntactical support for programmers indicating the presence of members added by inter type declarations is implemented in the AspectJ Development Tools (ajdt) [2]. However ajdt has no support to determine the impact of inter type declarations on system semantics.

A preliminary version of the static crosscutting analysis and the application of the Snelting/Tip criterion to show noninterference has been published in [23]. Based on this paper Zechmann [26] implemented the proposed algorithms and extended them to also cover overloading in his master thesis (in German). We will discuss his work in more detail in Section 4.5. This chapter wraps up all these results and completes them.

Bibliography

[1] ALDRICH, J. Open Modules: Modular Reasoning about Advice. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2005), vol. 3586 of *Lecture Notes in Computer Science (LNCS)*, Springer, pp. 144–168.

[2] ANDY CLEMENT, A. C., AND KERSTEN, M. Aspect-Oriented Programming with AJDT. In *Proceedings of AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003* (July 2003).

[3] APEL, S., AND BATORY, D. When to use features and aspects?: a case study. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (New York, NY, USA, 2006), ACM Press, pp. 59–68.

[4] BACON, D. F., AND SWEENEY, P. F. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1996), ACM Press, pp. 324–341.

[5] BATORY, D., JOHNSON, C., MACDONALD, B., AND VON HEEDER, D. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Trans. Softw. Eng. Methodol. 11*, 2 (2002), 191–214.

[6] BATORY, D., AND O'MALLEY, S. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol. 1*, 4 (1992), 355–398.

[7] BRACHA, G., AND COOK, W. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1990), ACM Press, pp. 303–311.

[8] CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2000), ACM Press, pp. 130–145.

[9] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming* (London, UK, 1995), Springer-Verlag, pp. 77–101.

[10] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] HANENBERG, S., SCHMIDMEIER, A., AND UNLAND, R. AspectJ Idioms for Aspect-Oriented Software Construction. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP)* (Irsee, Germany, June 2003).

[12] JONATHAN G. ROSSIE, J., AND FRIEDMAN, D. P. An Algebraic Semantics of Subobjects. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1995), ACM Press, pp. 187–199.

[13] LADDAD, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[14] LEAVENS, G. T., AND MILLSTEIN, T. D. Multiple Dispatch as Dispatch on Tuples. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), ACM Press, pp. 374–387.

[15] LISKOV, B. H., AND WING, J. M. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[16] LIU, J., BATORY, D., AND LENGAUER, C. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ACM Press, pp. 112–121.

[17] MIKHAJLOV, L., AND SEKERINSKI, E. A Study of the Fragile Base Class Problem. *Lecture Notes in Computer Science (LNCS) 1445* (1998), 355–382.

[18] OSSHER, H., AND TARR, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach, 2000. Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.

[19] REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 432–448.

[20] RYDER, B. G., AND TIP, F. Change Impact Analysis for Object-Oriented Programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2001), ACM Press, pp. 46–53.

[21] SNELTING, G., AND TIP, F. Semantics-Based Composition of Class Hierarchies. In *ECOOP'02, Spain, June 10-14, 2002. Proceedings* (January 2002), vol. 2374 of *Lecture Notes in Computer Science (LNCS)*, p. 562.

[22] STOERZER, M. Analytical Problems and AspectJ. In *Proc. 3rd German Workshop on Aspect-Oriented Software Development, Essen, Germany* (March 2003).

[23] STOERZER, M., AND KRINKE, J. Interference Analysis for AspectJ. In *Proc. FOAL Workshop* (2003).

[24] TARR, P., OSSHER, H., HARRISON, W., AND STANLEY M. SUTTON, J. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 107–119.

[25] TIP, F., AND PALSBERG, J. Scalable Propagation-based Call Graph Construction Algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2000), ACM Press, pp. 281–293.

[26] ZECHMANN, H. Dynamische Impact Analyse für aspekt-orientierte Programme mit Hilfe des Call Graphen. Master's thesis, Universität Passau, October 2004.

4

Chianti-based Change Impact Analysis

This thesis so far gave an overview of AspectJ as the currently most popular aspect-oriented language and, based on a study of available published case studies, identified several major problems of AspectJ and most other currently available aspect-oriented languages.

As outlined in Section 2.2 one of the thesis' goals is to show that tools can considerably alleviate the problems introduced by aspect-oriented techniques. Tools can be used to analyze a program and thus help programmers understand the program and especially effects of applied aspects and code modifications.

A very interesting analysis technique to enable such tools is *change impact analysis*. This chapter gives an overview of the goals of change impact analysis and in detail introduces the change impact analysis technique of [46, 51]. This chapter also shows how change impact analysis can be used to find bugs (Section 4.3) and support early release of changes (Section 4.4). In Section 4.5 we finally show how these techniques can be lifted to analyze the effects of aspects on an existing (aspect-oriented) system.

4.1 Basic Concepts of Change Impact Analysis

Software is bound to change. In general a software system will be changed often during its lifetime, to fix problems or to adapt or enhance its functionality as a reaction to changed requirements. This is an integral part of software and completely independent of the underlying programming language or even programming paradigm used. However, although changing software on the one hand is very easy—just edit the code—on the other hand it is very hard to make sure that all changes work as expected. As even today up to 80% [38, 18] of the total cost for a software system stems from the maintenance phase, it is highly necessary that the impact of changes can be accurately and efficiently determined in advance to enable cost-efficient adaptations.

Note that assessing maintenance cost figures is difficult, as maintenance cost also includes perfection and adaptation and not only corrective efforts. As large software systems are major investments of companies, they are often extended and modified to cope with new or chaining requirements over decades, thus obviously resulting in a large percentage of 'maintenance cost' in their total life time cost compared to the initial development effort.[1] The important

[1] Let me illustrate this with an example: From my professional experience gained from working for a major IT

thing to note is that it is very important to understand the impact of adding new or changing existing modules on the existing system.

A good example to outline the importance of change impact analysis is the Y2K problem. While the underlying "change" in requirements is easy to understand—systems have to represent dates using four digits instead of only two, as two digits ran over and reverted to "00" in the year 2000—the consequences of this would-be simple change consumed estimated billions of euros world-wide—a tremendous cost. This problem resulted in research producing several analysis approaches tailored to trace flow of date-information in software systems, and often without these techniques a cost-efficient adaptation of existing software might not have been possible at all.

Manual analysis of code artifacts affected by a proposed change in general is very hard, as software systems today are large and complex. Automated software change impact analysis can leverage necessary efforts, as its major goal is to identify these software artifacts—from requirements over design, code, and tests to documentation. Deficiencies in or negligence of impact analysis techniques often results in increased costs and poor performance of change implementation as well as in questionable quality.

Following [6], change impact analysis can be subdivided in two major areas, *dependency analysis* and *traceability analysis*. Traceability is "the ability to trace between software artifacts generated and modified during the software product life cycle", i.e. traceability analysis focuses on identifying a concern from requirements-analysis to design to code artifacts, etc. Dependency analysis is defined as the process of "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change". For the reminder of this thesis we will use the term *(Change) Impact Analysis* as a synonym for Dependency Analysis; Traceability Analysis is no main topic of this thesis.

One of the major problems when analyzing change impact is that the direct impact can in turn result in transitive impacts. This observation has been captured by the terms *side and ripple effects*: following [22], a side effect is an error or other undesirable behavior that occurs as a result of a modification. [55] defines ripple effects as the effects caused by making a small change to a system which affects many other parts of a system. Ripple effect analysis emphasizes the tracing of impacts in source code when the source code is changed.

One of the main goals of Software Change Impact Analysis is to capture all relevant side- and ripple effects to give software engineers a *safe* estimate which software artifacts are affected by a proposed change. Transitive effects are the reason why manual change impact analysis is hard and error prone. Tools can automate this effort to some extent.

4.1.1 Changes and the Challenge of Impact Analysis

One of the main sources of change requests are changing requirements. Software is expensive and thus existing software systems are *major investments* for companies. As a result, such systems are not easily abandoned and replaced but instead are adapted to fit new or changed requirements emerging from a changed social, economical or legal environment.

Progress in hardware and algorithmic research is also a driving factor for change. Today computational power and matching algorithms are available nobody dreamed of 20 years ago. This computational power and these new algorithms allow to solve problems which clearly exhausted computational resources in the past.

As can be concluded from the above discussion, the lifetime of a large software system can be very long. Systems are maintained and used over decades, a very long time in rapidly evolving computer science. There are three important consequences stemming from this long life cycle:

service company in Germany, nearly no project starts 'from scratch', but builds on or extends existing systems with new modules to cope with new or changes requirements. Is this maintenance? Or development of new systems?

Lack of Tools: Old systems are often written in outdated languages. Most software in use today is still written in some COBOL dialect. Unfortunately for many of these languages development tools have not been maintained and thus as a worst case assumption are no longer available today. This might even include the compilers. Thus impact analysis in such environments is very hard and often completely manual.

Lack of Training: Programmer training at universities focuses on up-to-date techniques like modeling using the UML and object-oriented programming. So there is obviously a training mismatch. The underlying concepts in legacy systems are often relative low-level compared to state-of-the-art high level programming languages and concepts. Some programming concepts necessary to succeed in these environments are no longer taught today (manual memory management vs. garbage collection, manual handling of pointers vs. Java, programming without recursion, etc.).

Lack of System Knowledge: It is not very likely that the staff who initially developed the system is still working on it. Even if the original programmers are still available, it is not very likely that they exactly remember all the requirements and resulting dependences in the system. Thus developers faced with changing or new requirements have to deduce necessary information for change planning from available system artifacts, i.e. documented requirements, design decisions, specifications, test scenarios and finally the source code itself.

Additionally, the quality of the code in a software system tends to degrade with every change (law of entropy increase / second law of thermodynamics). Safely estimating the change impact with only limited system experience and a questionable or no longer recognizable design can be *very* hard.

To counter these effects, impact analysis tools can considerably help. In this thesis the focus is on dependency analysis, where code analysis techniques are used to derive dependences of software artifacts from the available code of the system, based on the semantics of the used programming language.

While most of the above discussion seems to cover legacy systems only, keep in mind that today we write the legacy systems of tomorrow. While today's programming concepts and languages are considerably more high-level than languages used 30 years ago, in 30 years current state-of-the-art concepts and languages will most likely be outdated. So the problems discussed above will potentially apply as well. However hopefully the computer scientists today will maintain development tools, so that these tools—or at least the necessary knowledge to build them—are available when needed. This thesis is a step in that direction.

4.1.2 Impact Analysis Process

To better understand impact analysis, we study why and how software systems change. In general a requirements change for a system emerges from changed conditions in the environment. Before a decision can be made whether to actually implement the new/changed requirement, it is important to give an estimate of the costs of implementation emerging from this change. Any change has to be cost-effective.

Therefore the requirement change is analyzed and traced down to the relevant (code) artifacts to be adapted in order to implement this change. Adaption of some artifacts on the other hand might transitively influence others (ripple and side effects). As a consequence the set of (code) artifacts affected by a change in general is larger than apparent at first sight. A simple but unfortunately often disregarded dependency for example is impact on (design) documentation—as a consequence design documents are often outdated. Comprehensive Impact Analysis should examine and trace dependences on all software artifacts, to increase confidence that all these documents are actually updated.

The core of impact analysis is a model reflecting dependences of different software artifacts. If we represent each artifact as a node and a dependency as an edge, a directed graph is a natural representation of such dependences. We will refer to this graph—or a similar representation of dependences—using the term *dependency model* in the following.

This model is general enough to represent all different kinds of dependences, as we did not limit the kind of artifact represented by a node. Thus it is possible to model dependences between documents created during several phases of the development process (e.g. from requirements to design documents to source code artifacts to test and back to requirements [6], p.10) as well as only a more code-centric view by only using source code artifacts like packages, classes, methods, statements or even single variables.

This dependency model is then evaluated to capture all the side- and ripple effects due to changes to directly affected code artifacts. Reachability analysis, i.e. calculating the transitive closure on the dependency model then is enough, although there are some interesting options here. For example presenting the complete closure might be too much, as transitive dependences are not necessarily affected by a change; this strongly depends on the kind of transitive dependency and thus on the internal structure of an affected artifact.

Based on this discussion, this thesis defines impact analysis as "identifying the consequences of a change, or estimating what needs to be modified to accomplish a change". Compared to the definition of [6], the "potential" has been dropped. Thus an optimal impact analysis result will safely and precisely identify the code artifacts which have to be adapted to implement a given change. *Safely* means that all relevant dependences and thus artifacts are captured, while *precisely* indicates that no additional irrelevant dependences and artifacts are reported.

4.1.3 Impact Analysis Algorithms

There are several different algorithms to calculate impacted artifacts for a given dependency model. Heuristic approaches use predefined rules and/or other heuristics to derive impacted artifacts from the dependency model. Note that these approaches in general only calculate a relaxation of the solution, but are not necessarily safe. For example stochastically guided algorithms use probabilities for dependences to determine affected artifacts. One could add a dependence probability to an edge in the graph representing the dependency model and multiply dependences encountered on a path from direct impacts to other reachable artifacts and abort search if the resulting probability drops below a given threshold.

Unguided/exhaustive approaches calculate the transitive closure for directly impacted artifacts in the dependency model in a brute force manner. The quality of exhaustive algorithms strongly depends on the quality of the underlying dependency model. If the model indeed captures all potential dependences, then the results of this kind of algorithms are safe. However, if the model over-approximates potential dependences, then the results will not be precise, and too many artifacts will be reported as affected. Thus the creation of a safe and precise dependency model is very important for change impact analysis.

Traceability Analysis

Before describing dependency analysis which is at the core of this thesis in more detail, we will first briefly discuss traceability analysis. Traceability of requirements down to the source code has attracted more interest during recent years.

Compared to dependency analysis capturing relevant dependences is hard to do automatically, as traceability analysis in general involves several documents in natural languages or other non-formal or only semi-formal documents. Thus traceability analysis often depends on manually or semi-automatically constructed relatively coarse grained dependency models.

As the model has been built manually it is potentially neither complete nor precise. Thus application of safe impact analysis techniques seems not too beneficial in this context.

Code or API documentation is an exception to that rule. Consider for example the Java documentation framework JavaDoc. Here, programmers can embed a special comment in the Java source code. The JavaDoc tool can then be used to automatically generate HTML API documentation from the source code. As a result, the dependency from code changes to documentation changes is more easily traceable (JavaDoc comments are part of the source code and next to the code artifact they describe).

Dependency Analysis

At the core of this thesis however is not traceability but dependency analysis. Subject to dependency analysis is only the source code itself as a subset of all documents of a software system.

Source code is considerably easier to analyze compared to natural languages. Code semantics and syntactical structure are in general (formally) defined[2], and thus code analysis can build on these semantics. Another major advantage besides a clear semantics is that dependency analysis can use a wealth of program analysis techniques originally designed to e.g. optimize code. These analysis techniques can often be reused to also capture dependences of code artifacts.

Examples are *program dependence graphs* as originally proposed by [21]. A program dependence graph (or PDG for short) is a data structure capturing program statements or variables as nodes and modeling possible flow of control or data among these nodes as edges. An application of these graphs is to analyze if two statements in a program can be reordered without changing code semantics (code motion problem). Naturally this is only possible if there is no data-flow from one statement to the other.

A typical problem for dependency analysis is to compute all ripple effects if a given statement is changed. To solve this problem, *program slicing* is a useful application to derive all program statements depending on or influencing a given statement (forward/backward slice). If a PDG is available, slicing is reduced to calculating all nodes reachable from the originally impacted statement, and the slice is the set of all nodes potentially affected by ripple effects of an original change. Slicing has been researched for years, compare for example [31] and [58].

Unfortunately slicing is also a relatively expensive technique, depending on the targeted accuracy. While the most simple algorithms have a theoretical complexity of O(n), the results achievable with such algorithms are rather imprecise and thus of limited value. More accurate algorithms however quickly reach high polynomial or even worst case exponential complexity. As shown by [43], a precise slice is even not computable in general as it depends on precise aliasing information.

Although we used slicing as an introductory example, there are several simpler techniques usable for dependency analysis. Examples are (ordered by increasing complexity) browsing, cross referencing, coverage analysis, control-flow analysis and finally data-flow analysis. As most modern slicing techniques rely on a PDG per definition representing the data-flow in a system, slicing can be classified as an advanced data-flow analysis technique.

The following sections will give a short exemplary introduction to coverage-based, control-flow based and data-flow based analysis techniques (without being exhaustive), as these techniques will be relevant in the remaining of this thesis.

Coverage Based Analysis Techniques: Coverage based techniques are dynamic analysis techniques. At the core of all dynamic analysis techniques is a controlled execution of a software system to record relevant execution data. A good technique to gather such data from

[2] At least by the used compiler. Java is a positive exception here as the language semantics are (mostly) defined by the Java Language Specification [24]. Note however that this definition is no formal language semantics.

a system is to trace execution of a test suite. To do so, several approaches exist, either by instrumenting the program or by using some tracing or debugging interface in the runtime system.

Gathered data for example can contain called methods (and returns), accessed data, etc. Traces allow to analyze a specific program run in detail. Compared to a static analysis technique, no approximations are necessary, i.e. the recorded data is precise. However, a major drawback of dynamic analysis is that any result derived from gathered data is only valid for the examined program runs but not in general, i.e. coverage based techniques are unsafe. An example of a coverage-based techniques is [35].

Wether dynamic analysis is applicable thus considerably depends on the context the analysis results are used in. Compared to the discussion above, dependency analysis based on dynamic analysis is not safe, as some impacts might be missed. On the other hand any results are precise. Programmers in general want a tool offering reliable data about potential problems. A large number of false positives is not acceptable for such tools. As a consequence, unsafe but precise dynamic analysis techniques might be preferable compared to imprecise static techniques if safety is no requirement.

Control-Flow Based Analysis Techniques: To reduce the amount of missed dependences, (static) call- and control-flow graphs have been analyzed. Reachability analysis in a call graph has been suggested as a starting point for dependence analysis in [6]. If a method is changed, any (transitive) caller is considered affected by the change. Call graphs however ignore data dependences and changes to data definitions and are thus unsafe. The precision of this approach also depends on the precision of the underlying call graph. For object-oriented languages (static) call graphs have to approximate dynamic binding, and are thus imprecise. Additionally, a calling relationship is not necessarily also a relevant change dependence, as this kind of analysis ignores the method-internal control flow. Callers can be completely unaffected by a change if they never execute a changed part of a method.

Data-Flow Based Techniques: Slicing as described in the initial example is a data-flow based technique. Slicing-based techniques are safe, but can be rather imprecise and expensive (or both). Case studies reported in literature show that slices can contain up to 90 % of the program [41], thus considerably reducing their effectiveness.

Hybrid Techniques: Recently many change impact analysis techniques, especially in the related field of *regression test selection*, use a hybrid approach. The goal of regression test selection is to determine whether a regression test is affected by a code change or not, to determine a minimal test suite to rerun. In other words regression test selection examines which tests—as a special kind of program artifact—are impacted by changes. Note that tests in general can also be connected to some requirements, as they test some functionality a program has to fulfill, i.e. indirectly regression test selection techniques can also be useful for traceability analysis. These hybrid approaches use dynamically gathered data together with static information derived from the source code of the system to improve result quality. One of these hybrid approaches is described in [49].

Different Approaches—Discussion

Rating dependency analysis techniques strongly depends on the needs the user has for the produced results. In general, dynamic techniques tend to be unsafe, but precise; static techniques in contrast tend to be safe, but imprecise. There is also a second trade-off between quality of the results and time spent producing them. For both static and dynamic techniques quality of analysis results can be improved by putting more effort into the analysis. Dynamic analysis in general profits if more (different!) program runs are analyzed, resulting in fewer

missed dependences. Static techniques can improve precision of results by using more precise but also more complex algorithms.

It is also important to note that the strict distinction between static and dynamic techniques is no longer possible, as newer approaches are either parameterizable with an either statically or dynamically created dependency representation [46] or simply combine both approaches to derive more reliable information for the user [20].

Recent research combining static and dynamic information has been very fruitful. If a combined approach is used, tools have an additional option in presenting results. Instead of classifying a code artifact as either (potentially) unaffected or (potentially) affected, a combination of the two approaches allows to establish three classes: (definitely) affected, potentially affected (uncertain information), and (definitely) unaffected artifacts, thus improving analysis results compared to both purely static and purely dynamic analysis techniques. Thus it seems that hybrid approaches are superior compared to purely static or dynamic approaches. However keep in mind that this in general also doubles the effort to calculate results.

Finally, the granularity of an approach also has to be considered, ranging from dependences on the module level down to relationships of individual statements. Such fine-grained information might not be relevant for users in all cases, but internally can improve precision at the cost of runtime and space complexity.

From these observations we classify available techniques, but abstain from rating these techniques against each other in general. Before discussing the change impact analysis technique of [46] in detail, we will first review related work from the area of change impact analysis and the closely related field of regression test selection.

4.1.4 Change Impact Analysis in Literature

We first review directly related work in the field of change impact analysis. Second, regression test case selection techniques are also discussed, as determining the subset of tests in a test suite affected by a code edit is a special case of change impact analysis. The literature study below is an updated and extended version of the study of related work in [46].

Change impact analysis techniques

As discussed before there is a wide range of different approaches. Early research in the field of change impact analysis mostly relied completely on static information [6, 32], while some modern approaches only utilize dynamic information, such as [35]. Hybrid methods like [41] however seem to be very successful recently. The change impact analysis method described in the following section (4.2) is such a combined approach, as it uses (i) static analysis for caculating a structured set of changes and (ii) dynamic call graphs to relate tests and changes.

In general impact analyses focus on finding *constructs of the program potentially affected by code changes*. First we discuss static techniques and then address the combined and dynamic approaches.

An early form of change impact analysis used reachability on a call graph to measure impact. This technique was presented by Bohner and Arnold [6] as “intuitively appealing” and “a starting point” for implementing change impact analysis tools. However, applying the Bohner-Arnold technique is not only imprecise but also not safe, because, by tracking only methods downstream from a changed method, it disregards callers of that changed method that can also be affected.

Kung *et al.* [32] described various sorts of relationships between classes in an object relation diagram (i.e., ORD), classified types of changes that can occur in an object-oriented program, and presents a technique for determining change impact using the transitive closure of these relationships.

More recently, Tonella's impact analysis [60] determines if the computation performed on a variable x affects the computation on another variable y using a number of straightforward queries on a concept lattice that models the inclusion relationships between a program's decomposition (static) slices [23]. Tonella reports some metrics of the computed lattices, but gives no assessment of the usefulness of his techniques.

A number of tools in the Year 2000 analysis domain [16, 44] use type inference to determine the impact of a restricted set of changes (e.g., expanding the size of a date field) and perform them if they can be shown to be semantics-preserving.

The change impact analysis technique in Orso et al. [41] uses a combined methodology, by correlating a forward static slice [59] with respect to a changed program entity (i.e., a basic block or method) with execution data obtained from instrumented applications. Each program entity change is thus associated with a set of possibly affected program entities. Finally, these sets are unioned to form the full change impact set corresponding to the program edit. Orso et al. [41] compare their change impact analysis to simple static change impact analyses consisting of reachability from a changed entity in static call graphs and interprocedural control flow graphs.

Law and Rothermel [35] present a notion of dynamic impact analysis that is based on whole-path profiling [34]. In this approach, if a procedure p is changed, any procedure that is called after p, as well as any procedure that is on the call stack after p returns, is included in the set of potentially impacted procedures. Breech et. al. [7] present a variant of this technique, where materialization of traces is avoided by online analyzing trace data using dynamic compilers.

Recently, the change impact techniques of [35, 41] were compared empirically in [42]. The results showed that these approaches are incomparable. In addition, *PathImpact* [35] does not seem practical for programs that generate large execution traces, whereas *CoverageImpact* [42] does seem practical, although it can be significantly less precise.

In [2] Apiwattanapong addressed the high space and time overhead of [35] and proposed a dynamic impact analysis technique based on execute-after sequences. These sequences capture execution events for methods such that always the first and last execution event is saved. The algorithm then considers all methods affected, which are executed *after* the first execution of a changed method. Compared to traces this tremendously reduces the overhead with comparable effectiveness. Both approaches however are—while per definition unsafe—also incomplete in the context of object-oriented programs, as the algorithms ignore dynamic dispatch and thus have no predictive power for inteded changes like addition of overriding/overloading methods.

Regression test selection

Regression Test Selection aims at reducing the number of regression tests that must be executed after a software change [49]. These techniques typically determine the entities in user code that are covered by a given test, and correlate these against those that have undergone modification, to determine a minimal set of tests that are affected.

Several notions of coverage have been used. For example, *TestTube* [8] uses a notion of module-level coverage, and *DejaVu* [49] uses a notion of statement-level coverage. For the latter a test is affected—the term *modification-traversing* is used in the paper—if the *execution traces*, i.e. the protocol of statements executed during test execution for running the test in both the unchanged and changed program version differs. To calculate modification-traversing tests, lines in execution traces (representing executed statements) are mapped to the nodes of control flow graphs (CfGs) for the individual methods in the system, for both the original and the changed program version. Both CfGs are the traversed simultaneously, guided by the execution trace. If the simultaneous traversal hits a node in the CfG, where the underlying statement is not equivalent for both program versions, then the test is modification-traversing.

This approach is safe and precise, in the sense that it selects all tests which are affected by changes, and never tests which do not execute changed code.

While the technique described above is for the C language, Harrold et al. present an extended version for Java as an object-oriented language in [27]. Here, the control flow graphs are extended to Java Interclass Graphs (JIGs) which additionally capture type and class hierarchy information, exceptional control flow and also model external code for libraries. Calling relationships between methods are modeled using Class Hierarchy Analysis. The authors claim that the resulting approach is again safe in the sense described above.

Bates and Horwitz [3] and Binkley [5] proposed fine-grained notions of program coverage based on program dependence graphs and program slices, with the goal of providing assistance with understanding the effects of program changes. In comparison to our work, this work uses more costly static analyses based on (interprocedural) program slicing and considers program changes at a lower-level of granularity, (e.g., changes in individual program statements).

In the work by Elbaum et al. [17], a large suite of regression tests is assumed to be available, and the objective is to *select* a subset of tests that meets certain (e.g., coverage) criteria, as well as an order in which to run these tests that maximizes the rate of fault detection. The difference between two versions is used to determine the selection of tests, but unlike our work, the techniques are to a large extent heuristics-based, and may result in missing tests that expose faults.

The change impact analysis of [41] can be used to provide a method for selecting a subset of regression tests to be rerun. First all the tests that execute the changed program entities are selected. Then, there is a check if the selected tests are *adequate* for those program changes. Intuitively, an adequate test set T implies that every relationship between a program entity change and a corresponding affected entity is tested by a test in T. In their approach, they can determine which affected entities are not tested (if any). According to the authors, this is not a safe selective regression testing technique, but it can be used by developers, for example, to prioritize test cases and for test suite augmentation.

4.1.5 Change Impact Analysis and AspectJ

In this thesis a combination of method- and statement-level static and dynamic impact analysis techniques will be used [46] to deal with the specific problems introduced by aspect-oriented language constructs. An important data structure for several analysis techniques proposed here is the call graph, either dynamically or statically constructed.

Based on call graph and structured delta information, we propose techniques to deal with the potential pitfalls of aspect-oriented programming, in particular to determine the impact of adding an aspect to a given software system, and to cope with evolution of aspect-oriented systems. Note that this is not the classical change impact analysis setting as outlined by [6], as the goal of our analysis is to identify the requirements (represented by test cases) affected by aspect introduction.

Before describing our application of Change Impact Analysis to AspectJ in Chapter 4.5, we will first introduce the change impact analysis approach of [46] by Barbara Ryder, Xiaoxia Ren and Frank Tip in detail in Section 4.2 as we based our approach on this technique. We also demonstrate its benefits to help programmers in locating failures (Section 4.3) and support team developed projects (Section 4.4). Sections 4.3 and 4.4 of this chapter present joint work with the PROLANGS research group at Rutgers, State University of New Jersey and Frank Tip of IBM Research.

4.2 The Change Impact Analysis of Chianti

While the goal—finding code artifacts affected by changes—and the basic approach of change impact analysis—building and evaluating a dependency model—is the same for any change impact analysis and any programming language, each programming paradigm has its own challenges. As we discuss a change impact analysis technique for Java, we have to address the peculiarities of object-oriented languages.

An important problem in this context is the correct analysis of dynamic dispatch in the dependence model. This is an interesting problem, as the behavior of virtual method calls can be affected due to creation of objects or addition and deletion of methods in different modules. This *non-locality of change impact* is qualitatively different and more important for object-oriented programs than for imperative ones (e.g., in C programs a precise call graph can be derived from syntactic information alone, except for the typically few calls through function pointers).

We build the analysis techniques presented here on a change impact analysis technique previously published in [46, 51]. In this section we will recapitulate this change impact analysis technique. This technique assumes that a test suite $\mathscr{T}$ of unit or regression tests associated with the system to analyze is available. As we are discussing *change* impact analysis, we also need two versions of the system to analyze, which will be called the *original* and the *edited version* in the following.

The dependence model for the analysis consists of two parts. First, from a structured comparison of the original and the edited program version the analysis derives a set of *atomic changes* $\mathscr{A}$. This is a purely static analysis. These changes are then mapped to nodes and edges of the call graphs created for the individual tests in $\mathscr{T}$. Thus call graphs for the tests are the second part of the dependence model. Here the analysis offers two options—either to use statically or dynamically created call graphs. For the experiments reported in this thesis *dynamic call graphs* have been used, as we examine specific tests; thus the dynamic call graphs are *precise*. As we use dynamic analysis we implicitly make the usual assumptions [27] that program execution is deterministic and that the library code and execution environment (e.g., JVM) remain unchanged.

The change impact analysis in a first step calculates the set of atomic changes $\mathscr{A}$, whose granularity is (roughly) at the method level. These atomic changes include all possible effects of the edit on dynamic dispatch. The next step is the construction of (dynamic) call graphs for the tests in the *original* program version. For the experiments reported here, we used the Java Virtual Profiling Interface (JVMPI) to trace execution of the test suite.

With these changes and the dynamic call graphs the analysis then determines the set of tests affected by some of the changes in $\mathscr{A}$—the so called *affected tests*. Informally the analysis checks if one of the nodes (representing methods called in the program) or edges (representing a certain method call) in the call graph can be associated with a change in $\mathscr{A}$. If such a node or edge can be found, then the test is affected.

For affected tests the behavior of the test in the edited program version might have changed. The analysis in a final step then determines the subset of all changes $\mathscr{A}$ affecting this particular test—the so called *affecting changes*. For this analysis a correct call graph in the edited program version is necessary. For affected tests this implies that these tests have to be rerun in the context of the *edited* program version to create appropriate call graphs. Informally we then collect all affecting changes by traversing the call graph and collecting all changes affecting either a node or edge in the call graph.

The primary goal of this change impact analysis was to provide programmers with tool support that can help them understand why a test is suddenly failing after a long editing session by isolating the changes responsible for the failure (i.e. the affecting changes derived in the last analysis step). From this work *Chianti*, a prototype change impact analysis tool implemented by Xiaoxia Ren, emerged. The original paper [46] also reports about the valida-

```
1  public class A {
2     public A(int i){ x = i; }
3     public void foo(){
4        x = x + 0;[1]
5     }
6     public void bar(){ y = x;[3] }
7     public void zap(){ }
8     public void zip(){ y = x;[5] }
9     public int x;
10    public static int y;[4]
11    public static int getY(){
12       return y;
13    }[6,7]
14 }
15
16 public class B extends A {
17    public B(int j){ super(j); }
18       public void foo(){ }
19       public void bar(){ x++;[2] }
20 }
21
22 public class C extends A {
23    public C(int k){ super(k); }
24    public void zap(){
25       x = 5;
26    }[8,9,10,11]
27 }
28
29 class D extends A {
30    public D(int l) {
31      super(l);
32    }
33    public void foo(){ x--;[12] }
34 }
```

(a)

```
35 public class Tests
36 extends TestCase {
37
38    public void testPassPass(){
39       A a = new A(5);
40       a.foo(); a.bar();
41       Assert.assertTrue(a.x == 5);
42    }
43
44    public void testPassFail(){
45       A a = new C(7);
46       a.foo(); a.zap(); a.zip();
47       Assert.assertTrue(a.x == 7);
48    }
49
50    public void testFailPass(){
51       A a = new B(8);
52       a.foo(); a.bar(); a.zip();
53       Assert.assertTrue(a.x == 9);
54    }
55
56    public void testFailFail(){
57       A a = new B(6);
58       a.foo(); a.bar();
59       Assert.assertTrue(a.x == 11);
60    }
61
62    public void testCrashFail(){
63       A a = new D(5); a.foo();
64       int i = a.x / (a.x - 5);
65       Assert.assertTrue(a.x == 5);
66    }
67
68 }
```

(b)

Figure 4.1: **(a)** Original and edited version of example program. The original program consists of all program fragments *except* those shown underlined. The edited program is obtained by adding all underlined code fragments. Each added fragment is labeled with the numbers of the corresponding atomic changes. **(b)** Tests associated with (both versions of) the example program.

tion of this tool against the 2002 revision history (taken from the developers' CVS repository) of *Daikon*, a realistic Java system developed by M. Ernst et al. [19]. *Chianti* has been integrated closely with Eclipse, a widely used open-source development environment for Java (see *www.eclipse.org*). *Chianti* is also a base component for the tools we introduce in the following sections.

4.2.1 The Change Impact Analysis of *Chianti*—By Example

Figure 4.1(a) shows two versions of a small example program. Here, the original version of the program consists of all program fragments *except* for those shown underlined; the edited version is obtained by adding all the underlined code fragments. Associated with the program are five *JUnit* tests, `testPassPass`, `testPassFail`, `testFailPass`, `testFailFail` and `testCrashFail` as shown in Figure 4.1(b).

Atomic Changes. The change impact analysis (presented in full detail in [46]) relies on the computation of a set of atomic changes $\mathscr{A}$, that captures all source code modifications at a semantic level amenable to analysis. *Chianti* uses a fairly coarse-grained model of atomic changes, with change categories such as added classes (**AC**), deleted classes (**DC**), added methods (**AM**), deleted methods (**DM**), changed method bodies (**CM**), added fields (**AF**), deleted fields (**DF**), change field initializer (**CFI**) and lookup changes (**LC**) (i.e., changes to dynamic dispatch), to mention the most important change categories. Regarding changes to method bodies (**CM** changes), note that *Chianti* generates *one* **CM** change regardless of the number of statements within the respective method's body that have been changed, as it uses a method-level analysis. Note further that the set of changes in this example is only a subset of all change categories *Chianti* defines. Changes in (static) initializers for example also result in a respective change category. Other changes not directly captured by an atomic change are mapped to atomic changes to capture the resulting differences in program semantics. For details, refer to [46].

Chianti also computes syntactic dependences between atomic changes. Intuitively, an atomic change A_1 is dependent on another atomic change A_2, if applying A_1 to the original version of the program without also applying A_2 results in a syntactically invalid program (i.e., A_2 is a *prerequisite* for A_1, $A_2 \preceq A_1$). These dependences can be used to construct syntactically valid intermediate versions of the program that contain some, but not all of the atomic changes. In related work, the tool *Crisp* has been created to serve that purpose. For more details on change dependences and *Crisp* refer to [9, 45].

It is important to understand that the *syntactic* dependences do not capture all *semantic* dependences between changes (e.g., consider changes related to a variable definition and a variable use in two different methods). This means that if two atomic changes, A_1 and A_2, affect a given test T, then the absence of a *syntactic* dependence between A_1 and A_2 does not imply the absence of a *semantic* dependence; that is, program behaviors resulting from applying A_1 alone, A_2 alone, or A_1 and A_2 together, *may all be different*.

Example 4.2.1 (Atomic Changes) *Figure 4.2 shows the atomic changes that define the two versions of the example program, numbered 1 through 12 for convenience. Each atomic change is shown as a box, where the top half of the box shows the category of the atomic change (e.g.,* **CM** *for changed method), and the bottom half shows the method or field involved (for* **LC** *changes, the declaring class and method are shown). An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2. Consider, for example, the addition of the assignment* `y = x` *in method* `A.zip()`*. This source code change corresponds to atomic change 5 in Figure 4.2. Adding this assignment will lead to a syntactically invalid program unless field* `A.y` *is also added. Therefore, atomic change 5 is dependent on atomic change 4, an* **AF** *change for field* `A.y`*.*

In some cases, a single source code change is decomposed into several atomic changes. For example, the addition of `A.getY()` produces atomic changes 6 and 7, where the former models the addition of an empty method `A.getY()`, and the latter the addition of its method body. Observe that atomic change 7 is dependent on atomic change 6, reflecting the fact that a method must exist before its body can be added. Change 7 is also dependent on change 4 (an **AF** change for field `A.y`), because adding the body of `A.getY()` will result in a syntactically invalid program unless field `A.y` is added as well.

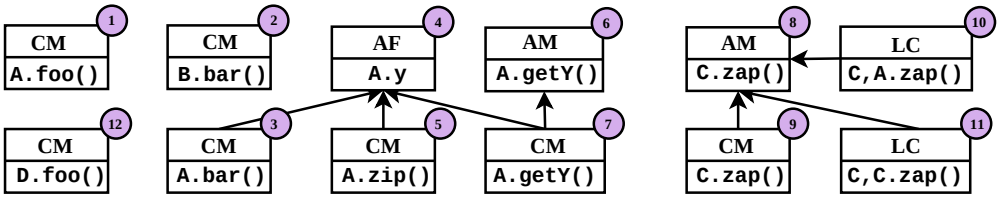


Figure 4.2: Atomic changes inferred from the two versions of the program.

The **LC** atomic change category models changes to the dynamic dispatch behavior of instance methods. In particular, an **LC** change $(Y, X.m())$ models the fact that a call to method $X.m()$ on an object of run-time type Y results in the selection of a different method. Consider, for example, the addition of method `C.zap()` to the program of Figure 1. As a result of this change, a call to `A.zap()` on an object of type `C` will dispatch to `C.zap()` in the edited program, whereas it dispatches to `A.zap()` in the original program. This change in dispatch behavior is captured by atomic change 10. Note, **LC** changes also may be generated as a result of a source code change affecting the class hierarchy, such as changing a method from *abstract* to *non-abstract* or from *private* to *public* [46].

Determining Affected Tests. In order to identify those tests that are affected by a set of atomic changes, a call graph is constructed for each test in the *original* program.[3] The analysis can work with call graphs that have been constructed either using static analysis, or by observing the actual execution of the tests (we used dynamic call graphs for the experiments reported here).

Example 4.2.2 (Affected Tests) *Figure 4.3 shows the call graphs for the tests of figure 4.1(b) in the original program. Edges corresponding to dynamic dispatch are labeled with a pair $< RT, M >$, where RT is the run-time type of the receiver object, and M is the method referenced at the call site. A test is determined to be affected if its call graph (in the original program) contains either (i) a node that corresponds to a* **CM** *(changed method) or* **DM** *(deleted method) change, or (ii) an edge that corresponds to a* **LC** *(lookup) change. In Figure 4.3 clearly all five tests are affected, because they each execute at least one method corresponding to a* **CM** *change. For example, the call graphs for* `testPassPass()` *and* `testPassFail()` *contain nodes corresponding to the changed method* `A.foo()` *(change 1).*

Determining Affecting Changes. In order to compute the set of changes affecting a given test, we construct a call graph for that test in the *edited* program. These call graphs are shown in Figure 4.4. The set of atomic changes that *affect* a given test includes: (i) all atomic changes for added (**AM**) and changed (**CM**) methods that correspond to a node in the call graph (in the edited program), (ii) lookup changes (**LC**) that correspond to an edge in the call graph, and (iii) their transitively prerequisite atomic changes.

Example 4.2.3 (Affecting Changes) *For example, the call graph for* `testPassFail` *in Figure 4.4 contains nodes corresponding to methods* `A.foo()`*,* `C.zap()`*, and* `A.zip()`*. These nodes correspond to atomic changes 1, 9, and 5 in Figure 4.2, respectively. The call graph for* `testPassFail` *also contains an edge labeled* `<C,A.zap()>`*, corresponding to atomic change 10. From the dependences in Figure 4.2, it can be seen that change 9 requires change 8, and change 5 requires change 4. Therefore, the affecting changes for* `testPassFail` *are 1, 4, 5, 8, 9, and 10. Similarly, we determine that 1, 3, 4 are the affecting changes for* `testPassPass`*, that 2, 4, 5 are the affecting changes for* `testFailPass`*, that only change 2*

[3] Call graphs contain one node for each method, and edges between nodes to reflect calling relationships between methods.

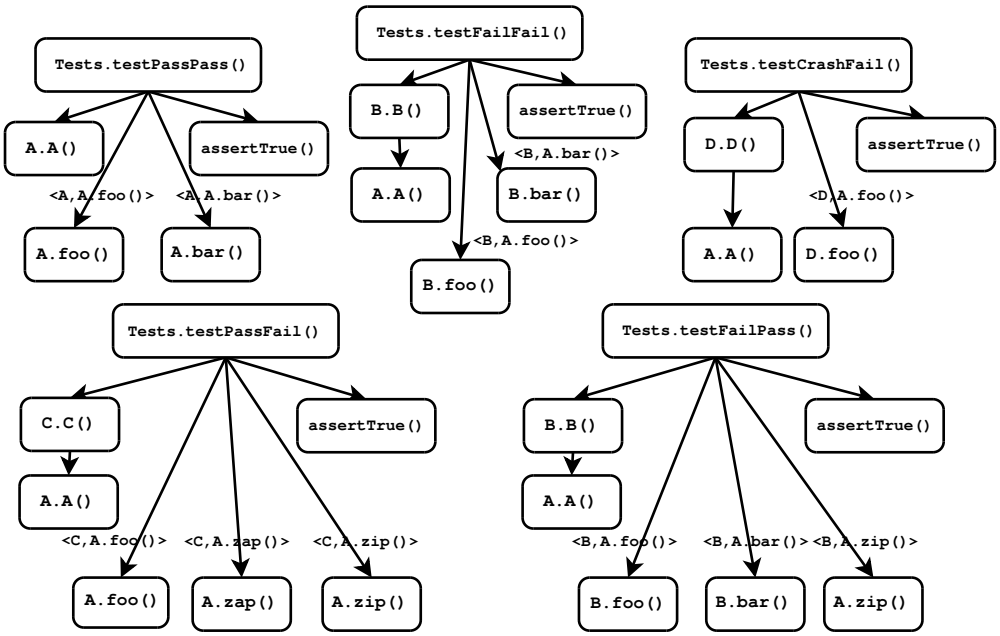


Figure 4.3: Call graphs for the original version of the program.

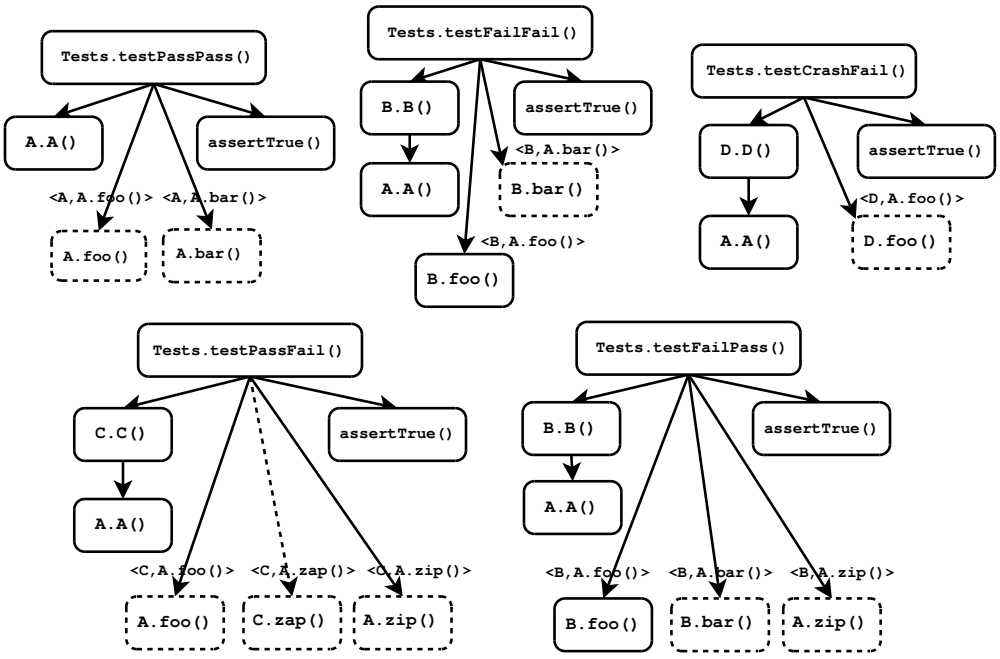


Figure 4.4: Call graphs for the edited version of the program. Dashed boxes indicate changed/added methods, and dashed arrows indicate changed calling relationships between methods (lookup changes).

Test	Affecting Changes
`testPassPass`	1, 3, 4
`testPassFail`	1, 4, 5, 8, 9, 10
`testFailPass`	2, 4, 5
`testFailFail`	2
`testCrashFail`	12

Table 4.1: Tests and Affecting Changes

$$
\begin{aligned}
&AT(\mathscr{T})_{\mathscr{A}} = \\
&\quad \{\, T_i \mid T_i \in \mathscr{T},\ Nodes(\mathscr{P}, T_i) \cap (\mathbf{CM} \cup \mathbf{DM})) \neq \emptyset \,\} \cup \\
&\quad \{\, T_i \mid\ \ T_i \in \mathscr{T},\ n, A.m \in Nodes(\mathscr{P}, T_i), \\
&\qquad\quad n \rightarrow_{B,\ X.m} A.m \in Edges(\mathscr{P}, T_i), \\
&\qquad\quad \langle B, X.m \rangle \in \mathbf{LC},\ B <^* X \,\}
\end{aligned}
$$

$$
\begin{aligned}
&AC(t)_{\mathscr{A}} = \\
&\quad \{\, a' \mid a \in Nodes(\mathscr{P}', T) \cap (\mathbf{CM} \cup \mathbf{AM}),\ a' \preceq^* a \,\} \cup \\
&\quad \{\, a' \mid\ a \equiv \langle B, X.m \rangle \in \mathbf{LC},\ B <^* X, \\
&\qquad\quad n \rightarrow_{B,\ X.m} A.m \in Edges(\mathscr{P}', T), \\
&\qquad\quad \text{for some } n, A.m \in Nodes(\mathscr{P}', T),\ a' \preceq^* a \,\}
\end{aligned}
$$

Figure 4.5: Affected Tests and Affecting Changes.

affects `testFailFail` *and that only change 12 affects* `testCrashFail`. *The table in Figure 4.1 summarizes these results.*

4.2.2 Formal Definitions

We will use the equations in Figure 4.5 (taken from [51]) to more formally define how *Chianti* finds affected tests and their corresponding affecting atomic changes, in general. Assume the original program $\mathscr{P}$ is edited to yield program $\mathscr{P}'$, where both $\mathscr{P}$ and $\mathscr{P}'$ are syntactically correct and compilable. Associated with $\mathscr{P}$ is a set of tests $\mathscr{T} = T_1,...,T_n$. The call graph for test T_i on the original program, called G_{T_i}, is described by a subset of $\mathscr{P}$'s methods $Nodes(\mathscr{P}, T_i)$ and a subset $Edges(\mathscr{P}, T_i)$ of calling relationships between $\mathscr{P}$'s methods.[4] Likewise, $Nodes(\mathscr{P}', T_i)$ and $Edges(\mathscr{P}', T_i)$ form the call graph G'_{T_i} on the edited program $\mathscr{P}'$. Here, a calling relationship is represented as $D.n() \rightarrow_{B,X.m()} A.m()$, indicating possible control flow from method $D.n()$ to method $A.m()$ due to a virtual call to method $X.m()$ on an object of type B.

To derive the subset of $\mathscr{T}$ affected by at least one change in $\mathscr{A}$, we apply the first equation in Figure 4.5. Note that the equations are parameterized with the set of changes $\mathscr{A}$, although $\mathscr{A}$ does not appear directly in the formulas. However, the sets of **AM**, **CM**, **DM** and **LC** changes depend on $\mathscr{A}$.

The equations always union two sets. The first set comprises all tests, where at least one of the nodes in the call graph of the test (in the *original* version of the program) can be associated with either a **DM** or **CM** change, i.e. a method which has been executed in the original program version has either been changed or deleted. The second set now examines edges in the call graph. Recall that edges are labeled with the calling method, the actual type of the caller and the called method, e.g. $D.n() \rightarrow_{B,X.m()} A.m()$. A test is also affected if the

[4] Note that for simplicity we identify methods and nodes in the call graph as well as calling relations and edges.

set of changes contains a lookup change for this respective call. If $\mathscr{A}$ contains a **LC** change $B,X.m()$ the above edge would result in addition of the respective test to the set of affected tests.

The second formula is used to determine the set of affecting changes. Here, we apply the formula to the newly created call graphs for each affected test T in the *edited* program version. Similarly to the first formula deriving the set of affected tests, we examine the call graph of each affected test. However, we do not terminate the traversal once a node or edge matching a change is found but instead collect all affecting changes. Again the affecting changes consist of two parts. First we examine for each node if it is associated with either an **AM** or **CM** change (**DM** changes of course are not relevant for the new call graphs). The second set the examines the edges and collects relevant associated lookup changes. Finally all transitive prerequisite changes are collected to derive the set of affecting changes.

In this thesis we only give a relative short introduction to the change impact analysis of *Chianti*, as this is prior work by B.Ryder, F.Tip, X.Ren and O.Chesley which we use to build our analysis techniques on. Note that Chianti statically analyzes changes in the class hierarchy to derive relevant lookup changes, and thereby also captures lookup changes due to overloading/overriding, threads and concurrency, and exception handling. Lookup changes are also generated if method visibility or inheritance relations change. For details however we refer to [46].

4.3 Finding Buggy Code using JUnit/CIA

While the calculation of affecting changes can already considerably help programmers if tests unexpectedly fail, a combination of atomic changes with test results can further help to increase *focus on failure inducing changes*, as we will show in this section. This section reports about joint work together with the PROLANGS research group at Rutgers and Frank Tip of IBM Research. A previous version of this work has been published in [56].

4.3.1 The Scenario

In modern software development, coding and testing are performed in interleaved fashion to assure code quality. Current development strategies rely heavily on the availability of a test suite to allow a programmer to quickly assess the impact of edits on program functionality. Difficulties occur when testing reveals unexpected behaviors, such as assertion failures or exceptions.

If a test failure occurs, the programmer however only knows that a bug has been introduced, but not exactly which part of the edit is responsible for the test failure. If the edits are trivially small, it may be easy to find the buggy code. However, as a code base and its test suite grow in size, running *all tests* after each minor change may become infeasible[5], and the number of changes that occur between successive executions of the test suite is likely to increase. This is especially true if some supposedly unrelated tests fail after an edit is done, once the *whole test suite* is rerun in contrast to just rerunning single module tests. Then, when test failures occur, it may be difficult to isolate the failure inducing change(s), and tedious manual debugging may be needed. While *Chianti* already eases this task by providing the set of affecting changes per test, the set of affecting changes can be large. With this work we want to provide additional focus on the problematic among all affecting changes.

In this section we present an approach for identifying failure inducing changes in a system with an associated regression test suite. In contrast to Extreme Programming (XP) [4] where

[5] For example, the Eclipse compiler had a test suite of 8803 tests (4830 parser tests + 3973 regression tests) on January 1, 2005. Executing this test suite takes more than 45 minutes on an AMD Athlon64 3200Mhz PC with 2GB RAM. Note that this is also the problem the large body of work on *regression test case selection* tries to solve.

the number of changes between test runs tends to be small, we assume that the size of an edit can become sufficiently large to make the identification of failure inducing changes a difficult task, and to make automated assistance with this task desirable.

Our change classification technique relies on the change impact analysis presented in Section 4.2 [46] to find the tests potentially affected by an edit (i.e., a set of changes), and to associate with each such test, a set of affecting changes. It then classifies these affecting changes as *Red*, *Yellow*, or *Green*, depending on whether they affect

(i) tests whose outcome has *improved*,

(ii) tests whose outcome has *degraded*,

(iii) tests whose outcome has remained *unchanged*,

(iv) or some combination of the above cases.

To explore the usefulness of change classification we designed a number of classifiers that assign the colors *Red*, *Yellow*, and *Green* to changes in slightly different ways. Our goal has been to develop classifiers for which *Red* changes are highly likely to be failure inducing, *Green* changes are highly unlikely (if not impossible) to be failure inducing, and *Yellow* changes fall somewhere in between. With these classifiers we set out to answer the following research questions:

1. Does it work? Can we distinguish failure inducing changes from other changes through change classification?

2. Which classifier is best? Is there a single change classifier that is always superior to all others, or do different classifiers work better for different applications?

3. If there is no "best" change classifier, is there a set of characteristics of an application that can be used to predict the classifier that will be most effective for it?

To answer these questions, we implemented five change classifiers in a tool called *JUnit/CIA*, an extension of the Eclipse component that integrates the popular *JUnit* testing framework with the Eclipse IDE (see `www.junit.org` and `www.eclipse.org`). The name of the tool reflects the fact that the functionality of *JUnit* is extended with features for Change Impact Analysis. *JUnit/CIA* relies on *Chianti* [46] for:

(i) dividing a program edit into its constituent *atomic changes*,

(ii) identifying tests *affected* by the edit by correlating (dynamic) call graphs for the tests with the atomic changes, and

(iii) determining *affecting changes* for each of these tests.

JUnit/CIA then classifies changes according to one of the five classifiers and visualizes them using a small extension of *JUnit*'s user-interface in Eclipse.

We envision *JUnit/CIA* to be used when running conventional JUnit tests reveals an unexpected test failure. *JUnit/CIA* can be used to compare a current faulty program version to an earlier, successfully tested program version derived either from Eclipse's local history, extracted from a version control repository or simply stored as a local copy in the workspace. *JUnit/CIA* then classifies changes to help the programmer identify the failure inducing ones. The programmer fixes the problem, and a new successfully tested program version is created.

The main contributions of this research are:

1. We designed a method to identify failure inducing changes in which changes are classified as *Red*, *Yellow*, or *Green* according to one of several change classifiers.

2. We implemented this method in a practical tool, *JUnit/CIA*, based on *JUnit* and Eclipse.

3. We conducted two case studies in which we measured the relative effectiveness of change classification on applications for which we manually identified failure inducing changes. These case studies indicated that change classification can focus programmer attention effectively on likely sources of failure; however, they were inconclusive with respect to selecting a "best" classifier.

The following sections describe how we classify changes, first informally by continuing the previously used example, followed by an in depth description of the different classifiers we used.

4.3.2 Change Classification—By Example

We will continue the example we used in Section 4.2.1 (page 78) to show how we classify changes. In contrast to *Chianti*, we now also use test results to create additional focus on potentially failure inducing changes among all changes affecting a given test.

We assume that the tests of Figure 4.1(b) will be used with both the original and edited versions of the program. The name of each test indicates its outcome in each version of the program; for example, `testPassFail` passes in the original program, but fails in the edited version. By examining the edited program and tests, we can observe that the addition of method `C.zap()` causes the failure of `testPassFail` and that this is the only test failure due to the edit. Note, we in general assume that the reason for the failure of tests like `testFailFail` failing in both program versions is the same in both versions of the program. For the example this is actually the case as `B.bar()` does not have the expected side effect.

Our classification of tests is based on the *JUnit* test result model in which a test can PASS, FAIL (i.e., an assertion failure) or CRASH (i.e., an exception is caught by the *JUnit* runtime). Note that the latter case only addresses *unexpected exceptions* which were not anticipated by the test case. It is easily possible to write *JUnit* test cases that handle expected exceptions and then PASS.

Thus far, we have seen that there are 12 atomic changes, and that the behavior of each of the five tests is affected by one or more of these changes. The goal of change classification is to answer the following question: *Which of those 12 changes are the likely reason(s) for the test failure(s)?* We provide an answer to this question by classifying the changes according to the tests that they affect. Intuitively, our goal is the following:

- A change that affects only *improving tests*, (i.e., tests such as `testFailPass` that fail in the original program, but that succeed in the edited version) is classified as *Green*. For example, change 12 (**CM** for `D.foo()`) only affects `testCrashFail` and thus should be colored *Green*. We consider CRASH to be a worse result than FAIL, because in conducting the experiments described in Section 4.3.6, we observed several bugs (typically of-by-one errors resulting in `ArrayIndexOutOfBoundsExceptions` or accesses via not initialized references resulting in `NullPointerExceptions`) that resulted in changing the result of a test from FAIL to CRASH.

- A change that affects only *worsening tests*, (i.e., tests such as `testPassFail` that succeed in the original program, but that fail in the edited version) is classified as *Red*. For example, changes 8, 9, 10 (**AM** and **CM** for `C.zap()` and **LC** for <`C`,`A.zap()`>) only affect `testPassFail`, so they are *Red*.

- A change that affects both improving tests and worsening tests is classified as *Yellow*. For example, change 4 (**AF** for `A.y`) affects both `testPassFail` and `testFailPass` and therefore is *Yellow*.

Intuitively, *Red* changes are most likely to be the reason for a test failure, followed by *Yellow* changes, and then *Green* changes.

How to associate colors with changes becomes less obvious when changes also affect tests that have the same outcome in both program versions. Section 4.3.3 defines a number of classifiers that follow different strategies. For two of these change classifiers (R_s/G_r, R_s/G_s), only changes 8, 9 and 10 are colored *Red*, exactly the failure inducing changes cited earlier for this example.

4.3.3 Test Model and Classification

Before classifying changes we first have to establish some properties of the test suite. Definition 4.3.1 below formalizes this test result model[6] and introduces an ordering in which passing tests are preferred over failing tests, and failing tests are preferred over crashing tests.

Definition 4.3.1 (Test Result Model) *Let $\mathscr{R}$= { PASS, FAIL, CRASH} be the set of all test results. Furthermore, we define the following ordering on test results:*

$$\text{CRASH} < \text{FAIL} < \text{PASS}.$$

For a given test T, we will use $R_{orig}(T)$ and $R_{edit}(T)$ to represent the result of test T in the original program and the edited program, respectively, where $R_{orig}(T), R_{edit}(T) \in \mathscr{R}$. Definition 4.3.2 below uses this notation to classify tests as worsening or improving.

Definition 4.3.2 (Test Classification) *Let $\mathscr{T}$ be the set of all tests. Then the sets WT and IT of worsening tests and improving tests, respectively, are defined as follows:*

$$\begin{aligned} WT &= \{\, T \in \mathscr{T} \mid R_{orig}(T) > R_{edit}(T) \,\} \\ IT &= \{\, T \in \mathscr{T} \mid R_{edit}(T) > R_{orig}(T) \,\} \end{aligned}$$

In the definitions below, we will again use the notation $AT(A)$ to represent the tests in $\mathscr{T}$ affected by atomic change $A \in \mathscr{A}$ (i.e., the set of all atomic changes between two versions) and $AC(T)$ to represent the atomic changes affecting a given test $T \in \mathscr{T}$.

4.3.4 Change Classification

In this section, we define criteria for change classification and present several change classifiers based on these criteria. Definition 4.3.3 defines auxiliary change sets *Worsening*, *SomeFailFail*, *SomePassPass*, *Improving*, and *OnlyPassPass*. *Worsening* and *Improving* are the sets of changes that affect at least one worsening test, or at least one improving test, respectively. *SomeFailFail* and *SomePassPass* are the sets of changes that affect at least one test that crashes/fails or passes in both versions, respectively. Finally, *OnlyPassPass* is the set of changes that only affect tests that pass in both versions.

Definition 4.3.3 (Change Influence) *Depending on the classified tests a change affects, we define auxiliary change sets as follows:*

$$\begin{aligned} \textit{Worsening} &= \{\, A \mid A \in \mathscr{A},\ WT \cap AT(A) \neq \emptyset \,\} \\ \textit{Improving} &= \{\, A \mid A \in \mathscr{A},\ IT \cap AT(A) \neq \emptyset \,\} \\ \textit{SomeFailFail} &= \{\, A \mid \exists T \in AT(A), \\ &\qquad R_{orig}(T) = R_{edit}(T) \in \{\, \text{FAIL}, \text{CRASH} \,\} \,\} \\ \textit{SomePassPass} &= \{\, A \mid \exists T \in AT(A), \\ &\qquad R_{orig}(T) = R_{edit}(T) = \text{PASS} \,\} \\ \textit{OnlyPassPass} &= \{\, A \mid \forall T \in AT(A), \\ &\qquad R_{orig}(T) = R_{edit}(T) = \text{PASS} \,\} \end{aligned}$$

Criteria		
Color	**relaxed**	**strict**
Red	R_r: $(A \notin Improving \wedge A \in Worsening)$	R_s: $(A \notin Improving \wedge A \in Worsening \wedge A \notin SomePassPass)$
Green	G_r: $A \in OnlyPassPass \vee (A \in Improving \wedge A \notin Worsening)$	G_s: $A \in OnlyPassPass \vee (A \in Improving \wedge A \notin Worsening \wedge A \notin SomeFailFail)$
Yellow	$A \notin Red, A \notin Green, AT(A) \neq \emptyset$	

Table 4.2: Definitions of four methods for classifying atomic changes into *Red*, *Yellow*, and *Green* changes.

We can finally classify changes as *Red*, *Yellow*, or *Green*. Intuitively, our goal is to classify changes such that *Red* changes are highly likely to be the reason for test failures, *Yellow* changes are possibly problematic, and *Green* changes are correlated with successful tests. There are several ways in which one could design such a classifier, and it was not clear to us *a priori* which approach would work best in practice. As we wanted to explore the potential of change classification, our approach was to define five different classifiers that each partition the set of changes into *Red*, *Yellow*, and *Green* subsets in slightly different ways. In Section 4.3.6 we will present a comparative evaluation of these different classifiers in two case studies. The five classifiers we defined are a first step to explore the potential of change classification, and we plan to investigate the effectiveness of other classifiers as future work.

The first classifier is called *simple* and relies *only* on test results in the edited program. A change is classified *Red* if it only affects failing or crashing tests, *Green* if it only affects passing tests, and *Yellow* otherwise.

To define the remaining four classifiers, we use a *relaxed* and a *strict* criterion based on the *development of test results* for the two versions for each color, as shown in Table 4.2. We will refer to these criteria as R_r, R_s, G_r and G_s, where the capital letter represents the color, and the subscript represents the criterion used, where *r* indicates *relaxed* and *s strict*. Tests that are new or that have been deleted in the edited program have no effect on the classifiers built from these criteria, as they do not correlate with improved or degraded test results.

Intuitively, the G_r criterion marks as *Green* any change that affects improving tests but not worsening tests, as well as any change that only contributes to tests that succeed in both versions of the program. While this is a reasonable criterion, it may have the somewhat counterintuitive effect that a *Green* change may affect a test that fails in the edited version of the program.

Example 4.3.1 *In the example in Figure 4.1, change 2 affects both* `testFailPass`*, an improving test, and* `testFailFail`*; it will be colored Green by the G_r criterion.*

The G_s criterion eliminates such potentially confusing effects by requiring that all *Green* changes must only affect tests that succeed in the edited program, causing change 2 to be colored *Yellow*. Note that both the G_r and the G_s criteria have the desirable property that changes classified as *Green* are never failure inducing, since they never affect any worsening test.

The difference between R_r and R_s is similar. The R_r criterion marks as *Red* any change that affects worsening tests but not improving tests. This is reasonable, but it may have the counterintuitive effect that a change that affects a test succeeding in both versions of the program may still be *Red* (e.g., change 1 in our example). The R_s criterion further restricts *Red* changes to affect only tests that FAIL or CRASH in the edited program. Any changes that are colored neither *Red* nor *Green* are classified as *Yellow* if they affect some tests.

[6]Our approach can easily be adapted to accommodate other test result models with, for example, a single error state or multiple fine-grained error states.

Change	*simple*	R_r/G_r	R_s/G_r	R_r/G_s	R_s/G_s
1	*Yellow*	*Red*	*Yellow*	*Red*	*Yellow*
2	*Yellow*	*Green*	*Green*	*Yellow*	*Yellow*
3	*Green*	*Green*	*Green*	*Green*	*Green*
4	*Yellow*	*Yellow*	*Yellow*	*Yellow*	*Yellow*
5	*Yellow*	*Yellow*	*Yellow*	*Yellow*	*Yellow*
6	*Gray*	*Gray*	*Gray*	*Gray*	*Gray*
7	*Gray*	*Gray*	*Gray*	*Gray*	*Gray*
8	*Red*	*Red*	*Red*	*Red*	*Red*
9	*Red*	*Red*	*Red*	*Red*	*Red*
10	*Red*	*Red*	*Red*	*Red*	*Red*
11	*Gray*	*Gray*	*Gray*	*Gray*	*Gray*
12	*Red*	*Green*	*Green*	*Green*	*Green*

Table 4.3: Classification of the atomic changes of Figure 4.2 according to the *simple* classifier and the 4 composite classifiers based on the criteria defined in Table 4.2.

As we can apply these two criteria for *Red* and *Green* independently, we obtain four classifiers by combining them. We will refer to these classifiers as R_r/G_r, R_s/G_r, R_r/G_s, and R_s/G_s. Note that there is an asymmetry in the four non-*simple* change classifiers. A change that affects only tests that PASS in both versions is always classified as *Green*, whereas a change that affects only tests that FAIL in both versions is always classified as *Yellow*. To motivate this decision, recall that the purpose of our change classification is to reveal failure inducing changes. A change that only affects passing tests by definition is not failure inducing (for the current test suite) and is therefore classified as *Green*. In contrast, if a change A affects a test that fails in both versions, the failure in the edited version may reflect the same problem as before, or it may now be due to A; therefore, *Yellow* seems a more appropriate choice than *Red*.

Some changes do not affect any tests. We classify a change A as *Gray*, if it has no affected tests (i.e., $AT(A) = \emptyset$). This is a coverage issue rather than a debugging issue, as it indicates that the test suite should be expanded to cover *Gray* changes as well.

Example 4.3.2 *Table 4.3 shows how the changes of the example of Figure 4.2 are classified according to our five classifiers.*

4.3.5 The *JUnit/CIA*-Prototype

To evaluate our change classifiers we created the tool *JUnit/CIA*, implemented as an Eclipse plug-in that builds on the analysis component of the previously developed *Chianti* tool [46]. *JUnit/CIA* uses the version of the program that is currently in the Eclipse workspace as the *edited version*, and either uses another existing project as the *original version* or retrieves a previous version from the local history that corresponds to the last time the test suite was executed. (The local history is a local RCS repository maintained by Eclipse that records all textual changes.) Dynamic call graphs for the tests are obtained by monitoring their execution using the JVMPI profiling interface.

We envision *JUnit/CIA* to be used when running conventional JUnit tests reveals a new test failure. *JUnit/CIA* can be used to compare the faulty program version that exists at that point in time to an earlier, consistent program version derived either from Eclipse's local history or extracted from a version control repository. *JUnit/CIA* then classifies changes to help the programmer identify the failure inducing ones. The programmer fixes the problem, and a new consistent program version is created.

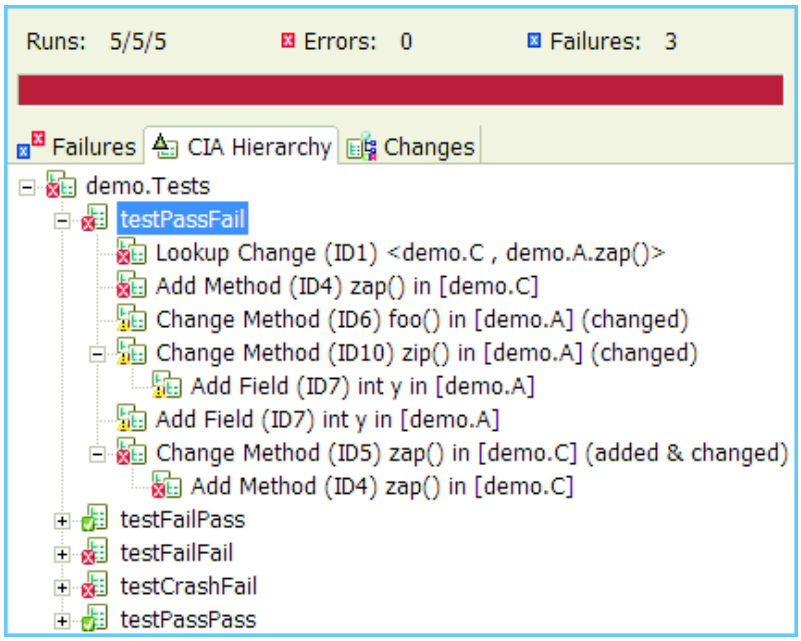


Figure 4.6: *JUnit/CIA* hierarchy view

The user-interface of *JUnit/CIA* extends that of the *JUnit* Eclipse component as follows: (i) in the *CIA* hierarchy view, affecting changes are shown in a tree-view underneath each test, where expanding the tree reveals prerequisite changes (see Figure 4.6), and (ii) an additional view shows all the changes organized by category (i.e., **AM**, **CM**, etc.). In each of these views, colored icons are associated with changes to indicate if they are *Red*, *Yellow*, *Green*, or *Gray*, and double-clicking on a change causes a standard Eclipse compare view of the associated original and edited code to appear.

Dealing with Infinite Test Runs

During our first case study with student projects, we encountered several situations where tests did not terminate. To handle such cases, we implemented a time-out mechanism where the execution of a test is aborted after a specified number of seconds. In such cases, we used the dynamic call graph obtained by executing the program up to that point, and consider the test result to be CRASH. We extended the standard *JUnit* launch configuration to allow users to specify this time-out option.

Note however that this approach in general can not guarantee that the call graph is actually complete in all cases, as we cannot distinguish a program in an infinite loop from very long running program which would actually terminate with no failures. Choosing a reasonable time out is thus important. For our experiments we chose a time out of 10 s, as in general the tests finished in less than a second.

Dealing with Libraries

In order to improve performance, we implemented a filtering mechanism that allows users to avoid tracing of methods in the standard libraries. Although, by assumption, such methods do not contain any changes, they may execute virtual method calls that dispatch to methods in user code (i.e., call-backs), and such dispatch operations may exhibit changed behavior when overridden library methods are added, deleted, or changed.

Example 4.3.3 (Changed Library Dispatch) *Assume that we change the classes presented*

Listing 4.1: Change of library-internal method dispatch due to changes in user code.

```
1  class Client {
2    public void foo() {
3       C c = new C(``some class'');
4       Set set = new HashSet();
5       set.add(c);
6       System.out.println(set);
7    }
8  }
9
10 class C {
11   String itemName;
12   C(String name) {
13      itemName = name;
14   }
15 }
```

in Listing 4.1 by adding an own implementation of the `toString()` *method to class C, so overriding* `Object.toString()`*. As a result, the behavior of the program changes, as in this example the dispatch for a call to* `toString()` *for a C-object now executes the new method and no longer* `Object.toString()` *as before. As the corresponding edge in the call graph for the call* `HashSet.toString()` $\rightarrow_{<\texttt{C,toString()}>}$ `Object.toString()` *is* within *the library part of the call graph, the changed behavior (or affection) will not be detected by the analysis.*

As we assume that library code and runtime environment remained unchanged, changed dispatch behavior is the only way how additional library call-backs can occur. As we want the analysis to be safe in this case, we provide a way to conservatively approximate all these potential influences. In the following, we will extend the underlying change impact analysis accordingly.

Definition 4.3.4 (Library and System types) *Let $\mathscr{C}$ be the total set of types. Then $\mathscr{C}$ is partitioned into Lib $\subseteq \mathscr{C}$, the set of types defined by program libraries, and App $\subseteq \mathscr{C}$, the set of types defined in source code.*

Beside the location where a type is defined, inheritance relations are relevant.

Definition 4.3.5 (Subtyping relation) *Let X, X' be types. If X is a super type of X', we write $X \trianglerighteq X'$ (or $X \triangleright X'$, if $X \neq X'$). If class X defines (i.e. provides an implementation for) a method m, we write defines(X,m).*

Let $X \in App$ be a type and $L \triangleright X \in Lib$ its super type and assume an add method change $\mathbf{AM}_{\texttt{X.m()}}$, overriding method `L.m()`. This can result in the following lookup changes:

- $\mathbf{LC}_1 =< X, L.m() >$ (the lost lookup)
- $\mathbf{LC}_2 =< X, X.m() >$ (the new lookup)

The edge corresponding to a call to $\mathbf{LC}_1$ can represent a library internal call and might thus not be contained in the partial call graphs.

Definition 4.3.6 (Capturing library-internal Lookup Changes) *In the presence of partial call graphs, we extend the definition of affectedTests to affectedTests$_{PCG}$ as follows. Let AM be the set of all add method changes. Let*

$$\begin{aligned} AM_{lib} \quad = \quad & \{a \in AM \text{ with resulting lookup changes} \\ & \mathbf{LC}_1 =< L, L.m() >, \mathbf{LC}_2 =< L, X.m() >\} \end{aligned}$$

be the set of add method changes resulting in newly reachable dispatch targets for library call-backs. Then for X ∈ Lib, Z ∈ App

$$\begin{aligned} affectedTests_{PCG} \quad = \quad & affectedTests \cup \{T \in \mathscr{T} \mid \exists A \in AM_{lib}, A = \mathtt{AM}_{\mathtt{Z.m()}}, \\ & \exists Z : X \trianglerighteq Z \wedge \nexists Y : X \rhd Y \trianglerighteq Z : defines(Y,m), \\ & \exists n \in Nodes(\mathscr{P}, T) : n = \mathtt{Z.} < \mathtt{init} >\} \end{aligned}$$

is the set of all tests potentially affected by these add method changes.

In Definition 4.3.6 $\mathtt{AM}_{\mathtt{Z.m()}}$ indicates the change resulting from the addition of method `Z.m()` and the predicate $defines(Y,m)$ is `true`, iff class Y has an own version of member m. With this definition we add all tests to the set of affected tests, where a newly added method potentially can be called by checking if a constructor for a type allowing access to this new method is called.

4.3.6 Effectiveness of Change Classifiers

We conducted two case studies with *JUnit/CIA* to find failure inducing changes in student programs and in *Daikon* [19], respectively. In each study, we first determined the actual failure inducing changes by manual examination of the code and then measured the effectiveness of each of the classifiers in identifying those changes. Here, effectiveness is measured by determining how much additional focus on failure inducing changes is provided by the change coloring, compared to the set of (uncolored) affecting changes reported by *Chianti*. Ideally, we would like to see the failure inducing changes for a test colored *Red*, and all other affecting changes *Green* or *Yellow*. Therefore we examined if change classification provides additional focus on failure inducing changes compared to the plain affecting changes from *Chianti*.

To assess the quality of our results we need those changes that are actually failure inducing.

Definition 4.3.7 (Failure Inducing Changes.) *Given a worsening test*[7]*, we can selectively undo a subset of its affecting changes, and observe whether or not the test outcome on the resulting intermediate program is worsening, with respect to the original version outcome. If the test is not worsening (on the intermediate version), then that subset contains* failure inducing *changes.*

For our case studies, we manually derived the failure inducing change sets for each application, making a best effort to obtain as small a subset as possible. Ideally, our classifiers should color exactly these changes *Red*.

Note that failure inducing change sets are not necessarily unique—in general there may be different independent or overlapping sets of changes which are failure inducing. Due to this observation a clear concept of "minimality" is hard to define. However in the experiments we report below we did not (consciously) experience any of these cases.

In the student programs study, one classifier was superior by correctly focusing programmer attention on the failure inducing changes in 47.5% of the 444 worsening tests with more

[7] PASS to FAIL, PASS to CRASH, or FAIL to CRASH

than 2 affecting changes; in addition this classifier provided misleading information in only 1 case (we will clarify this below).

In the *Daikon* study, we studied a pair of versions separated by a total of 6093 atomic changes in which two test failures occurred. Here, one of the tests was affected by 35 atomic changes, and the other by 34 atomic changes. In this study, a *different* classifier was very effective by focusing the programmer's attention on 4 of the 35 changes for the first test, and on 3 of the 34 changes for the second one. For both of these Daikon tests, the failure inducing changes were among the few changes that were colored *Red*.

While it seems contradictory that the case studies suggest that different classifiers should be preferred, this is not unexpected in an empirical study. Nevertheless, it is interesting to observe the different characteristics of the code analyzed in the studies to help explain the different outcomes. The student programs study is concerned with the initial development of an application, and is characterized by small differences between versions, and a mixture of improving and worsening tests. In the Daikon study, on the other hand, the application under consideration is more mature, the sets of changes between successive versions is much larger, only a few worsening tests occur, and no improving tests. Therefore, although we make recommendations for when each of the two preferred change classifiers should be used, it is clear that further investigation is needed, and we consider such investigations to be a fruitful topic for future work.

Evaluating Information Retrieval Techniques

Before discussing our experiments in detail, we first review important metrics used in the area of *information retrieval* to measure the performance of a specific retrieval technique. For a detailed overview of basic concepts of information retrieval technology refer to [61]. Our change classification technique tries to find the failure inducing changes among all affecting changes, thus the metrics of information retrieval theory are applicable. In this section relevant metrics are defined in terms of *desired and retrieved documents*; in our case the desired documents are the failure inducing changes, and the retrieved documents are those changes colored *Red*.

The two most important metrics in this context are *recall* and *precision*. Informally recall is the percentage of desired documents retrieved, while precision is the number of desired among all retrieved documents. Note that it is very easy to define a technique with a value of 1 for recall—simply return all documents. However such results are worthless without an equally high precision. On the other hand returning only obvious hits might result in a high precision, but a low recall. It is thus important that a certain information retrieval technique yields *both a high recall and a high precision*.

Definition 4.3.8 (Recall and Precision) *Let D be the set of all desired documents, R the set of all retrieved documents and S the search space, i.e. the set of all documents. Then:*

$$Recall = \frac{|D \cap R|}{|D|};\ and\ Precision = \frac{|D \cap R|}{|R|}.$$

Both Recall and Precision always result in values in $[0,1]$.

Note that recall and precision in a way are dual to *false positives* and *false negatives*, terms which are usually used in the field of program analysis. A false positive *fp* in this context can be defined as a retrieved, but not desired document, i.e. $fp \in (R - D)$. A false negative *fn* on the other hand is a desired, but not retrieved document, i.e. $fn \in (D - R)$. In order to normalize these values for our experiment described below we divided the absolute number of false negatives and false positives by the number of desired documents $|D|$ and retrieved documents $|R|$, respectively.

The metrics discussed above are adequate to measure the quality of an information retrieval technique for a *single* data point. Note however that—in order to evaluate the quality of a technique—a single data point is clearly not enough. To simply take averages of precision and recall values for several data points however is clearly not sufficient. Consider the following example. For a technique T and data point d_1 we calculate $recall = 1$ and $precision = 0.2$, for a second data point d_2 $recall = 0.2$ and $precision = 1$. I.e. the underlying technique has considerable deficiencies for both recall and precision depending on the data point. However, the average for *recall* and *precision* is 0.6—an acceptable value.

To avoid such problems, we used the average of the F_1 metric in our experiments.

Definition 4.3.9 (F_1 Metrics) *The F_1 metric is defined as the harmonic mean of precision and recall*

$$F_1 = \frac{2 * precision * recall}{precision + recall}.$$

As harmonic mean of Recall and Precision, F_1 always result in values in $[0,1]$.

The value of the F_1 metric is 0 if either recall or precision are 0, and 1 only if both precision and recall are 1, i.e. it combines both metrics and also weighs both metrics as equally important. For the above example we get F_1-values of 0.3 in both cases, and thus also as average—which is considerably lower than the averages for recall and precision we got before. Averaging F_1 values thus does not hide weaknesses of a technique as taking averages of recall and precision does.

A second possibility to give a detailed overview of the quality of a technique is to use *scatter plots*. These diagrams show recall on the x-axis and precision on the y-axis. Each data point then results in a single dot in the diagram. For our experiment however we observed several data points with equal recall/precision values. We thus also added the frequency of hits in the diagram by using a bubble chart. Optimally all data points should be in the upper right corner of the diagram, as this area corresponds to both high recall and precision values. In contrast to the averaged F_1-values a scatter plot thus gives a non-aggregated view of the different data points.

In our experiments we also observed data points where recall and precision were undefined. Recall is undefined for version pairs without worsening tests (then by definition the amount of desired documents $|D|$, i.e. in our case failure inducing changes, is 0). As our classifiers—besides the *simple* classifier—will never color any changes *Red* for these versions, we only considered version pairs with failure inducing changes in our evaluation. As we only intend our tool to be used in the context of worsening test, the excluded cases are not relevant for our technique.

We also encountered data points where precision was undefined, as our classifiers did not color any changes *Red*. We also excluded those cases, as here our classifiers, while not being helpful, do no harm either. In these cases, all changes were colored *Yellow*, i.e. programmer attention is not directed to the wrong changes. Note that the number of remaining data points depends on the classifier in this case, as change colors vary by classifier.

Case Study 1: Student Projects

Overall, we analyzed source code from 40 small student projects of an undergraduate programming course at the University of Passau. In this course, students implemented Dinic's Maximum Flow algorithm [14] using a predefined set of mandatory interfaces. The students were provided with a set of public *black box* tests that had to be successfully executed in order for students to pass the course. We also defined an additional *secret* test suite, whose existence was known to the students, although no details of these tests were available. Although the students had to agree that their code could be used for research purposes, they did not know that their data would be used to evaluate change classifiers. Course management was

Number of Version Pairs	
written by students	1175
that contain meaningful changes	556
with associated worsening tests	110
with identifiable failure inducing changes	98
where versions pairs differ by >1 change	61

Table 4.4: Selection of meaningful version pairs from the student data.

Classifier	
R_r/G_r	38
R_s/G_r	20
R_r/G_s	38
R_s/G_s	20
simple	20

Table 4.5: Versions with defined Precision.

provided using the web-based *Praktomat* system [65]. Students frequently submitted their solutions to Praktomat, which then automatically compiled them and ran the tests. Praktomat automatically saves all submitted versions in a database, so that these versions were available to us for this case study.

Analyzed code base. Some minor post-processing of the student code was needed to make it suitable for our experiments. As Praktomat uses black box testing, the public tests were coarse-grained regression tests for `DejaGNU`, an open-source black box regression-testing framework.[8] Our post-processing consisted of writing equivalent *JUnit* tests with assertions based on the mandatory interfaces, and adding fine-grained unit tests. In a few cases, several interpretations of the mandatory interfaces existed (e.g., node numbering in the graph could start at 0, or at 1), and we rewrote the tests for specific student solutions to uniformly use the same approach. We also commented out debugging output in a few cases for performance reasons. None of these changes affected the semantics of the submitted code in fundamental ways.

On average, each of the final, graded solutions consisted of 950 LoC of commented Java source code. We analyzed a total of 1175 version pairs written by 40 students. Of these 1175 version pairs, 556 contained meaningful changes,[9] and 110 of these 556 version pairs had associated worsening tests. For 98 of these 110 version pairs, we could manually identify the *failure inducing changes*. In the remaining 12 cases we were unable to determine the failure inducing changes due to the size of the edit or non-deterministic test behavior. Since we are interested in techniques for automatically determining failure inducing changes, we need version pairs that differ by more than one change (otherwise, the reason for the failure is obvious). Eliminating the version pairs that differ by one change resulted in a final set of 61 version pairs (out of the 98) that we used as the basis for evaluating the 5 change classifiers presented in Section 4.3.3. Our classifiers did not color changes *Red* in all of the 61 versions. As discussed in Section 4.3.6, we thus removed cases where no change was colored *Red* from presentation in the scatter plots as precision in these cases is undefined. The process of selecting version pairs is illustrated by Table 4.4; Table 4.5 gives an overview of the number of finally remaining data points for the different classifiers.

Per-Version-Pair Evaluation. The 61 version pairs contained a total of 401 atomic changes. Table 4.6 shows how the different classifiers associate colors with these changes. From left to right, the columns of the table indicate the total number of changes classified as *Red*, *Yellow*, and *Green*, respectively. For example, the R_r/G_r classifier finds 138 *Red*, 126 *Yellow* and 137 *Green* changes. Table 4.7 in contrast shows the number of change colors if we remove those versions where our classifiers do not color any change *Red*. The additional column labeled "#DP" indicates the number of relevant data points in these cases.

[8] See `www.gnu.org/software/dejagnu/`.

[9] Our analysis considers two versions the same if they differ only in layout or comments, or are completely equal. The relatively high number of versions without changes is due to coding style requirements for the course, which were addressed by the students late in their implementations and the use of Praktomat as a testing and development system.

Classifier	*#Red*	*#Yellow*	*#Green*
R_r/G_r	138	126	137
R_s/G_r	77	187	137
R_r/G_s	138	200	63
R_s/G_s	77	261	63
simple	119	238	44

Table 4.6: Coloring of changes according to the 5 classifiers (cumulative statistics over 61 version pairs).

Classifier	#DPs	*#Red*	*#Yellow*	*#Green*
R_r/G_r	39	138	45	40
R_s/G_r	22	77	53	12
R_r/G_s	39	138	52	33
R_s/G_s	22	77	60	5
simple	26	119	48	3

Table 4.7: Coloring of changes according to the 5 classifiers (cumulative statistics over version pairs with defined precision).

Classifier	#DP	*recall Red*	*prec. Red*	F_1 *Red*
R_r/*	39	91.5	63.3	67.5
R_s/*	22	78.0	66.0	64.5
simple	26	66.0	43.8	48.9

Table 4.8: Recall, precision, and F_1-values for R_r, R_s and *simple* classifiers averaged over version pairs with defined precision.

To determine classifier quality, we manually identified failure inducing changes for each of the 61 version pairs. Then, we calculated *recall* and *precision* for each classifier. For *Red* changes, *recall* is the percentage of failure inducing changes colored *Red*, and *precision* is the percentage of actual failure inducing *Red* changes among all *Red* changes.

The choice of the criterion to classify *Green* changes has no effect on recall and precision for *Red*; thus we will discuss classifier results for *Red* changes independently of the *Green* criterion, and *vice versa*. Table 4.8 shows, on average over all version pairs where precision is defined (i.e. where our classifiers color at least one change *Red*), the average recall, precision, and F_1-values for the R_r/*, R_s/*, and *simple* classifiers. For an overview of the non-aggregated data, the reader is referred to [57]. Note that we also have data points where both precision and recall are 0, i.e. where our technique unfortunately provides misleading information. While there are few of these cases (1 for the R_r/*, 2 for the R_s/*, and 6 for the *simple* classifiers), this results in an undefined F_1-value. However, as we did not want to artificially increase the averages of F_1-values due to these cases, we added them with value 0 when calculating the averages, as 0 indicates the worst possible result for F_1.

It is easy to see that the *simple* classifier can be dismissed, because averages of precision, recall and—most importantly—also of F_1-values of both the R_r/* and the R_s/* classifiers are considerably higher. For the two remaining classifiers, it is not obvious *a priori* whether the R_s/* or R_r/* classifiers should be preferred. Ideally we would like to classify *all* failure inducing changes as *Red*, while not coloring any non-failure inducing changes *Red*. The recall and precision averages in Table 4.8 give inconclusive information, the R_s/* classifiers yield a better precision (66.0% vs. 63.3%) and thus, a lower false positive rate, but the R_r/* classifiers yield a better recall (91.5% vs. 78.0%) and thus a lower false negative rate. In this case we see that—beside hiding the true quality of a technique—averaging recall and precision is also less useful, as they do not give an accurate overview of the classifier quality. The F_1-value however allows such a decision: the R_r/* classifiers are superior here (67.5% vs. 64.5%) and are thus preferable.

For *Green* changes, recall is the percentage of all non-failure inducing changes colored *Green* and precision is the percentage of actual non-failure inducing changes among all *Green* changes. In selecting the preferred *Green* criterion, we only consider the F_1-values for the *Green* changes. For *Green* changes the precision will always be 100% since *Green* changes never affect worsening tests for our development-based classifiers. The */G_r classi-

fiers produce a F_1-value of 21.1% versus 17.6% for the */G_s classifiers (averaged over all 61 data points), meaning that the former are more successful at classifying non-failure inducing changes as *Green*. Consequently, the */G_r classifiers are clearly preferable to the */G_s ones. In summary, for this case study, it is clear that the R_r/G_r classifier produces the best results.

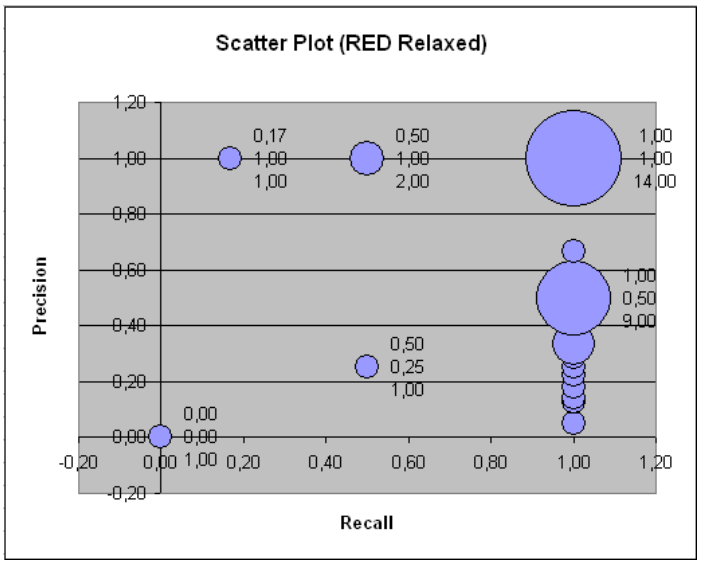


Figure 4.7: Scatter Plot for the R_r/* Classifiers, based on 39 data points.

We finally discuss scatter plots for the different classifiers. We start with the R_r/* classifiers. Their scatter plot is shown in Figure 4.7. The x-axis shows the recall, the y-axis the precision value. The size of the bubble indicates how many data points exhibit a certain recall / precision combination. Optimally all entries should be in the upper right corner for such a diagram.

Note that the R_r/* classifiers perform pretty well—we only have 5 of 39 data points where recall is less than 100% (i.e. no all failure inducing changes have been colored *Red*). Note that one of these data points is located at the origin of the coordinate system. This is the data point where both recall and precision are 0, i.e. where our techniques actively points the programmer in the wrong direction. For the other 34 data points, recall is 100%, but precision varies, from 5% to 100%, with a concentration at 50% (9 data points) and 100% (14 data points). For all 34 of these cases, our classification technique helps by coloring some non-failure inducing changes *Yellow*, so providing additional focus on the failure inducing changes among the *Red* changes.

For the R_s/* classifiers we only had 22 data points to compare. Of these, 2 data points are located at the origin, indicating misleading information; in total 6 data points have a recall of less than 100%, i.e. in these cases we miss failure inducing changes. For the remaining 14 cases again all failure inducing changes are colored *Red* (i.e. recall is 100%), with precision varying from 5% to 100%, again with a concentration at 50% (5 data points) and 100% (8 data points). Note that, while the distribution of the data points is qualitatively similar to the R_r/* classifiers, the number of meaningful data points is considerably lower (22 vs. 39), and we have 2 (vs. a single) data points where this classifier provided misleading information. Thus the R_r/* classifiers are superior compared to the R_s/* classifiers for this case study.

The *simple* classifier finally—while providing meaningful data for 26 data points—can be clearly dismissed when considering its scatter plot (Figure 4.9). Here, 6 data points provide misleading information and we have considerably more data points where recall is less than 100%, i.e. the *simple* classifier misses most failure inducing changes. While still a majority

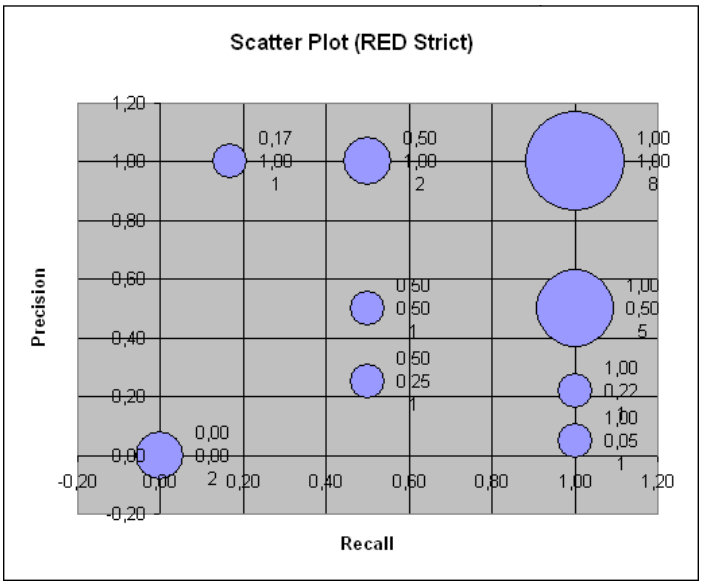


Figure 4.8: Scatter Plot for the R_s/* Classifiers, based on 22 data points.

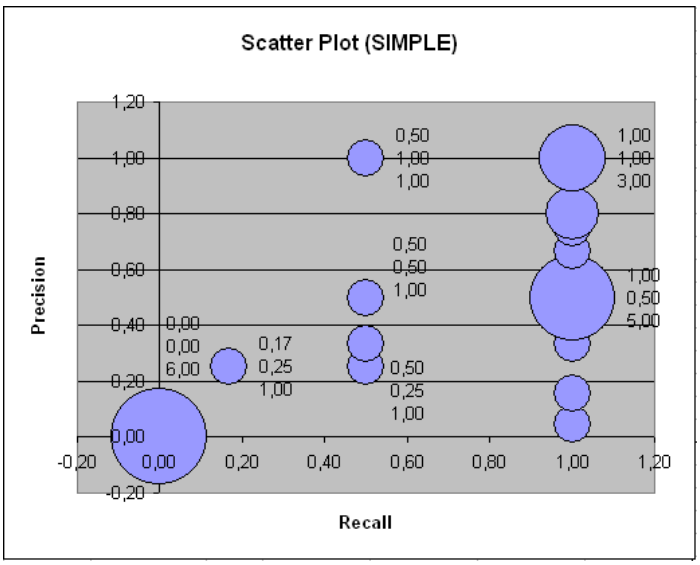


Figure 4.9: Scatter Plot for the *simple* Classifier, based on 26 data points.

of the 26 data points has a recall of 100%, we only have 3 (!) data points where precision is 100% as well.

Per-Test Evaluation. As a final step in this case study, we measured how often change classification helps the programmer find the failure inducing changes for a given test failure. For this "per test" view, we examine 444 worsening tests in the 61 version pairs under consideration that have 2 or more affecting changes. The baseline for comparison is the uncolored set of affecting changes as calculated by *Chianti*. For 211 of the 444 worsening tests, we calculated that the R_r/* classifiers colored all the failure inducing changes *Red*, and some of the other affecting changes *Yellow*. This means that the R_r/* classifiers were successful at focusing programmer attention in 47.5% (211/444) of the tests. This property was evidenced in

only 25.5% (113/444) of the worsening tests using the R_s/* classifiers and 15.1% (67/444) for the *simple* classifier. In addition, the R_r/* classifiers colored failure inducing changes *Yellow* when some non-failure inducing changes were *Red* in only a single cae. In such cases, the coloring would be *misleading*, because it would direct the programmer to look at changes that are not failure inducing, while missing some of the failure inducing ones. All other classifiers resulted im more data points with such misleading information.

Conclusions. For this case study, the R_r/* classifiers are superior compared to the other classifiers as they resulted in the best average F_1-value and only a single data point with misleading information. Additionally, the R_r/* classifiers were able to color changes *Red* in 39 of 61 cases (vs. only 22 and 26 cases for the R_s/* and *simple* classifier, respectively).

Case Study 2: Daikon

Daikon [19] is a system for discovering likely invariants in software systems using dynamic analysis. We extracted several versions of Daikon from the CVS repository, but (unfortunately for our purpose) could not find any worsening unit tests. This illustrates a common problem in obtaining evaluation data for our method. In general changes are only committed if all tests succeed, (i.e., there are no worsening tests in repositories). However, we noticed that several unit tests changed between the Daikon versions *Daikon/2002-11-11* and *Daikon/2002-11-19*, and reusing the old tests with the edited version produced 2 test failures. In the experiments discussed below, we treat these test failures as worsening tests. For the Daikon version pair under consideration, a total of 61 tests were defined, of which 40 were affected by the edit (there were also 7 new tests and 3 deleted tests). The two versions differed significantly, as a total of 6093 atomic changes were reported by *Chianti*.

The first test, `testXor`, was affected by 35 atomic changes. Manual inspection of the code revealed that two **CM** changes to methods `daikon.diff.XorVisitor.shouldAddInv1()` and `daikon.diff.XorVisitor.shouldAddInv2()` were responsible for the test's failure. The R_r/* classifiers failed to focus on these changes since they classified all 35 affecting changes as *Red*, because the they affected no improving tests. Both R_s/* classifiers correctly identified the 2 failure inducing changes as *Red*, as well as 2 of 33 remaining changes, with the rest classified as *Yellow*. In other words, the R_s/* classifiers were very successful at correctly providing focus on only 4 out of 33 affecting changes, including the appropriate ones.

The second test, `testMinus`, produced a similar result. This test was affected by 34 changes, and we manually identified the failure inducing change to be a **CM** change to method `daikon.diff.Diff.shouldAdd()`. Again, the R_r/* classifiers were not useful because they classified all 34 changes as *Red*. The R_s/* classifiers were very effective by classifying the failure inducing change as *Red*, only two other changes as *Red*, and the 31 remaining changes as *Yellow*. Thus, we provide focus on only 3 of 34 changes, including the appropriate one.

The tests for these two Daikon versions were either worsening (i.e., PASS to FAIL) or successful in both versions (i.e., PASS to PASS). Because of this restricted set of test outcomes, the *simple* classifier and the R_s/* classifiers produce the same coloring of the changes. If there had been tests that failed in both versions (i.e., FAIL to FAIL), then the *simple* classifier would have been less successful at focusing the programmer's attention on failure inducing changes than the R_s/* classifier. These limited test outcomes seem coincidental, and may be the result of our constructed tests; therefore, we prefer the R_s/* classifiers for the Daikon data.

Of the 6093 changes separating the two Daikon versions, 5715 were classified as *Gray* due to the low coverage of the Daikon unit test suite. Of the remaining 378 changes, 338 were *Green*, 33 *Yellow*, and only 7 *Red* using the R_s/G_r classifier. In this case study, our approach reduced the number of changes to be examined from 6093 to 35 (or 34) affecting changes for each worsening test, and then further reduced the number to 4 (or 3) using the *Red* changes obtained from the R_s/* classifiers.

Conclusions. The R_s/* classifiers outperformed the R_r/* classifiers on the Daikon ver-

sion pair, by focusing programmer attention on the failure inducing change(s) within the *Red* changes. Thus, this study selects the R_s/G_r classifier as best.

Assessment

With our current untuned research implementation of *JUnit/CIA*, constructing dynamic call graphs slows down the execution of tests by more than a factor of 10. For the student project case study this was insignificant, but for the Daikon case study, the timings were unacceptable for interactive use. In our experiments with Daikon, constructing the dynamic call graphs for all unit tests for a given version takes about 4 minutes on average, and computing and classifying the atomic changes takes less than 2 minutes. To address this performance issue, we envision a scenario where programmers run their tests normally, until they encounter a worsening test. Only then do they rerun the tests using *JUnit/CIA* to perform change classification. In addition, we expect that dynamic call graph construction can be made significantly more efficient.

The main outcome of our two case studies is a positive demonstration that change classification may focus programmer attention on parts of an edit that may be the root cause of unexpected worsening test behavior. While more extensive empirical investigation of larger programs is necessary to fully validate this claim, the success of change classification in these studies is undeniable.

As is common, the case studies also raise unexpected questions. For example, the two case studies select contradictory choices for the "best" classifier, but this can be explained by considering the behavior of the associated test suites. In defining our classifiers, we have assumed that the parts of the program executed by different tests will overlap, and therefore, some changes will affect more than one test. If this assumption is violated, then all the changes affecting a worsening test will be colored *Red*, offering no focus on the failure inducing changes. Thus, the success of classification depends on some properties of the tests used.

The student projects exhibit a mixture of improving and worsening tests, and the $R_r/*$ classifiers work best here. On the other hand, in the Daikon study, there are only a few worsening tests and no improving tests. The $R_r/*$ classifiers are hindered by the lack of improving tests, which prevents any affecting changes of a worsening test from being colored *Yellow*. Since the $R_s/*$ classifiers do not color changes *Red* that affect tests with the same outcome in both the original and edited program, they are able to focus programmer attention on a *subset* of the changes affecting the worsening tests. Thus the $R_s/*$ classifiers perform better on the Daikon case study.

Given these empirical results, we suggest that programmers use the R_r/G_r classifier during development when both improving and worsening tests exist. If only worsening and same-outcome tests occur, then the R_s/G_r classifier seems to be the better choice. It is possible that through experience, development organizations will be able to select the appropriate classifier for their projects.

Additional questions raised by our investigations include the following: Does the choice of classifier depend on other factors we have not yet considered, including programmer experience level, software maturity (i.e., in active development versus maintenance), etc.? Are there properties of the test suite which can suggest the appropriate classifier to use?

One of the most interesting questions is to investigate other classifiers. Our current classifiers only operate on discret value sets, both for affected tests and also for assigned colors. One option might now be to assign a color on a continuous scale between *Green* and *Red* to changes based on the percentage of worsening tests among the affected tests. While this is still a rather simple approach, more sophisticated statistical classification techniques exist which might be worth to examine in this context.

Another option might be to explore machine learning based approaches, for example neural networks, which are known to provide useful adaptive classification properties. This

would have the additional advantage that each developer is provided with a classifier which will over time optimally adapt itself to his development and test strategy, however at the cost that the classifier needs programmer feedback.

4.3.7 Related Work

For related work on change impact analysis in general refer to Section 4.1.4. This section presents related work on failure detection.

Delta Debugging. In the work on *delta debugging*, the reason for a program failure is identified as a set of differences between versions [64], inputs [67], thread schedules [10], or program states [66, 11] that distinguish a succeeding program execution from a failing one. A set of failure inducing differences is determined by repeatedly applying different subsets of the changes to the original program, and observing the outcome of executing the resulting intermediate programs. By examining the outcome of each execution (*pass, fail*, or *inconsistent*), the set of failure inducing changes is narrowed down using efficient binary-search techniques.

Our work and delta debugging are different approaches for identifying failure inducing changes, each with its strengths and weaknesses. Delta debugging determines whether or not a change is failure inducing by observing the effect of its presence or absence in two program executions. Executing intermediate program versions helps narrow down the reason for a program failure but, in the worst case, a number of executions proportional to the number of changes is required. In contrast, our approach identifies reasons for failures using the results of distinct tests that execute different subsets of the changes, and requires a suite of tests with this property. The two approaches may complement each other. In principle, the use of a rich model of changes with interdependences could improve the efficiency of delta debugging by reducing the number of intermediate programs that are constructed/executed. Conversely, our method could be made more precise by executing tests on intermediate program versions, and taking their results into account.

Comparing Dynamic Data Of Different Executions. Several debugging approaches rely on comparing dynamic information associated with succeeding and failing runs. Reps et al. [48] compare path profiles from different executions in order to expose incorrect Year 2000 date-related computations that give rise to the execution of different paths. Harrold et al. [28] evaluate the effectiveness of comparing path profiles (and other run-time metrics) for distinguishing successful executions from failing ones. They found a strong correlation between differences in path profiles and different execution behavior; similar findings held for their other metrics. Jones et al. [30, 29] use the colors red, yellow, and green to visualize the statements executed by failing tests only, by both succeeding and failing tests, and by passing tests only, respectively. They found this *discrete* visualization to be "not very informative, as most of the program is yellow" and also propose a *continuous* visualization where a gradual scale of color and brightness reflects both the absolute number of tests, and the relative percentages of passing and failing tests that execute a given statement. Our work differs from their discrete approach because we visualize the correlation between *changes* and their affected tests, whereas Jones et al. visualize the correlation of *statements* with test results. Our approach is likely to be more useful for locating failure inducing changes because the number of executed changes is likely to be far smaller than the number of executed statements, and because the execution of different statements by a failing test may be due to a change in a completely different part of the program. Ruthruff et al. [50] also use a continuous color scale to indicate the contribution of cells in a spreadsheet to incorrect values. In this work, the user indicates whether or not computed values are correct, and dependences between cells are used to compute the likelihood that (the formula in) a given cell contributes to an incorrect value.

Renieris and Reiss [47] use tracing data from one faulty and several successful runs to detect failures in C programs. They build a model from the traces, calculate a difference between the models of the faulty and the successful runs and map this difference back to

source code artifacts, which finally forms the report. Dallmeier et al. [12] present a technique for localizing errors by comparing sequences of method calls in passing and failing runs of a program. Their experiments indicate that comparing method call sequences is a better defect indicator than a simple coverage-based metric, such as the one by Jones et al. [30], and that comparing sequences of method calls *on the same object* is an even better predictor.

Statistical Techniques. Some researchers use statistics to calculate the likelihood that a specific predicate is related to a fault. Liblit et al. [36, 37] present statistical analyses in which information is gathered about the number of times that certain predicates are executed by deployed applications, in order to detect predicates whose outcome correlates with a crash. A low sampling frequency is used to ensure low run-time overhead, so a large number of samples is needed to obtain meaningful data. A number of strategies is presented that allow one to quickly rule out certain predicates as being related to failures. Liu et al. [39] propose a statistical model-based approach to localize bugs and define the "evaluation bias" of a predicate, which measures the probability of a predicate being "true" in one execution. Then, the evaluation patterns in correct and incorrect runs are compared to identify those predicates that are likely to be bug-relevant. A comparison of their model with Liblit's method [37] and Cleve and Zeller's method [11] shows that they can localize more bugs (68/130 in the Siemens suite) in certain contexts. While we do not use statistical methods to classify changes yet, investigating new classifiers based on such methods might be a fruitful area for future work.

Fault Localization Techniques. A program slice [62, 59] w.r.t. an incorrect value contains all statements that may have contributed to that value, and will generally include the statement(s) that contain the error. Slices may become very large, and techniques such as *dicing* [40] have been proposed, where a slice w.r.t. an erroneous value is intersected with a slice w.r.t. a correct value. DeMillo et. al. [13] define a *critical slice* w.r.t. a failing test t to contain all "critical" statements that, when omitted, cause program execution to reach a designated failure statement with different values for referenced variables. Gupta et al. [26] propose an approach that integrates delta debugging with program slicing to narrow down the search for faulty code. First, delta debugging is used to identify a minimal failure inducing input, and a forward dynamic slice is computed from this input. Then, they obtain a backward dynamic slice with respect to the erroneous output, and the intersection of these two slices may potentially contain the faulty code.

Our approach and program slicing can both be used for finding faults, but there are two significant differences. Slicing is a fine-grained analysis at the statement level that can be used to inspect a failing program to help locate the cause of the failure. Our work focuses on failures that are due to the application of a set of changes, and our analysis is at the method level.

Continuous Testing and Test Factoring. Saff and Ernst present two techniques for identifying test failures early, when reasons for these failures are easy to identify. In *continuous testing* [52, 54], tests are run whenever the CPU is idle. *Test factoring* [53] automatically derives fast unit tests from slow system-wide tests using dynamic analysis. Change classification complements these techniques by reducing the amount of time needed to fix bugs.

4.3.8 Conclusions and Future Work

There are three main contributions of the research presented in this section: First, we presented an approach for change classification that helps programmers identify the changes responsible for test failures. As part of this approach, we proposed several change classifiers that associate the colors *Red*, *Yellow*, or *Green* with changes, according to the likelihood that they were responsible for test failures. Second, we implemented these change classification techniques in *JUnit/CIA*, an extension of the *JUnit* component of Eclipse. Third, we conducted two case studies in which we investigated whether or not change classification can be a useful tool for focusing the programmer's attention on failure inducing changes.

Furthermore, in response to the 3 research questions posed in Section 4.3.1, we conclude that:

- Change classification can successfully distinguish failure inducing changes from other changes. Specifically, in the student programs case study, programmer attention was focused on failure inducing changes in 47.5% of the worsening tests. In the Daikon case study, programmer attention was focused very effectively on a small superset of the failure inducing changes.
- There is no single change classifier that always works best. In the student programs case study, R_r/G_r is the classifier of choice. However, in the *Daikon* case study, R_r/G_r failed to provide any focus on the failure inducing changes, and the R_s/G_r classifier was highly effective.
- Based on these results, and on the characteristics of the systems being analyzed we suggest that programmers use the R_r/G_r classifier during initial development, when small differences between versions exist along with a mixture of improving and worsening tests. If versions differ more significantly, and if only worsening tests occur, then the R_s/G_r classifier seems to be the better choice.

While these results are promising, it is clear that more experimentation and/or a user study are needed for a conclusive validation of the approach. Other topics for future work include an in-depth analysis of factors we have not considered so far such as programmer experience level and properties of test suites. We also plan to develop other classifiers that, for example, take into account the frequency that a change affects a worsening test.

Acknowledgements. We are grateful to Martin Robillard, Gregg Rothermel, Joe Ruthruff, and Andreas Zeller for their constructive comments on this research and previous publications [56]. We also appreciate the excellent help provided by Ophelia Chesley in the experimentation and the advice provided by David Madigan and Vadakkedathu T. Rajan on the analysis of the student programmer case study. This research was supported by NSF grant CCR-0204410 and, in part, by IBM Research.

4.4 Supporting Early Release of Changes

Before investigating the use of change classification in the context of aspect-oriented programming we will first examine a second application of change impact analysis in the context of plain Java: the use of change classification to allow early release of changes to a reporistory in order to support team-developed projects.

4.4.1 Early Release of Changes—Why?

The size of large software projects today is measured in "man years", describing the code a software engineer can produce in a year. Clearly a single developer cannot create such a system in an acceptable amount of time, thus today software development is a highly cooperative process—teamwork is important.

However, while reducing the time to market, development in a team results in coordination problems, one of them is parallel and conflicting edits of a code base. Although it is customary to assign clear responsibilities for different modules, in general it cannot be avoided that team members are affected by changing code. Occasionally developers even change files in parallel, and consequently have to integrate changes later when all changes should be released to a repository. As this effort is higher when more changes have to be integrated, it is desirable to commit changes frequently to keep the amount of changes to integrate small.

Today software development in general is backed up by a test suite available for programmers to check their progress. While a test suite can be essential for program correctness, there is also a problem connected to it. In current development practice, it is customary to release changes to a version control repository only when all tests succeed. As a result, the intervals between *commits* of changes to a repository can be long, and significant differences may exist between successive versions. In the presence of multiple developers, the existence of significant changes between successive versions may complicate the task of integrating these changes. Determining *committable changes* that can be exposed safely to others by early release of changes to a repository enables developers to reduce the amount of time spent subsequently on change integration by allowing early anticipation of these changes by others.

To determine subsets of changes that can be committed safely in the presence of failing tests, we need to resolve when a set of changes should be considered committable. This is basically a management question, and one possible commit policy to answer it is the following.

Don't commit changes in the presence of failing tests.

However, this policy is often unnecessarily restrictive. Consider a situation where a test T fails in both the original and the edited version of the program. Then, the failure of T in the edited program may be caused by the new changes, or it may be due to the same reason that caused T's failure in the original version.[10] It is often sufficient to ensure that no *additional* test failures occur due to committed changes, which corresponds to the following less restrictive commit policy.

Don't commit changes if they degrade any test result.

This section describes how *Chianti* can be used to break up an edit into a set of *atomic changes* and calculate a subset of these changes which can be committed without degrading any test result. Similarly to change classification discussed before, we use *Chianti* to compare two subsequent versions $\mathscr{P}$ and $\mathscr{P}'$ of a software system and derive a set of coarse grained atomic changes $\mathscr{A}$. Using call graphs, each test T of a test suite $\mathscr{T}$ is then correlated to its *affecting changes AC*(T). We the determine a committable subset of $\mathscr{A}$ using this change-test correlation together with test results in both versions as input.

The contribution of this work is to provide a configurable algorithm with which the user can trade accuracy versus runtime on demand to calculate a set of committable changes. Releasing those changes is guaranteed not to degrade the result of any existing unit test.

4.4.2 Early Release of Changes—By Example

Figure 4.10(a) shows two versions of a small example program. We use a new example in this section instead of reusing the previous example to better illustrate the process. Again, the original version of the program consists of all program fragments *except* for those shown underlined; the edited version is obtained by adding all the underlined code fragments. Associated with the program are five *JUnit* tests, `test1` to `test5`, as shown in Figure 4.10(b). We assume that the tests of Figure 4.10(b) will be used with both the original and edited versions of the program. The observant reader may verify that in the original version all tests pass, and in the edited version `test1` is the only failing test.

Atomic Changes. Early release of changes again builds on the change impact analysis of *Chianti* [46]. In the context of committable changes syntactic dependences—and also *semantic dependences*—between atomic changes are especially important. While *Chianti* calculates syntactic dependences, we will introduce an algorithm to approximate effects of semantic dependences as well.

[10] Determining why T fails in the edited program is beyond the scope of the analysis in this work.

```
1  public class A {
2    int y = 0;[1]
3    int zip(int x) {
4      x = zap(x);[2]
5      return x + 2;
6    }
7    int zap(int x) {
8      return 2 * x;
9    }[3,4]
10   int foo(int x) {
11     return 2 *[5] x;
12   }
13   int bar(int x) {
14     return x / 2[6];
15   }
16   int wiff(int x) {
17     System.out.println(x);[7]
18     return x*x;
19   }
20   void waff(int x) { y = x;[8] }
21 }
22 public class B extends A {
23   int foo(int x) {
24     return x;
25   }[9,10,11]
26 }
```

(a)

```
1  public class Tests
2  extends TestCase {
3
4    public void test1() {
5      A a = new A(); a.bar(3);
6      assertEquals(5, a.zip(3));
7    }
8    public void test2() {
9      A a = new A();
10     int x = a.bar(2);
11     assertEquals(2, a.foo(x));
12   }
13   public void test3() {
14     A a = new A();
15     a.foo(42);
16     assertEquals(25, a.wiff(5));
17   }
18   public void test4() {
19     A a = new A(); a.bar(5);
20     a.waff(5);
21   }
22   public void test5() {
23     A b = new B();
24     assertEquals(5, b.foo(5));
25   }
26 }
```

(b)

Figure 4.10: **(a)** Original and edited version of example program. The original program consists of all program fragments *except* those shown underlined. The edited program is obtained by adding all underlined code fragments. Each change is labeled with the numbers of the corresponding atomic changes. **(b)** Tests associated with (both versions of) the example program.

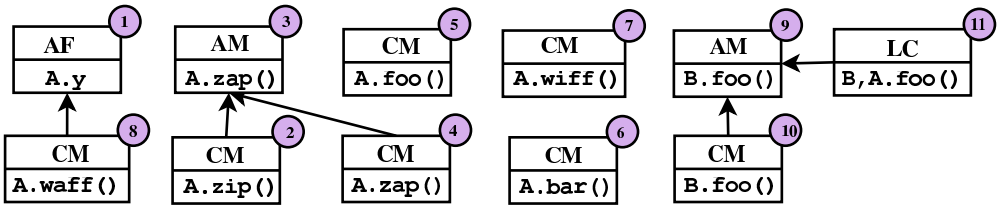


Figure 4.11: Atomic changes inferred from the two versions of the program.

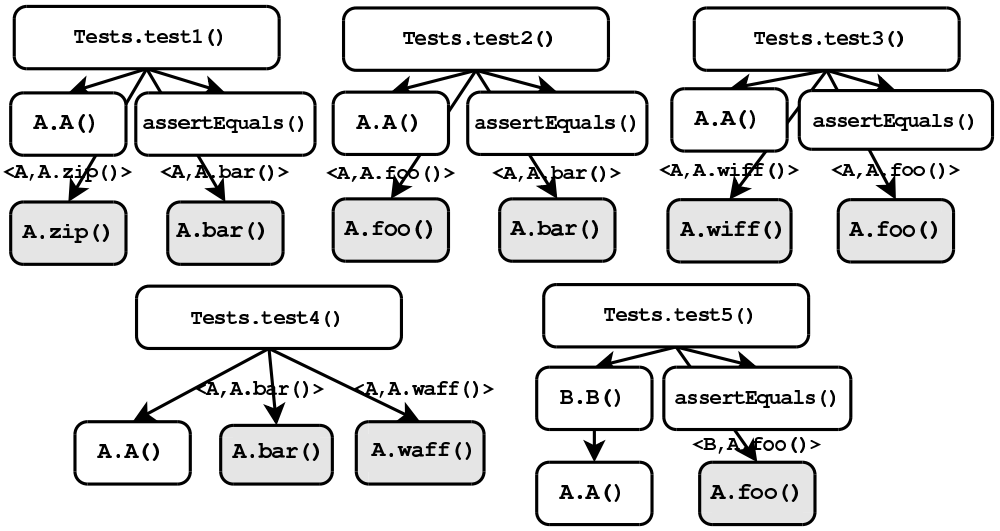


Figure 4.12: Call graphs for the original version of the program.

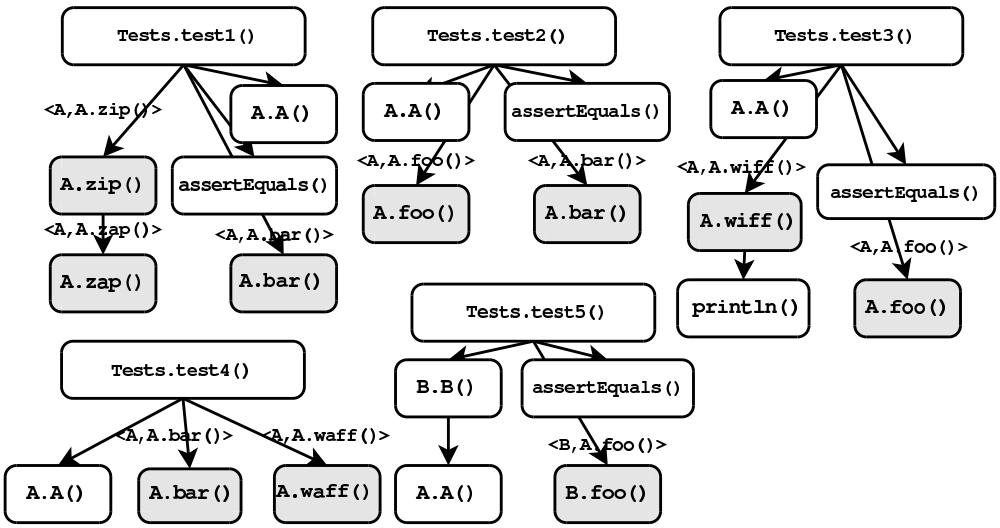


Figure 4.13: Call graphs for the edited version of the program.

Example 4.4.1 (Atomic Changes, Syntactic Dependences) *Figure 4.11 shows the atomic changes that define the two versions of the example program, numbered 1 through 11 for convenience. (Here, we reused the same notation as described in Figure 4.2 on 79.) Consider, for example, the addition of the assignment* `y = x` *in method* `A.waff()`*. This source code change corresponds to atomic change 8 in Figure 4.11. Adding this assignment will lead to a syntactically invalid program unless field* `A.y` *is also added. Therefore, atomic change 8 is dependent on atomic change 1, an* **AF** *change for field* `A.y`*.*

While respecting syntactic dependences when adding a set of changes to a system guarantees that the resulting system is compilable, this is not enough if we want the resulting system to comply with the commit policy that no test result should be degraded.

Example 4.4.2 (Semantic Dependence) *Test* `test2` *demonstrates semantic dependences. The test passes if neither change 5 nor change 6 (changes to* `foo` *and* `bar`*) is applied or if both changes are applied. Applying only one change will result in a failing tests, although there is no syntactic dependence among changes 5 and 6.*

Recall that in some cases, a single source code change is decomposed into several atomic changes. For example, the addition of `A.zap()` produces atomic changes 3 and 4, where the former models the addition of an empty method `A.zap()`, and the latter the addition of its method body. Observe that atomic change 4 is dependent on atomic change 3, reflecting the fact that a method must exist before its body can be added.

Determining Affected Tests. We use *Chianti* in order to identify those tests that are affected by a given atomic change. Figure 4.12 shows the call graphs for the tests of Figure 4.10(b) in the original program. We again labeled edges corresponding to dynamic dispatch with a pair $< RT, M >$, where RT is the run-time type of the receiver object, and M is the method referenced at the call site. In Figure 4.12 all five tests are affected, because they each execute at least one method corresponding to a **CM** change (shaded nodes in the figure). For example, the call graphs for `test1` and `test2` contain the node corresponding to changed method `A.bar()` (change 6).

Determining Affecting Changes. *Chianti* is also used to compute the set of changes affecting a given test. Figure 4.13 shows the test call graphs (created in the context of the *edited* version) necessary to calculate these affecting changes.

Example 4.4.3 (Affecting Changes) *The call graph for* `test1` *shown in Figure 4.13 contains nodes corresponding to methods* `A.zip()`*,* `A.zap()` *and* `A.bar()`*. These nodes correspond to atomic changes 2, 3, 4 and 6 in Figure 4.11, respectively. Similarly we determine that 5 and 6 are affecting changes for* `test2`*, that 5 and 7 are affecting changes for* `test3`*, that 6, 8 and 1 as its prerequisite are affecting changes for* `test4` *and that changes 9, 10 and 11 are affecting changes for* `test5`*.*

Committable Changes. Assume that after finishing the edit, the programmer now wants to commit his changes to a repository. However, as `test1`'s outcome has degraded, the commit policy does not allow to simply commit all changes. For this simple example it might be easy to fix the problem (i.e. make `test1` PASS again), but in general, especially for unexpected test failures, this can be considerably more problematic as it might involve invasive code changes. A release of the changes thus has to be postponed.

Employing our techniques, it is however possible to automatically derive a subset of *committable changes* which can be released to a repository safely without breaking any test. Note that such a subset in general is not unique, and that our techniques may also fail to derive an optimal solution (i.e. some committable changes might be missed); however our approach yields conservative results. For the given example our algorithm will determine—depending on the runtime the user is willing to invest—that minimally changes 9, 10 and 11 and up to changes 1 and 7 to 11 are committable for this example.

4.4.3 Tests, Changes and Dependences

If we follow the policy "Don't commit any change degrading test results", we want to find a program version in between the original and the edited version where a maximal subset of all atomic changes $\mathscr{A}$ is applied, but where no degrading test result (compared to the original version) can be observed. We will call programs defined by the different applied subsets of $\mathscr{A}$ *(program) configurations* in the following.

For a given test T, we will reuse the notation $R_{\mathscr{P}}(T)$ to represent the result of test T in program $\mathscr{P}$, where $R_{\mathscr{P}}(T) \in \mathscr{R}$, the set of all possible test results (see Definition 4.3.1). Definition 4.4.1 below uses this notation to classify tests as worsening and non-worsening (similar to Definition 4.3.2).

Definition 4.4.1 (Test Classification) *Let $\mathscr{T}$ be the set of all tests. Then the sets $WT_{\mathscr{P},\mathscr{P}'}$ and $\overline{WT}_{\mathscr{P},\mathscr{P}'}$ of worsening tests and non-worsening tests (with respect to programs $\mathscr{P}$ and $\mathscr{P}'$), respectively, are defined as follows:*

$$\begin{aligned} WT_{\mathscr{P},\mathscr{P}'} &= \{\, T \in \mathscr{T} \,|\, R_{\mathscr{P}}(T) > R_{\mathscr{P}'}(T) \,\} \\ \overline{WT}_{\mathscr{P},\mathscr{P}'} &= \{\, T \in \mathscr{T} \,|\, R_{\mathscr{P}'}(T) \leq R_{\mathscr{P}}(T) \,\} = \mathscr{T} - WT_{\mathscr{P},\mathscr{P}'} \end{aligned}$$

Using the set of atomic changes $\mathscr{A}$ and syntactical change dependences we can calculate a first approximation of committable atomic changes. Note however that this approximation still ignores semantic dependences, i.e. the resulting set in general is too large. However we use this set as a starting point for our further analysis.

Definition 4.4.2 (Potentially Committable Changes) *The set of* potentially committable changes *is defined as follows:*

$$\mathscr{A}_{PotComm} = \{A \,|\, A \notin Worsening, \forall A' : A' \preceq^* A : A' \in \mathscr{A}_{PotComm}\},$$

where $\preceq^*$ indicates the transitive closure for syntactic dependences and Worsening is defined as in Definition 4.3.3.

Definition 4.4.2 states that a change A is committable if: (i) A does not affect any worsening tests and (ii) all of A's prerequisite changes are committable.

If an intermediate version is created from the original program using the configuration given by $\mathscr{A}_{PotComm}$, it is guaranteed that the resulting program is compilable. However, committing only a subset of all changes can result in different program semantics compared to both the original and the edited program, i.e. for two atomic changes A_1 and A_2, semantics for $\mathscr{P}$, $\mathscr{P} \oplus \{A_1\}$, $\mathscr{P} \oplus \{A_2\}$, and $\mathscr{P} \oplus \{A_1, A_2\}$ can all be different, where $\mathscr{P} \oplus \{A_1\}$ describes the configuration created by applying change A_1 to the original program $\mathscr{P}$. We use the term *semantic dependences* to describe differences in test results due to these effects. As demonstrated by Example 4.4.2 discussing `test2` and changes 5 and 6, semantic dependencies can change test results and thus have to be considered.

Definition 4.4.3 (Semantic Dependences) *A set of atomic changes C_1 is* semantically dependent *on a set of atomic changes C_2, iff*

$$\exists T \in \mathscr{T} : R_{\mathscr{P} \oplus (C_1 \cup C_2)}(T) \neq R_{\mathscr{P} \oplus C_1}(T).$$

Note that the relation implicitly defined above in general is not reflexive or transitive, considerably complicating the task to find an optimal set of committable changes. In Definition 4.4.2 only syntactic dependences have been captured, semantical dependences have not been taken into account yet. As a consequence, the set of potentially committable changes currently is too large, and the desired property that releasing these changes without breaking tests in the repository will not be maintained in general.

It turns out that detecting semantic dependences among changes is a hard problem for static program analysis. While in some cases computing *data dependences* in either the original or the edited version (reaching definitions problem) might be enough, for other cases semantical dependences are considerably harder to detect, especially as data dependences can emerge or be removed by applying different subsets of atomic changes. As a consequence it is necessary to examine all $n!$ different configurations resulting from n changes individually. An exhaustive search here is clearly not feasible. In the following we will thus refrain from applying a sophisticated static analysis, but instead use an hybrid approach to statically and experimentally detect semantic dependences.

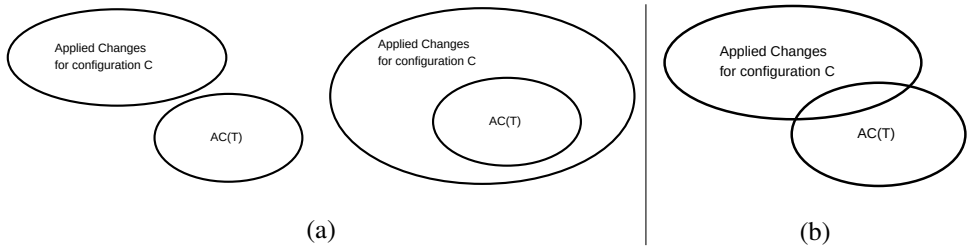


Figure 4.14: Relevant and irrelevant tests. Figure (a) shows the case where either all changes affecting a test T are applied or not applied. Figure (b) shows the relevant case that only a subset of the affecting changes is applied, so potentially breaking semantic dependences.

4.4.4 Detecting Semantic Dependences

We will first examine when semantic dependences are relevant at all. Informally this is the case if not all changes affecting a given test can be committed.

Definition 4.4.4 (Configuration) *$C = \mathscr{P} \oplus \mathscr{A}_C$ is the configuration under consideration, where $\mathscr{A}_C \subseteq \mathscr{A}$ is the subset of changes applied to configuration C. For a set of changes $\mathscr{A}_0$ we write $\mathscr{A}_0^*$ to address the set of changes resulting from $\mathscr{A}_0$ by transitively adding all prerequisite changes, such that $\mathscr{P} \oplus \mathscr{A}_0^*$ is guaranteed to be compilable.*

Semantical dependences can be relevant, if for a given test T the set of affecting changes is *no subset* of $\mathscr{A}_C$. In this case the behavior of T potentially changes as semantical dependences may exist between changes in $AC(T) \cap \mathscr{A}_C$ and $AC(T) \cap \overline{\mathscr{A}_C}$, which are broken by not applying the complete set.

Figure 4.14 illustrates the above graphically. Figure (a) shows the case where either all changes $AC(T)$ affecting a test T are applied or not applied, i.e. $AC(T) \cap \mathscr{A}_C = AC(T) \vee AC(T) \cap \mathscr{A}_C = \emptyset$. Figure (b) shows the relevant case that only a subset of the affecting changes is applied, so potentially breaking semantic dependences, i.e. $AC(T) \cap \mathscr{A}_C \subset AC(T)$.

Definition 4.4.5 (Affecting Applied Changes) *We call the set $\mathscr{A}_C \cap AC(T)$ the* affecting applied changes *for a test T and use the notation $AAC_C(T)$ for them, where C indicates the configuration.*

For each test T, committing a non-trivial subset of AC(T) (i.e. neither AC(T) itself nor $\emptyset$) might break semantic dependences among changes in AC(T) and thus potentially degrade the results of T, thus violating the commit policy. However, note that sometimes breaking semantic dependencies can be positive (by improving test results). In other cases there simply are no semantic dependencies (i.e. no visible change in test results). In either case $AAC_C(T)$ can be committed without degrading test results.

Example 4.4.4 (Change Dependences, Relevant Tests) *To illustrate the problem consider Figure 4.15, showing tests and their affecting changes and dependences (both syntactic and semantic) among changes for the example presented in Section 4.4.2. We know the results of each test in the original and the edited version of the program. For the given example,* `test1` *is the only worsening test, all other tests pass in $\mathscr{P}$ and $\mathscr{P}'$. Thus all changes affecting* `test1` *are considered to be not committable.*[11] *As a result for tests* `test2` *and* `test4` *now only a subset of the affecting changes is applied (change 6 has been removed). Hence results of these*

[11] Note that this already is a first approximation, as in general only a subset of the affecting changes might be responsible for the degrading results. If the failure inducing subset of the affecting changes is known, the algorithm can be started with this subset instead of all affecting changes as input to improve accuracy. However, for now we will remove changes 2, 3, 4 and 6 and only apply the remaining changes.

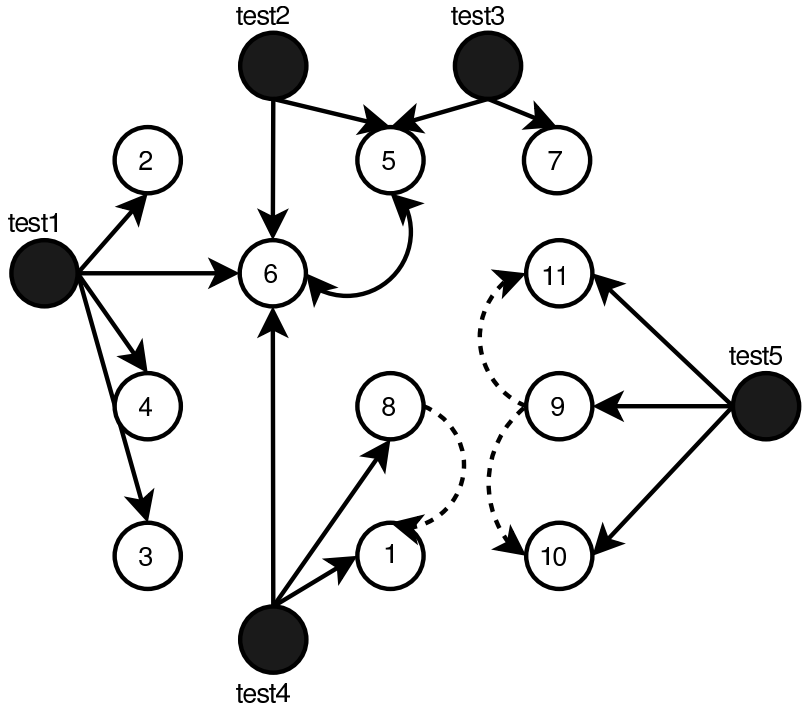


Figure 4.15: Graph illustrating relations of tests and changes. Black circles represent tests, white circles changes. Arrows indicate that a test is affected by the respective change; solid arcs show semantic, dotted arcs syntactic dependences.

tests might change compared to the known results $R_{edit}(\texttt{test2})$ *and* $R_{edit}(\texttt{test4})$*. Only those two tests are relevant for further analysis. We generalize this observation by defining a set of relevant tests as follows.*

Definition 4.4.6 (Relevant Tests) *For a given configuration C, the set of* relevant tests *is defined as*

$$\mathscr{T}_{rel} = \{T \in \mathscr{T} : AAC_C(T) \notin \{\emptyset, AC(T)\}\}.$$

To calculate relevant semantic dependences we propose an algorithm based on a combination of static approximation and also optimistically *repeatedly re-executing relevant tests* in a program configuration automatically created by our system. This approach has a classical time/space trade-off: The more test runs we use to compute the solution, the more precise our results are, but the longer it takes to compute them. An important characteristic of our approach is that the user can *interactively decide* how much time he wants to use to improve results—aborting the algorithm is possible after each iteration, and will always yield safe results.

Properties of the Search Space

Before discussing the algorithm, we will first examine the search space to find a solution in. Consider the schematic representation of the search space in Figure 4.16. The left part of the table shows the different *configurations*. Each box represents a change, and an "x" indicates this change has been applied in the given configuration. For a given set of n changes, there are 2^n different program configurations, clearly making an exhaustive search impossible.

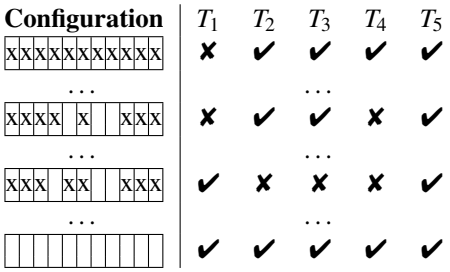


Figure 4.16: Illustrating the Search Space

The right side of the table shows the *development of test results* for the given configuration. The symbol ✔ indicates that for the given set of applied changes the test is in $\overline{WT}_{\mathscr{P},C}$, ✘ shows that the respective test is in $WT_{\mathscr{P},C}$.

The two extrema of the search space are the original version (no change applied) and the edited version (all changes applied). Each set of committable changes has the property that no test is in *WT* for the corresponding configuration. Obviously the empty set (i.e. the original program) is a solution to this problem. However, we are interested in finding a *maximal solution* or at least a good approximation of it.

This optimization problem is in a way the dual problem to finding a *minimal* set of failure inducing changes as addressed by *Delta Debugging*. It is thus worth reviewing the properties of this search space as also examined there [64, 66, 67]. However, in contrast to Delta Debugging we do not examine a single but multiple tests and their results, and we also use a structured set of atomic changes instead of pure textual differences.

Monotonicity A set of changes $\mathscr{A}$ is *monotone*, if

$$\forall C \subseteq \mathscr{A} : \exists T \in \mathscr{T} : R(T) = \text{✘} \implies \forall C' \supset C : R(T) \neq \text{✔}$$

holds, i.e. monotonicity states that adding additional changes to a configuration where we can observe failing tests will not make these tests pass.

Unambiguity A set of changes $\mathscr{A}$ is *unambiguous*, if

$$\forall C_1, C_2 \subseteq \mathscr{A} : \exists T \in \mathscr{T} : R_{C_1}(T) = \text{✘} \wedge R_{C_2}(T) = \text{✘} \implies R_{C_1 \cap C_2} \neq \text{✔}$$

holds, i.e. applying the intersection of changes applied at two configurations sharing a worsening test T will not result in a configuration where this test passes.

Consistency A set of changes $\mathscr{A}$ is *consistent*, if

$$\forall C \subseteq \mathscr{A}, \forall T \in \mathscr{T} : R_C(T) \neq ?$$

holds, where '?' represents an unresolved test outcome (e.g. a non compilable program).

Unfortunately it turns out that for configurations composed of structured atomic changes—similarly to the unstructured changes of [64]—neither monotonicity nor unambiguity hold in general. Consider `test2` as an example. Applying only change 5 results in test failure, while additionally adding change 6 fixes the failure (i.e. the set of changes is not monotone). As applying change 6 alone results in a test failure as well, $\mathscr{P}_{orig} \oplus \{5\}$ and $\mathscr{P}_{orig} \oplus \{6\}$ also give an example for inconsistency of the change set. As a consequence it is hard to find a constructive algorithm, as it is impossible to deduce properties of subsets from the superset or vice versa.

However, in contrast to Delta Debugging we know about syntactic dependences among changes, allowing us to only consider *compilable configurations*. All valid configurations in our setup (i.e. those respecting syntactic dependences among changes) are thus consistent, as we are only interested in test results, but not in the particular failure reason. Note that a significant subset of all versions can be ruled out as inconsistent, as each legal configuration has to respect syntactic dependences among changes, or otherwise a non-compilable program will result. By always creating the transitive closure with respect to syntactic dependencies among changes, a significant amount of all configurations can be identified (thus reducing the number of configurations). Although considerably reducing the search space this is however not enough to conquer the combinatorial explosion.

The desired solution is bound by the empty set (corresponding to the original program) and the set of failure inducing changes approximated by $\mathscr{A}_{PotComm}$, as this set is identified by removing all changes from $\mathscr{A}$ (directly or transitively) affecting worsening tests. $\mathscr{A}_{PotComm}$ reduces the search space as again changes are pruned. Note that $\mathscr{A}_{PotComm}$—if a valid committable configuration—not necessarily is the/an optimal solution, as we remove *all* changes affecting a worsening test, while a subset might be enough. However this reflects our desire to have a scalable algorithm also useful with large systems/test suites. Thus the original version forms a lower bound for our search space, while to set of potentially committable changes $\mathscr{A}_{PotComm}$ is an upper bound for our search.

Although ensuring consistency and using $\mathscr{A}_{PotComm}$ as an upper bound might prune a considerable amount of the 2^n possible configurations, examining the whole search space is still infeasible. In this paper, we thus use the association of test and changes to develop a *constructive algorithm* to derive the desired results in spite of the non-monotonicity and ambiguity of the search space.

Pessimistic Static Approach

While a sophisticated static analysis to find semantic dependences is hard, a *conservative approximation* can be used to determine a safe subset of committable changes. However, as this is a very crude and pessimistic approach, a significant number of committable changes are potentially lost, thus reducing the benefits of early release of changes. For this first approximation we simply prune all changes from the configuration affecting tests where only a part of its affecting changes are committable. This can be expressed by refining the Definition of $\mathscr{A}_{PotComm}$ (4.4.2) by adding a third condition.

Definition 4.4.7 (Committable Changes) *The set of pessimistic committable changes is defined as:*

$$\mathscr{A}_{Committable} = \{A \,|\, A \notin Worsening, \forall A' : A' \preceq^* A : A' \in \mathscr{A}_{Committable}, \\ \forall T \in AT(A) : A'' \in AC(T) \implies A'' \in \mathscr{A}_{Committable}\}.$$

Definition 4.4.7 states that a change A is committable if, additionally to the criteria discussed before for any test T affected by A, all of T's affecting changes must be committable. This condition encodes the conservative assumption that semantic dependences may exist between any pair of changes that affect a given test. This condition means that for each test, either all or none of its affecting changes will be committed.

The set of changes satisfying these conditions can be easily computed by removing all changes in a connected component of a worsening test in terms of the dependence graph as shown in Figure 4.15. The problem with this algorithm is that it always assumes harmful semantic dependences and thus removes too many changes. If the test suite is not modular, this might easily fall back to the original version plus changes not affecting any test (i.e. $A_{Committable}$

is the set of *Gray* changes) as earlier experiments with this algorithm have shown [56]. We thus consider this approach the bottom-line to compare our more advanced algorithms with.

Example 4.4.5 (Static Approach) *Applying Definition 4.4.7 to Example 4.10 will result in* $A_{Committable}$ *= { 9, 10, 11}, i.e. only the changes affecting* `test5` *are committable, as can easily be verified by considering Figure 4.15.*

Note that this approach computes an intermediate configuration in the search space statically without rerunning any tests. This computation is only exploiting the knowledge about change-test association and worsening tests. We do not consider (or gather) any knowledge about test results in intermediate versions yet.

Optimistic Dynamic Approach

Due to the imprecision of the purely static approach we developed a dynamic solution to this problem. The idea is to optimistically assume that $\mathscr{A}_{PotComm}$ is indeed a set of committable changes and start the computation with $\mathscr{A}_{C_0}$=$\mathscr{A}_{PotComm}$. However, we then verify this assumption by rerunning the relevant test in $\mathscr{T}_{rel}$ for the resulting intermediate configuration. In this new configuration, several test outcomes are possible:

Test Improvement or Unchanged Result: The important fact to note is that in these cases the (new) test results do not conflict with the commit policy as no test results *degrade*, although we potentially ignored or even break some semantical dependences (in the case of improving tests). In this case no further action has to be taken.

Worsening Test: If rerunning relevant tests reveals worsening tests for this setup, we treat them similarly to worsening tests ($T \in WT$) and remove all affecting changes from $\mathscr{A}_{C_0}$. Note that removing changes potentially changes $\mathscr{T}_{rel}$. For the resulting configuration we thus have to recalculate this set and continue to check all relevant tests.

Listing 4.2 more precisely describes the algorithm to compute the desired solution. After an initialization step, the algorithm first calculates relevant tests for the given configuration (lines 7-11), then creates the program version resulting from this configuration (thereby respecting syntactic dependences in line 13) and reruns each relevant test in this program version (line 15). If the test result degraded compared to the original program version, then the optimistic assumption that no semantic dependences have been broken is falsified, and thus all changes affecting this test are removed (line 17). Note that in this case we not only remove all changes directly affecting a worsening test, but also all changes depending on one of the removed tests as well, outlined by applying a function *pruneNoPreds*. This is necessary to guarantee that the system created in the next step will compile. The steps described above are repeated as long as there are relevant tests left ($\mathscr{T}_{rel} \neq \emptyset$, line 19). Running time of the algorithm is linear in the number of changes and termination is guaranteed, as in each iteration either changes are removed from $A_{Working}$ or the set of relevant changes per definition is empty, causing the algorithm to terminate.[12]

Example 4.4.6 (Dynamic Algorithm) *Applying this algorithm to our example we have to rerun tests* `test2` *and* `test4` *in the intermediate version cerated by applying all but the changes affecting* `test1` *to the original program. The observant reader may verify that, while* `test4` *still succeeds,* `test2` *now fails due to the (now uncovered) semantic dependence among changes 5 and 6. As a consequence of this result, change 5 is removed from* $A_{Working}$ *for the next iteration. As change 5 also affects* `test3`*, we now have to rerun this test in the new configuration. However,* `test3` *still succeeds, thus no other change has to be removed. Consequently there are no more relevant tests and the algorithm terminates, resulting in* $A_{Committable}$ *= {1, 7, 8, 9, 10, 11}.*

[12] If $A_{Working} = \emptyset$, then also $\mathscr{T}_{rel} = \emptyset$.

Listing 4.2: Dynamically Calculating Committable Changes

```
1  input: A_Start
2  output: A_Committable
3
4  A_Working = A_Start; A_Last = ∅;
5  T_rel = {T ∈ T : AAC_Start(T) ∉ {∅, AC(T)}};
6  do {
7    ∀T ∈ T_rel do {
8      AAC_Last(T) = A_Last ∩ AC(T);
9      AAC_Working(T) = A_Working ∩ AC(T);
10   }
11   T_rel = {T ∈ T : AAC_Last(T) ≠ AAC_Working(T) ∧ AAC_Working(T) ∉ {∅, AC(T)}};
12   A_Last = A_Working;
13   create P ⊕ A_Working;
14   ∀T ∈ T_rel do {
15     rerun T_rel in P ⊕ A_Working
16     if (R_{P⊕A_Working}(T) < R_P(T))
17       A_Working = pruneNoPreds(A_Working − AC(T));
18   }
19 } while (T_rel ≠ ∅); (**)
20 return A_Working as A_Committable;
```

It is possible that the solution calculated by Algorithm 4.2 falls back to the static solution, however the assumption that semantic dependences always exist is clearly overly conservative, and thus Algorithm 4.2 yields better results in general—however at the costs of runtime. Iteratively rerunning tests is very expensive compared to the purely static approach.

Mixed Dynamic/Static Approach

To deal with the weaknesses of both approaches—the unsatisfactory precision yielded by the static algorithm and the long runtime of the dynamic algorithm—we can combine both algorithms by first allowing k dynamic iterations and then post-processing the resulting change set using the static approach. Using this setup the end user can use this parameter k to decide interactively how much time he wants to invest to calculate committable changes. The user can abort calculation before each test run, resulting in a switch from the dynamic to the static algorithm. As the results of the static algorithm always yields a valid solution, the combined algorithm also produces a safe result.

To extend the dynamic algorithm we only need to replace line 19 marked with (**) in Algorithm 4.2 with the following two lines:

```
19 } while (T_rel ≠ ∅ ∧ notAborted);
20 A_Working = static_algorithm(A_Working);
```

The variable *notAborted* indicates a check if the user has aborted the computation. This simple replacement now allows to combine the strength of both algorithms. The call to $static\_algorithm(A_{Working})$ simply performs the calculation shown in Definition 4.4.7, where $\mathscr{A}_{Committable}$ is replaced with $A_{Working}$.

Example 4.4.7 (Hybrid Algorithm) *Consider the application of the dynamic algorithm to our example. Assume that the user aborts the calculation after tests* `test2` *and* `test4` *have been rerun. So after removing change 5, we apply the static algorithm. As* `test3` *now is*

relevant, we remove all changes reachable through `test3`*, i.e. change 7. The resulting set of committable changes is thus smaller ($A_{Committable}$ = {1, 8, 9, 10, 11}), but we saved one iteration as test* `test3` *has not been rerun.*

Note that there are two important differences compared to running the static algorithm up front. First, the set of relevant tests is different, and second the resulting dependence graph has also been thinned out by the previous dynamic calculation steps, in general resulting in more and smaller connected componentes.

4.4.5 Conclusions and Future Work

First, in this section we addressed the problem that anticipation of changes by other team members can be delayed, as the desire to maintain a consistent repository prevents developers to release changes at will in the presence of locally failing tests. As a result merging and conflict resolution can be expensive due to the large amount of accumulated changes. The algorithm presented here allows to automatically derive a set of committable changes so leveraging these efforts. The simple example used in this section is clearly not enough to draw any conclusions, so more and especially larger real world case studies about the effectiveness are needed. We are currently working on finishing the implementation and calculating committable changes for the Daikon version pair used as experiment in Section 4.3.6 (page 97). If early release of changes is beneficial at all depends on how many changes on average are committable, and on how many iterations in the dynamic algorithm are needed to calculate them.

An interesting research question might also be if there is a value for the number of iterations k which can be recommended to achieve optimal results in terms of the underlying time/precision trade-off. The underlying assumption is that changes removed in later iterations are less relevant (in terms of failure-induction) than those removed in earlier iterations.

To finally show the effectiveness of the tool, a user study would be necessary.

4.5 Change Impact Analysis for Aspect-Oriented Programming

In this final section we describe how the techniques discussed in previous sections can be lifted to deal with the peculiarities of aspect-oriented programming.

In this context, we distinguish two scenarios. Change impact analysis can be used to determine the effects of adding aspects to an existing (otherwise unchanged, potentially already aspect-oriented) system. This scenario has been implemented and we report about case studies we conducted with our prototype.

In the presentation of the underlying theory however, we extend this approach and show how change impact analysis can also be used to deal with system evolution for aspect-oriented systems in general, including addition and removal of aspects and modification of both aspect and base code.

4.5.1 Analysis Variants and Example

The first scenario examines a single program version to gather information about kind and effects of applying aspects. This information is vital for programmers to estimate if and how the semantics of their base modules are modified by applying an aspect. Knowledge about semantic effects of aspects enables programmers to focus their attention on relevant aspects when reasoning about system correctness. This scenario thus directly attacks problems P1, P2 and P4 stated in Chapter 2.3.

```
1  class C {
2    int x, y;
3
4    void foo() {
5      x = 0;
6    }
7    void bar() {
8      x = -1;
9    }
10   void zip() {
11     x = 1;
12   }
13   void zap() {
14   }
15 }
16
17 aspect A1 {
18   pointcut p1(C c):
19     call (* C.*(..)) &&
20     target(C) && !withincode
21       (* testPassFail ())[1,13];
22
23   after (C c): pc1(c) {
24     c.x++;
25     c.y++;[2]
26   }
27 }
```

```
28 aspect A2[3,4,5,6,7,8] {
29   pointcut p2(C c):
30     set(int C.x) && this(c);[9,10,11]
31   after(C c): p2(c) {
32     c.x = c.x * 2;
33   }
34 }
35
36
37 class Tests extends TestCase {
38   public void testFailFail() {
39     C c = new C(); c.foo();
40     assertTrue(c.x < 0);
41   }
42   public void testPassFail() {
43     C c = new C(); c.bar();
44     assertTrue(c.x != -1);
45   }
46   public void testFailPass() {
47     C c = new C(); c.zip();
48     assertEquals(3, c.x);
49   }
50   public void testPassPass() {
51     C c = new C(); c.zap();
52     assertTrue(c.x >= 0);
53   }
54 }
```

Figure 4.17: Simple example program. The original version $\mathscr{P}$ consists of all code fragments except those underlined, the new version $\mathscr{P}'$ is created by adding the underlined code fragments. While aspect A_2 is not changed, we will only apply the aspect to $\mathscr{P}'$, but not to $\mathscr{P}$ (indicated by underlining A_2). Aspect A_1 is applied to both program versions.

The second scenario examines change in aspect-oriented systems. Thus these techniques address multi version program analysis. Although this scenario at first seems to be considerably different compared to the single version case, this is actually not the case. Even in the first scenario two program versions are implicitly compared—a version with, and a version without application of the aspect(s) of interest. The core assumption however is that only the set of applying aspects but neither base modules nor aspects itself change. For the second scenario this restriction is removed, i.e. we also capture the effects of changes to base modules or aspects itself. Obviously this directly attacks evolution problems of aspect-orientation (P3), but as well problems related to scalability (P1, P2).

To demonstrate the change impact analysis in the context of AspectJ, we will use the simple example program shown in Listing 4.17 as example. The original version $\mathscr{P}$ consists of all code fragments except those underlined, the new version $\mathscr{P}'$ is created by adding the underlined code fragments. While aspect A_2 is not changed, we will only apply the aspect to $\mathscr{P}'$, but not to $\mathscr{P}$ (indicated by underlining A_2). Aspect A_1 is applied to both program versions.

4.5.2 Calculating Changes

To transport change impact analysis as introduced in [46] to aspect-orientation, a necessary prerequisite is to extend the calculation of atomic changes—as implemented in the Chianti

Category	Semantics
ARI	Add Reference to Interface
DRI	Delete Reference to Interface
ARS	Add Reference to Superclass
DRS	Delete Reference to Superclass
ARA	Add Reference to Aspect
DRA	Delete Reference to Aspect

Table 4.9: New change categories to capture Hierarchy Modifications

framework for Java—to AspectJ.

From a theoretical point of view, AspectJ programs are a superset of Java programs. As such, all change categories present in Java can also occur in AspectJ programs. However, due to the aspect-oriented language extensions in AspectJ, we also need some additional change categories. The following briefly reviews the new language constructs in AspectJ (compared to Java) and discussed how we derive atomic changes for them, if appropriate.

Handling Inter Type Declarations

Handling of inter type declarations in terms of changes is straightforward, as inter type declarations simply add new methods and fields to a class, although not in it's lexical scope. Consequently, we generate **AM**, **CM**, **AF**, **AFI**, **CFI**, and **LC**-changes for inter type declarations accordingly.

Note that to derive **LC**-changes, we use the lookup changes calculated for the static crosscutting non-interference criterion presented in Chapter 3. This modeling reduces inter type declarations to traditional member additions, and thus allows to easily handle them.

Hierarchy Changes

Beside inter type declarations, static crosscutting in AspectJ also allows to change the direct superclass and add additionally implemented interfaces (Section 3.2.2). This is however also possible by using traditional source code modifications.

The *Chianti* system currently does not capture such changes in the type hierarchy directly, but reports **CM** changes for methods containing `instanceof` statements referring to the modified classes. However, as these changes are relevant for aspect-oriented programs—just consider pointcut semantics containing `within`, `this` or `target` predicates—we additionally explicitly model such changes to capture potential semantical changes. Table 4.9 gives an overview of the new change categories introduces to record respective changes.

We generate a respective change each time the superclass (**ARS**, **DRS**), super aspect (**ARA**, **DRA**) or the set of implemented interfaces (**ARI**, **DRI**) changes, either due to static crosscutting or due to regular edits of `extends` or `implements`-statements in the source code.

We also report virtual **CP** (change pointcut) changes for pointcuts now potentially selecting additional or less joinpoints (using the 'TypeName+' syntax). These **CP** changes syntactically depend on the hierarchy changes.

Pointcuts and Advice

Beside static crosscutting constructs we have to handle changes in pointcuts and advice. As advice is a method-like construct, it is tempting to handle advice similarly to methods. This is however insufficient, as for advice two components have to be considered: the advice header and the advice body. The advice header contains information about the available joinpoint context and *the pointcut* this piece of advice is bound to.

Category	Semantics
AA	Add an empty aspect
DA	Delete an empty aspect
AAD	Add Advice Definition
CAH	Change Advice Header
CAB	Change Advice Body
DAD	Delete Advice Definition
AP	Add new Pointcut Definition
CP	Change Pointcut Definition
DP	Delete Pointcut Definition

Table 4.10: New change categories to capture Hierarchy Modifications

The advice body is similar to a method body and is also handled similarly. If the advice body is changed, we generate a **CAB** change (change advice body). However, if only the binding in the advice header is changed, we generate a **CAH** change (change advice header) instead. Similar to methods, we also have change categories to indicate addition or removal of a piece of advice (**AAD** (add advice definition) and **DAD** (delete advice definition), respectively).

Pointcut definitions can occur as a part of the advice header or as individual aspect elements. Consequently we also have change categories to indicate addition, change, and removal of pointcut definitions, **AP**, **CP**, and **DP**, respectively.

Aspects

Beside classes AspectJ finally also introduces aspects as modules containing these new constructs. Changes for new or removed aspects are generated similarly to classes, i.e. we generate **AA** and **DA** changes (add, delete aspect). Table 4.10 gives an overview of changes induced by pointcuts, advice and aspects.

Example 4.5.1 (Changes Due to Code Edits) *When examining $\mathscr{P}$ and $\mathscr{P}'$, we can only see changes in aspect A_1. In particular, pointcut* `pc1` *has been changed, resulting in a* **CP** *change (change 1) for this pointcut (line 21), as well as the advice body, resulting in a* **CAB** *change (change 2, line 25).*

*Addition of aspect A_2 results in a set of changes. First, the aspect itself is added (***AA** *A_2, change 3), then in the aspect a pointcut (***AP** *`pc2` and* **CP** *`pc2`, changes 4 and 5) and a piece of advice (***AAD** *6,* **CAH** *7 and* **CAB** *8) are added, resulting in the respective changes.*

Note that these changes—equivalently to the changes discussed in the Chianti work—also have dependences among each other. If for example a piece of advice is deleted, we not only generate a **DAD** change, but also a **CAB** and a **CAH** change, where the **DAD** change depends on, to illustrate that first the advice body and the advice header has to be deleted, before this piece of advice can be removed. Similarly to *Chianti*, a change A_1 is syntactically dependent on a change A_2, if adding A_1 without also adding A_2 yields a syntactically incorrect AspectJ program. As these simple syntactic dependences are similar to the ones described for *Chianti* in [46] we will not discuss them any further here.

The calculation of atomic changes $\mathscr{A}$ between an original program $\mathscr{P}$ and an edited program $\mathscr{P}'$ as presented in this section has been implemented by Jürgen Graf as a part of his master thesis [25]. Although directly usable for the application described here, originally these changes were used to explain pointcut deltas as described in Chapter 5.

Other AspectJ Constructs

Note that there are other language constructs in AspectJ which we did not explicitly discuss up to now. Theses include `declare precedence`, `declare soft` and `declare error/warning`. The `declare error/warning` construct results in compiler errors or warnings, but has no semantics in terms of runtime behavior and is thus not relevant in this context. The `declare precedence` statement is used to explicitly declare aspect precedence and will be examined in detail in Chapter 6.

The `declare soft` statement however can be used to virtually convert a checked Java exception into a runtime exception, thus changing the type and error handling behavior for this exception. While defining respective changes is simple, calculating their semantical effects requires in depth analysis of control and data flow graphs. Capturing these effects using call graphs however is difficult, and has thus been excluded for the following. In Chapter 6 however we describe an analysis of exceptional control flow which can also be adapted to capture effects of `declare soft` statements.

4.5.3 Aspects as Change Encodings

While the changes we discussed so far directly stem from edits of the source code, to actually capture effects of aspect application, this is not enough, as advice is applied non-locally, at the joinpoints selected by referenced pointcuts.

To capture these effects, we can however interpret the aspect itself as an *encoding of a set of changes*. While inter type declarations are irrelevant in this context (apart from resulting lookup changes which already have been calculated), advice—or better the joinpoint set where advice applies—has to be considered, as semantics at relevant joinpoints are potentially modified by adding the respective advice. To capture these effects, we *generate virtual atomic changes* for the program artifact containing an adapted joinpoint. For example if a `call`-joinpoint in a method $m()$ is affected by advice, we generate a **CM** change for $m()$. Note that this also leads to **CFI**, **CSFI** and **CAB** changes, if joinpoints in the respective program items are adapted by advice.

Generating these changes is straightforward and necessary if we want to analyze the impact of a new piece of advice *adv* added to a system. However if we want to analyze the impact of changes between two versions of an aspect-oriented system, generating these changes can be imprecise, as we also have to consider advice existing in the original system. Optimally we only want to generate a change for joinpoints also adapted in the original program version if semantics of the program at that joinpoint have changed. It is a fair assumption that this is the case if a piece of advice is added or removed, but we also have to assume a semantical change if a piece of advice adapting a certain joinpoint or a pointcut selecting this joinpoint (changed exposed context!) has been changed. For multi-version impact analysis we thus have to conservatively generate changes derived from actual edits.

Let $\mathscr{P}$ and $\mathscr{P}'$ be the original and the edited program version, respectively. To calculate relevant joinpoints where virtual changes have to be generated, we analyze the matching information available from the aspect weaver. Abstractly, this is a relation associating advice and joinpoints, represented as tuples (adv, jp). By comparing the two relations for both $\mathscr{P}$ and $\mathscr{P}'$, we can calculate the set of joinpoints where a piece of advice has been newly added and joinpoints where a piece of advice has been removed (for details refer to Chapter 5). In all these cases, a virtual atomic change is created for the program artifact containing the adapted joinpoint to indicate this potential semantic change.

Example 4.5.2 (Advice Induced Changes) *Aspect A_1 attaches advice to each method call invoking a method in* `C` *(lines 39, 43, 47, and 51). Aspect A_2 attaches advice to definitions of field* `x` *of class* `C` *(lines 5, 8, and 11).*

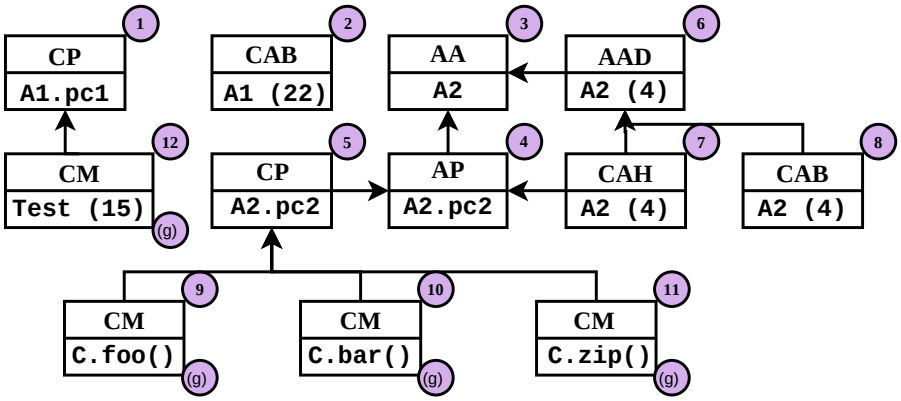


Figure 4.18: Atomic changes inferred from the two versions of the program.

However, not all of these adapted joinpoints are relevant. For aspect A_1, there is only one change in matching behavior resulting in a generated virtual atomic change. Due to change 1, the call to `bar` *is no longer adapted by advice, and we thus generate a* **CM** *change for method* `testPassFail()`, **CM** *change 12.*

As the addition of aspect A_2 results in the adaptation of the three joinpoints named above (lines 5, 8, and 11), we generate **CM** *changes for methods* `foo()`, `bar()`, *and* `zip()` *(changes 9, 10, and 11).*

To summarize the above, for the simple example program shown in Figure 4.17 we get the changes shown in Figure 4.18. In this figure (similarly to figure 4.2 in Section 4.2.1) each atomic change is shown as a box, where the top half of the box shows the category of the atomic change (e.g., **CM** for change method), and the bottom half shows the program artifact involved. To identify advice we gave the defining aspect's name and the line number the advice definition starts in. An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2. The marker (g) indicates that the respective change is a generated change to model changes in advice application behavior. Note that generated changes depend on the pointcut (or the advice header, respectively) which selects the relevant joinpoint.

4.5.4 Change Impact Analysis for AspectJ

Our goal is to reuse the change impact analysis of *Chianti* in the context of AspectJ. A major step in this direction was the definition of atomic changes for AspectJ in the previous section. Additionally we need call graphs to match these changes with. Similarly to *Chianti*, we can use either statically constructed or dynamically recorded call graphs. The research prototype we implemented to analyze the impact of new aspects uses the JDI interface to generate dynamic call graphs. While JDI has a severe performance punishment, this is only a technical restriction of our current prototype, as the tracing interface can be considerably improved by using different techniques, for example executing tests using a modified virtual machine. We also tried two other tracing approaches: first, we reused the JVMPI agent from *JUnit/CIA* and second we also used traces generated by TPTP. For details on the differences between these tracers refer to Section 4.5.7.

The resulting call graphs are however not directly usable for our analysis, as a dynamic call graph only shows called *byte code constructs*. As AspectJ is compiled to valid Java byte code, advice is no longer present as a program construct but instead transformed to traditional Java methods, and respective calls are inserted to invoke advice at adapted joinpoints. Fortunately one can create a mapping from the method names to the original piece of advice in the

respective program version which allows to use dynamic call graphs nevertheless. Note however that the mapping is not stable considering system evolution if aspects are modified. We thus used the dynamic call graphs only to analyze the impact of adding aspects to a existing system (single version analysis).

To extend the analysis to be able to use dynamic call graphs for multi-version program analysis, we suggest to mark advice to allow identification of a piece of advice independent of it's source code position as described in Section 5.2.3. Using this setup, each byte code method, including those generated for advice, can be traced to the respective source code method or advice in the respective program version (for a single version the mapping is stable). Using the identifier then allows to identify pieces of advice across program versions.

An alternative is to use static call graphs. The main challenge compared to object-oriented call graph construction is to deal with dynamic joinpoints (uncertain aspect application) and undetermined advice precedence at a given joinpoint, as this can result in a wrapping hierarchy for different pieces of advice. Static call graph construction for AspectJ programs is described in Section 6.4.2.

To summarize the above, we can define call graphs in the context of AspectJ and thus are able to provide the necessary input data structures for *Chianti* in this context as well. We can thus reuse the change impact analysis technique of *Chianti* to calculate affected tests and affecting changes.

4.5.5 Affected Tests and Affecting Changes

Calculating affected tests and affecting changes is now relatively easy. Compared to *Chianti* we only have to capture additional change categories introduced by aspect-oriented language constructs, i.e. in particular advice, as nodes representing advice are explicitly present in the call graphs.

The main challenge for multi-version program analysis in this context is to capture changed advice matching behavior due to **CP** or **CAH** changes, as such changes may result in additional or lost matches in the edited program version which are not visible in the call graph of the original program version. Note that the removal of a statement—and thus the associated joinpoint—is directly captured as a change of the respective program artifact. Fortunately, as we generate virtual changes to capture changes in advice matching and as these changes are directly linked to nodes in the call graph, no additional efforts are necessary to capture effects of **CP** and **CAH** changes.

We thus extend the original formulas of [51] by adding respective change categories for advice. The resulting formulas are shown in Figure 4.21. Note that we use the name of the change category as the name of the set containing all changes of this category.

We will now use these formulas and apply them to our example. Refer to Figure 4.19 for the call graphs of the four tests in the original version $\mathscr{P}$ and to Figure 4.20 for the call graphs in the edited version $\mathscr{P}'$. Shaded nodes in the graphs indicate nodes which have atomic changes associated with them.

Example 4.5.3 (Affected Tests) *Analyzing the call graphs for the original program version shows that all tests but* `testPassPass` *are affected. These three call graphs (shown in Figure 4.19) all contain a node representing the* `after`*-advice from aspect* A_1 *which is associated with a* **CAB** *change (change 2). Additionally each of these three call graphs also contains at least one node associated with a generated atomic change (changes 9, 10, and 11).*

For the affected tests, we can now also calculate the affecting changes.

Example 4.5.4 (Affecting Changes) *Analyzing the call graphs of the three affected tests in the edited program version (shown in Figure 4.20) yields the following results. By traversing the call graph of test* `testFailPass` *we collect the* **CM***-change for* `zip()` *(change 11), and*

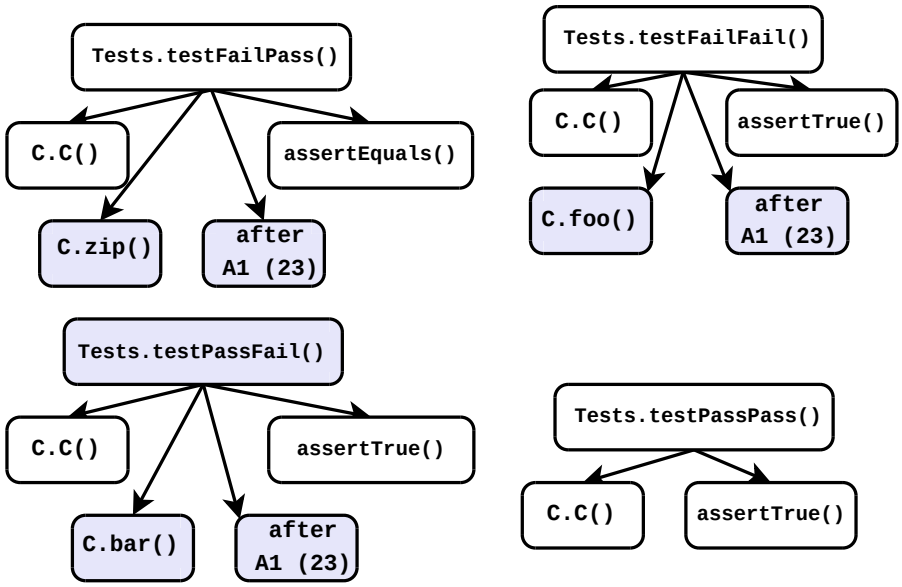


Figure 4.19: Call Graphs for original program version $\mathscr{P}$.

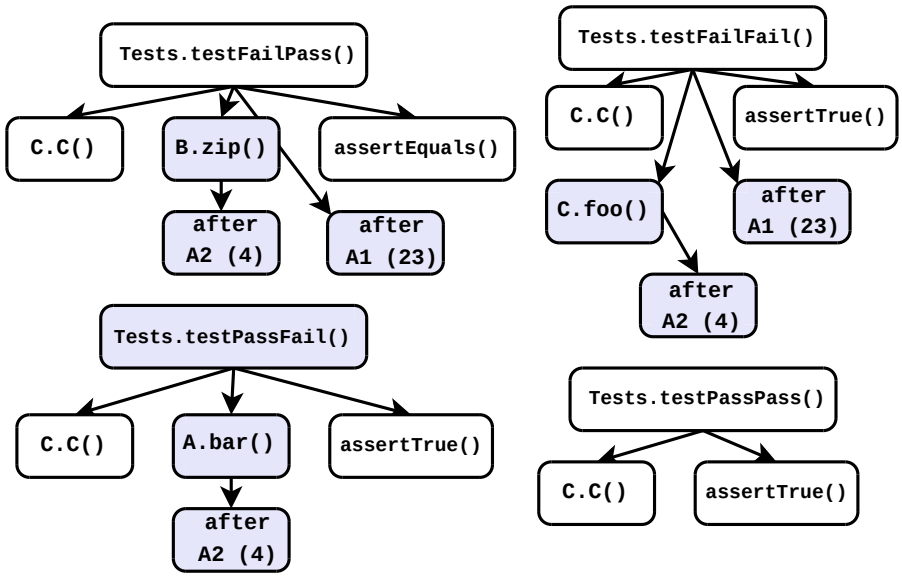


Figure 4.20: Call Graphs for edited program version $\mathscr{P}'$.

also the **CAB***-changes for the applied piece of advice in aspect* A_1 *(change 2) as well as the changes corresponding to the addition of the new piece of advice from* A_2 *(changes 6, 7, and 8). Similarly, for test* `testFailFail` *we get changes 2, 6, 7, 8, and 9 and for test* `testPassFail` *changes 6, 7, 8, 10, and 12.*

Note that the results we get by lifting the change impact analysis of *Chianti* to AspectJ can considerably help programmers to estimate the impact of (i) adding a new aspect to given system as well as (ii) modifying an aspect-oriented system.

In detail, we have the following results:

$$
\begin{aligned}
&AT(\mathscr{T})_{\mathscr{A}} = \\
&\quad \{\, T_i \mid T_i \in \mathscr{T},\ Nodes(\mathscr{P}, T_i) \cap (\mathbf{CM} \cup \mathbf{DM} \cup \mathbf{CAB} \cup \mathbf{DAD})) \neq \emptyset \,\} \cup \\
&\quad \{\, T_i \mid\ \ T_i \in \mathscr{T},\ n, A.m \in Nodes(\mathscr{P}, T_i), \\
&\qquad\quad n \rightarrow_{B,\ X.m} A.m \in Edges(\mathscr{P}, T_i), \\
&\qquad\quad \langle B, X.m \rangle \in \mathbf{LC},\ B <^* X \,\}
\end{aligned}
$$

$$
\begin{aligned}
&AC(t)_{\mathscr{A}} = \\
&\quad \{\, a' \mid a \in Nodes(\mathscr{P}', T) \cap (\mathbf{CM} \cup \mathbf{AM} \cup \mathbf{CAB} \cup \mathbf{AAD}),\ a' \preceq^* a \,\} \cup \\
&\quad \{\, a' \mid\ a \equiv \langle B, X.m \rangle \in \mathbf{LC},\ B <^* X, \\
&\qquad\quad n \rightarrow_{B,\ X.m} A.m \in Edges(\mathscr{P}', T), \\
&\qquad\quad \text{for some } n, A.m \in Nodes(\mathscr{P}', T),\ a' \preceq^* a \,\}
\end{aligned}
$$

Figure 4.21: Affected Tests and Affecting Changes.

Test T	$R_{orig}(T)$	$R_{edit}(T)$	Classification	Affecting Changes
`testPassPass`	PASS	PASS	–	(not affected)
`testPassFail`	PASS	FAIL	*WT*	6, 7, 8, 10, 12
`testFailPass`	FAIL	PASS	*IT*	2, 6, 7, 8, 11
`testFailFail`	FAIL	FAIL	(same result)	2, 6, 7, 8, 9

Table 4.11: Overview: Test Results, Test Classification and Affecting Changes

(i) By explicitly generating atomic changes for adapted joinpoints, the lifted change impact analysis makes changes to the semantics of program items with applying advice explicit. The relevant aspect-oriented language construct by construction is always the change a generated change depends on.

This is close to the information presented by *ajdt* [1]. Recently *ajdt* also added a *Cross-cutting Comparison View* which derives differences in matching behavior, however without explaining them. Our analysis offers considerably more information why semantics have changed. *Ajdt* for example does not calculate lookup changes.

(ii) As the change impact analysis of *Chianti* associates changes and tests, test failures due to aspect-oriented constructs can easily be traced to the responsible program items. In the context of aspect-orientation this is especially interesting, as the source code of a program artifact containing an adapted joinpoint shows no reference to the potentially failure inducing advice.

(iii) The results of the change impact analysis of *Chianti* can be used as a metric for the quality of the test suite. Statement coverage or branch coverage is not sufficient as a coverage criterion for an aspect. It is rather necessary to test each single application of an advice defined by an aspect. As each such application corresponds to a (generated virtual) atomic change, we can check if all these changes can be associated with at least a single test case.

4.5.6 Debugging Support for AOP

By lifting the change impact analysis of *Chianti* to AspectJ, we can now also directly reuse the change classification of *JUnit/CIA*. Change classification needs the relation of tests and changes as well as the test results as input. Both are available now, so we can analyze the above example using change classification.

Note that the test names, similarly as in the example presented in Section 4.3, indicate the result for these test in both program versions $\mathscr{P}$ and $\mathscr{P}'$. Test `testPassFail` for example passes in the original program version, but fails in the edited program. Table 4.11 gives an overview of test results and affecting changes. We now use this information to classify the changes of our example.

Example 4.5.5 (Change Classification) *We start with the auxiliary change classification and get Worsening={6, 7, 8, 10, 12}, Improving={2, 6, 7, 8, 11}, and SomeFailFail={2, 6, 7, 8, 9} (the remaining sets are empty). We then classify changes as follows: Red={10, 12}, Green={2, 9, 11}, Yellow={6, 7, 8}*[13]

As the observant reader may verify, in the above example, the *Red* changes exactly pinpoint the problem, undoing either change 10 or change 12 results in `testPassFail` to PASS again.

Note that both changes 10 and 12 are generated changes. Change 12 indicates the removal of the advice from A_1, while change 10 indicated the addition of the new piece of advice. Thus the change classification in this case not only enables programmers to focus on the problematic advice, but gives additional information, as we actually classify *advice at a certain joinpoint*.

When experiencing the test failure, the programmer examines the results of the change impact analysis. What he sees is that removal of advice from A_1 and addition of advice from A_2 results in the observed test failure. Additionally, as not all joinpoints adopted by the new piece of advice are *Red*, the problem seems to be related to the test-specific context rather than to the piece of advice in general. Consequently our analysis allows to derive joinpoint-specific results helping programmers to correctly estimate joinpoint-advice mismatches.

4.5.7 Implementation

Helmut Zechmann implemented the first version of the single-version aspect impact analysis based on the JDI tracer as part of his master thesis [63]. To implement the complete multi-version analysis introduced in this section (demonstrated by the example), the change calculation used for pointcut delta analysis 5 [25] has to be combined with this first prototype. While for a practical tool this extension is important, to demonstrate the applicability of our proposed lifting of the change impact analysis of *Chianti* to AspectJ, the current prototype however suffices.

In the single version analysis prototype, an aspect is interpreted as a change encoding, and the resulting changes are mapped to nodes and edges of dynamic call graphs. This scenario thus serves to analyze impact of an aspect to-be-added to a given (aspect-oriented) system. Using this prototype, we performed several case studies in the spirit of [46] to get a feeling for the number and kind of changes induced by aspects.

Note that this prototype only uses the subset of all changes we outlined in Section 4.5.3. To derive aspect impact, we only calculate **AM**, **AF**, and resulting **LC** changes (but not additional **CM** changes associated with each **AM** change as *Chianti* does), and changes to indicate adaptation of a joinpoint (e.g. a **CM** change for a method that contains a call adapted by a new aspect). In the following we will report which and how many changes resulted from the different aspects we found, inspired by the case study in [46]. Note that for these numbers changes beside the above-mentioned **AM**, **AF** and **LC** changes are in general generated virtual atomic changes. These numbers thus also give an interesting overview how many joinpoints are matched by a given aspect.

The calculation of changes in the style of *Chianti* has also been implemented, although we do not match these changes with the call graphs yet. Theses changes are instead used to explain pointcut deltas as demonstrated in Chapter 5.

[13] While the choice of the *Red*-criterion is irrelevant in this case, note that we chose the G_r criterion to classify *Green* changes here.

Trace Engines

Before discussing our case studies in detail we will first briefly introduce some options we discussed and the different tracing approaches we actually used for these case studies.

Tracing using an Aspect. A valid question—as we are dealing with aspect-oriented programs in this thesis—is: "Why not use an aspect for tracing?" Actually producing a trace with an aspect is very simple. However, the problem in this case are the subject programs. As these programs are not Java but AspectJ programs, they per definition contain aspects which we also want to trace. This however potentially results in aspect interference. While it is possible to specify that the tracing aspect has more precedence than any other (subject) aspect, such a directive is of course legal AspectJ code and can thus be also contained in any subject program—for a different aspect, so resulting in conflicting statements. We thus did not use a tracing aspect to trace aspect-oriented programs to avoid interference problems.

Instrumentation. Program instrumentation has two important disadvantages in the context of AspectJ. The byte code is already modified by the AspectJ weaver, and instrumenting modified byte code in some cases was problematic (especially with coverage tools). More importantly however, instrumentation results in an additional step where the user has to explicitly define which classes have to be instrumented and traced. Especially tracing calls to and from libraries in this context is not trivial. We thus preferred an approach which did not touch the class files.

Tracing via JDI. This is the first tracing engine we implemented for our prototype. The JDI (Java Debugging Interface) is a virtual machine interface and thus allows to trace every program without any necessary preprocessing. Using this interface allows fine grained access to nearly all program data at runtime. However, it has one important disadvantage—runtime. From all implemented tracers, the JDI-based tracer is by far the slowest.

The *JUnit/CIA* JVMPI Tracing Agent. The *JUnit/CIA* Tracing Agent is based on the JVMPI (Java Virtual Machine Profiling Interface). JVMPI agents are not written in Java, but in C or C++. The *JUnit/CIA* agent is a customized agent directly creating call graphs, thus avoiding dumping large trace files on disk. However, this agent was not available for the first prototype. We thus integrated it later in order to improve performance of the system. As our case studies showed, avoiding the trace files and creating call graphs in memory considerably improves performance.

Tracing with TPTP. TPTP is a fast publicly available JVMPI-based tracing agent. However, using TPTP it is not possible to access the runtime type of a call target. Without this information it is not possible to correctly match edges in the call graph with changed lookups. We could however use this tracer if no lookup changes existed in the system to analyze. As we only experienced few lookup changes in our case studies TPTP is a relevant alternative.

We used the TPTP profiling agent in the `HSQLDB` case study due to the considerably better performance (for more details refer to 4.5.9). Although TPTP does not allow to access the runtime type of the callee, and that consequently we cannot correctly associate edges in the call graph with **LC** changes it was applicable in that context as we did not experience any **LC** changes for any of the analyzed aspects in this case study.

For a detailed comparison of performance data of the above trace engines refer to Section 4.5.9. Functionally, all tracers yielded the same results in our case studies.

Version	Tests	Applied Aspects	Build Configuration
0	`BasicSimulation`	None	*basic*
1	`BasicSimulation`	`Timing`, `TimerLog`	*timing*
2	`BasicSimulation`	`Timing`, `Billing`	*billing*
3	`TimingSimulation`	`Timing`, `TimerLog`	*timing*
4	`TimingSimulation`	`Timing`, `Billing`	*billing*
5	`BillingSimulation`	`Timing`, `Billing`	*billing*

Table 4.12: Different Versions of `Telecom` Example used in our Case Study

4.5.8 Evaluation

We continue with a detailed discussion of case studies we conducted with our single version prototype. For a description of the programs that were subjects in our case studies, refer to Chapter 7.

The AspectJ Examples

Unfortunately the AspectJ examples are only small programs (few 100 LoC) and do not contain any JUnit tests. However, the `Telecom` example contains three *simulation classes* which execute the system in a controlled way. We used a subset of these classes to create equivalent JUnit tests to execute the system and create call graphs.

Some of the three simulation classes explicitly demonstrate features added by the aspects, thus only the configurations shown in Table 4.12 are compilable. We thus analyzed version pairs $(0,1)$, $(1,2)$, and $(3,4)$ to determine the impact of aspects `Timing` and `Billing` on the tests `BasicSimulation` and `TimingSimulation`.

Impact of `Timing` and `TimerLog` on test `BasicSimulation`: This setup corresponds to version pair $(0,1)$ from Table 4.12. Our tool correctly detects that the test is affected by the addition of the `Timing` aspect.

Figure 4.22 shows a screen shot of the AOPA tool suite presenting our analysis results to the user. The eclipse view presents the results in four categories, *Atomic Changes*, *Affected Tests*, *Safe Tests*, and *Untested Changes*. The *Atomic Changes* category presents all atomic changes derived from the aspects by change category. Each change is associated with the causing aspect construct. In the screen shot the first **CM** change for example is due to the `after`-advice in `Timing`. The *Affected Tests* category shows all affected tests together with their affecting changes. The *Safe Tests* category lists all tests not affected by any change. Finally the *Untested Changes* category shows all changes not affecting any tests. For each change or test, double clicking on the view entry opens the Java editor in Eclipse, and thus allows the programmer to quickly navigate to the relevant code locations.

Figure 4.23 (a) gives an overview of the atomic changes induced by the aspect. In this scenario we only add aspect `Timing` to the system. Aspect `Timing` results in 2 **AM**, 2 **CM**, 2 **CAB** and 2 **AF** changes. The two **AM** changes correspond to new methods defined in aspect `Timing`, the two **AF** changes to two fields added to classes `Customer` and

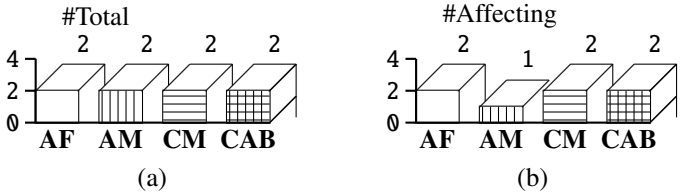


Figure 4.23: `Telecom` and `TimerLog` aspects and associated numbers of total (figure (a)) and affecting (figure (b)) Atomic Changes in each Change Category.

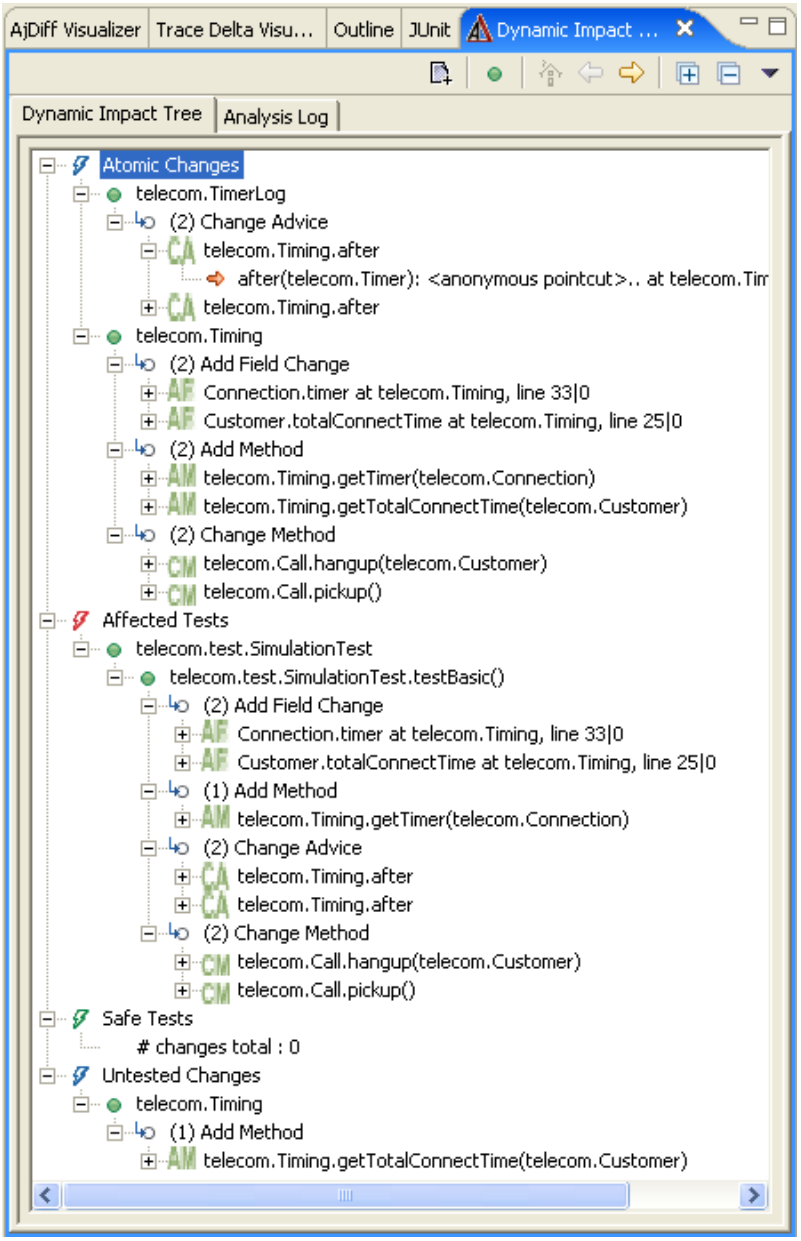


Figure 4.22: Screen shot of AOPA presenting the results of our Dynamic Impact Analysis for the `Telecom` example comparing build configurations *basic* and *timing*.

`Connection` via inter type declarations. The **CM** and **CAB** changes correspond to joinpoints adapted by the `Timing` aspect. As we only examine a single test in this example, all affecting changes actually affect this test.

The two **CM** changes model the two adapted joinpoints due to advice added by `Timing`, so showing the impact of advice. As advice defined in the `Timing` aspect itself is affected by aspect `TimerLog` (this aspect is always present), we also see two **CAB** changes modeling this advice influence.

Consider Figure 4.23 (b) for an overview of the affecting changes. From these changes, only the **AM** change (`Timing.getTotalConnectionTime`) is not affecting any test, as this

method is only called by the timing test (which does not compile when using the *basic* build configuration). We explicitly point out that both **CM** and also both **CAB** changes are covered by the (single) test.

This information can be used to give programmers some confidence about the quality of their test suite in the presence of aspects and advice. A *minimal coverage criterion* might be that each adapted joinpoint is executed by at least one test. Note that we do not generate **CM** changes associated with **AM** changes (as *Chianti* does), as we use **CM** (and **CAB** changes as well) to explicitly model joinpoints adapted by advice.

Impact of `Billing` on tests `BasicSimulation` and `TimingSimulation`: This setup corresponds to the two version pairs $(1,2)$ and $(3,4)$ from Table 4.12 (i.e. we discuss impact on both applicable tests here). Our tool correctly detects that both tests are affected by the addition of the `Billing` aspect. Figure 4.24 (a) gives an overview of the atomic changes induced by the aspect. Aspect `Billing` results in 6 **AM** changes. Four of these six changes correspond to inter type declarations in in the `Billing` aspect adding necessary methods to the three `Connection` classes and the `Customer` class. Similarly to the `Timing` aspect this aspect also introduces two fields to classes `Customer` and `Connection` to store the amount charged for each phone call. As two of the new methods override the introduced abstract method `Connection.callRate()` we also experience two generated lookup changes in this case. The remaining two changes model the addition of two methods to aspect `Billing` itself.

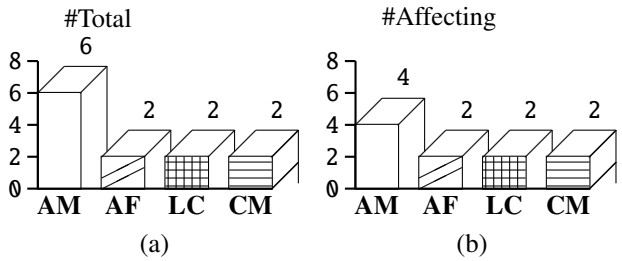


Figure 4.24: `Billing`-Aspect and associated numbers of total (figure (a)) and affecting (figure (b)) Atomic Changes in each Change Category.

Figure 4.24 (b) gives an overview of how many changes also affect tests. Note that only 4 of these 6 **AM** changes are actually covered when executing `BasicSimulation` and `TimingSimulation`. This is due to the fact that these two tests do not access any `Billing`-specific functionality and thus do not call all new aspect methods. Additionally the introduced method `Connection.callRate()` is abstract. Each test is affected by 10 changes, i.e. all affecting changes affect both tests.

The remaining two **CM** changes model potentially changed behavior due to the two pieces of advice `Billing` added to the system. Each change corresponds to one or more joinpoints adapted by one of the two pieces of advice. Note that the two **CM** changes are covered, as the two pieces of advice are executed even if the results of their calculations (the amount of money a customer is billed) is never accessed.

Impact of `Timing` and `Billing` on test `BasicSimulation`: This last setup corresponds to version pair $(0,2)$ from Table 4.12. In contrast to the first scenario above, we now removed the `TimerLog` aspect completely and then added both the `Timing` and the `Billing` aspect (thus the results are not just the addition of the above numbers).

Figure 4.25 (a) gives an overview of the atomic changes induced by the two aspects together. Figure 4.25 (b) gives an overview of how many changes also affect tests. As expected only the **AM** changes which were also uncovered in the previous scenarios remained uncovered.

Although we had no real test suite and only passing tests—and consequently no worsening test—in this case study, the programmer can nevertheless draw two

conclusions from these results. First, he can observe that each adapted joinpoint is executed by the tests, i.e. the programmer can have some confidence that addition of the `Timing` and `Billing` aspects will not break existing features of the `Telecom` system, as otherwise tests covering adapted joinpoints should fail. Second, the programmer can further see that not all functionality defined by the `Billing` aspect is already tested, as some unaffected **AM** changes remained. To summarize, by analyzing atomic changes and whether or not they affect a given test, programmers get an estimate if the current test suite covers aspects and further on which additional tests are needed.

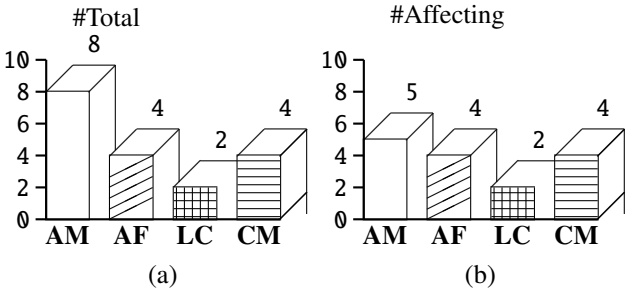


Figure 4.25: `Billing` and `Timing` aspects and associated numbers of total (a) and affecting (b) Atomic Changes.

abc-Tool Suite

The `Bean` Example: We compared the two versions with and without aspect `BoundPoint` with each other. As class `Demo` only compiles with applied aspect, we had to comment two lines from `Demo.main()`. Our analysis correctly determines that the single test case executing `Demo.main()` is affected by aspect application.

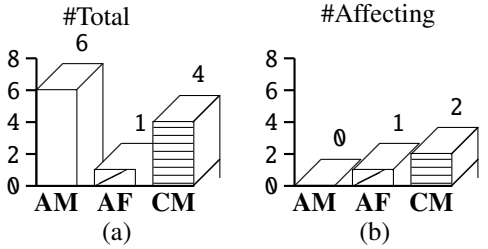


Figure 4.26: Total number of changes for the `Bean`-example (figure (a)) and of affecting changes (figure (b)).

Figure 4.26 gives an overview of the resulting changes. Note that the low coverage of the changes (as visible in Figure 4.26 (b)) is also due to modification of method `Demo.main()`, as we had to remove explicit calls to methods added by the aspect in order to make the version without aspects compilable.

As this example is very small, it is not very interesting from an analytical point of view. However `Bean` is an interesting performance test. Note that method `Demo.main()` contains a loop which executes 100000 times. Tracing this test case with JDI is already very expensive and produces a huge trace file (1.3 GB uncompressed). This demonstrates a weakness of our tracing approach, or better dynamic analysis in general, which we will discuss in detail in the next paragraph.

Examples based on `CertRevSim`: The `NullCheck`, `DCM`, and `LawOfDemeter` examples are all based on the `CertRevSim` application. Applying the aspects to this example application results in a huge runtime overhead (see Table 7.2 on page 218). Note that for `CertRevSim` the tests we defined were not unit tests, but rather had the characteristics of system regression tests as they resulted in complex calculations. Consequently even tracing the tests in the context of the base application results in huge trace files (3.7+ GB uncompressed). We failed to produce dynamic call graphs for these examples and were thus not able to analyze the impact of these aspects.

While this failure is unfortunate, it clearly shows the weakness of our current prototype and also our approach. Although the finally resulting call graphs for tests are relatively small (few KB on average), the size of the traces necessary to produce them can explode. We aborted

a trace for one of the tests for the `LawOfDemeter` example when the file had reached a size of more than 60 GB (uncompressed) and was still growing; runtime was also unacceptable for these tests (several hours). [14] Note that this is a general weakness of our approach, or better of dynamic analysis in general. While it is possible to avoid materialization of traces (for example by using the *JUnit/CIA* agent), the runtime overhead can be several levels of magnitude which is unacceptable especially for long program runs. Thus the limiting factor in our case is not static program size—all examples apart from `AJHotDraw` and `HSQLDB` are rather small—but execution time of tests. Unit tests which execute in split seconds can even be analyzed interactively. Suites which execute in several minutes however easily result in hours of runtime if they are traces. Not however that our method is targeted to analyze a unit test suite, as in this case the association of tests and changes is also most beneficial. From that point of view, the subjects we had to analyze are not optimal, as providing a meaningful test suite for an unknown system is not straight forward.

The `ProdLine` Example: We will discuss the results for running our tool on different build configurations of the `ProdLine` example. In contrast to the first examples in the *abc* suite discussed up to now, the single `ProdLine` test is a small unit tests and can thus be traced without problems.

We started with a minimal version only containing aspects `Graph`, `Benchmark`, and `NoPrinting`. We decided to apply these three aspects in the initial version of the program instead of using the raw base version without any aspects in order to be able to use the `main`-method in aspect `ProgTime` as a test to execute the system. Compared to V_0, V_1 adds aspects `Weighted`, `MSTKruskal`, and `MSTPrim`.

Figure 4.27 gives an overview of the resulting changes. The distribution of the changes shows that the `ProdLine`-example heavily relies on inter type declarations. All three aspects introduce a new method (the respective algorithm) and several fields which are also initialized, in part by intercepting joinpoints. Advice only results in 7 changes in total (represented by the **CM** and **CSI** changes). It is interesting to note that for this example all changes are actually covered by our single test case, which is also correctly identified as affected test. Consequently the advice has to contain code which executes introduced behavior. Examining the three aspects verified this assumption.

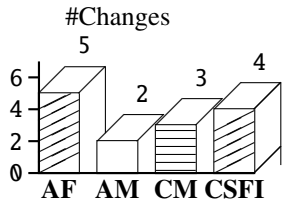


Figure 4.27: Changes for the `ProdLine`-example, $V_0 \rightarrow V_1$ (total and affecting changes, i.e there are no uncovered changes).

The (very specific) pieces of advice execute the respective introduced algorithm on the code and simply dump the result, without modifying any other system values.

Next we compare the minimal version V_0 (only aspects `Graph`, `Benchmark`, and `NoPrinting`) with version V_2 which adds aspects `CC`, `Cycle`, and `DFS`. Figure 4.28 gives an overview of the resulting changes. The distribution confirms the above assumption that the `ProdLine`-example heavily relies on inter type declarations. Again all three aspects introduce several new methods and fields, similar to the previous case study. Advice only adapts 5 joinpoints (represented by the **CM** and **CSI** changes) in total for this example. It is interesting to note that for this example again all changes are covered by our single test case, which is also correctly identified as affected test. This is again due to the patter used in `ProdLine`. New optional algorithms are introduced by inter type declarations, executed by advice (thus

[14] The next logic step to compress the traces failed due to a bug in the Java 1.4.2 API (Bug Id 5092263; the compression API only supports zipped files with an uncompressed size 2 GB; this is fixed for Java 1.5). The approach to split the trace in 2 GB chunks unfortunately also failed, as in this case reading the files triggered another Java API bug (Bug Id 4040920). We thus gave up for now and consider improvement of the tracer future work.

eliminating the necessity to add calls to this code in the base system), and finally the result is simply dumped to `System.out`.

We finally compare version V_0 with version V_3 which contains all aspects but aspects `ProdTime` (the profiling aspect we used as test case) and `Number`.

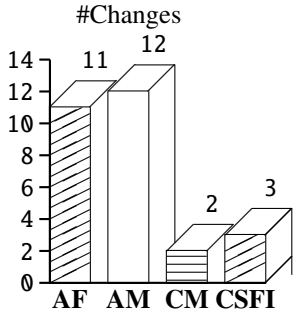


Figure 4.28: Overview of changes for the `ProdLine`-example, $V_0 \rightarrow V_2$ (total and affecting changes).

Figure 4.29 gives an overview of the resulting changes. As we now add two sets of disjoint aspects we previously analyzed in isolation, the number of changes now is the addition of the previously observed changes, as the observant reader may verify. Note this case study is useful to double-check if our implementation actually works as expected (summing up changes). More interesting however is the fact that we could easily uncover the pattern `ProdLine` uses to add new functionality, by examining change distribution and coverage. Thus the information provided by our tool supported us in understanding the system as it helped uncover the strategy internally used.

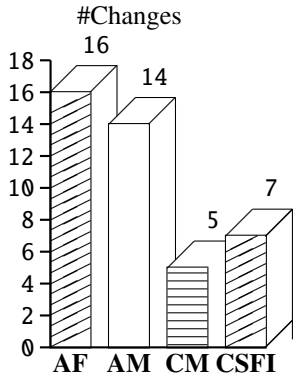


Figure 4.29: Overview of changes for the `ProdLine`-example, $V_0 \rightarrow V_3$ (total and affecting changes).

Adding the remaining two aspects `Number` and `ProdTime` then finally result in 2 **AF**, 3 **AM**, 2 **CM**, and 1 **CSFI** change. However, in contrast to all other examples in this case two untested changes (**AM** `ProdTime.main()` and **CM** `ProdTime.main()`) remain, as the `main()`-method of the aspect is not executed.

The `ProdLine`-example uses mainly inter type declarations to create the different possible instantiations of the system. It is interesting to note that—although there are many inter type declarations in the program—lookup changes do *not* occur. This strengthens our assumption (see Chapter 3) that lookup changes in the context of inter type declarations are likely accidental and not on purpose and thus should be reported by supporting tools.

The `Tetris` Example: As last program of the *abc* program suite we compare several versions of the `Tetris` program derived by using different build configurations. Although all aspects for `Tetris` are optional, several aspects semantically depend on each other. We thus chose to incrementally extend the set of applied aspects with respect to these dependences.

We start with comparing the vanilla `Tetris` base program without any aspects applied as version V_0 and the program version with two development aspects (`TestAspect` and `DesignCheck`) applied as version V_1. Addition of these two aspects only results in 2 **CM** changes due to a single pieces of advice. Aspect `DesignCheck` only contains two `declare warning` statements, and aspect `TestAspect` defines a single piece of `before`-advice, resulting in the 2 above mentioned **CM** changes. Nevertheless all four tests we defined for `Tetris` by using the *replay* feature are affected.

We continue by comparing version V_1 with version V_2 which is created by adding aspects `GameInfo` (which adds an info panel used by other aspects) and `Menu` (which adds

the game menu). These two aspects result in 1 **AM** and 3 **CM** changes.[15] All four tests are affected by adding the new aspects. While all three **CM** changes are covered, the **AM**(`actionPerformed(..)`) change remains uncovered. This is due to the fact that our tests do not access the menu bar and so uncovers a weakness of our test suite.

Version V_3 adds aspects `Counter` and `Levels` to `Tetris`. `Counter` keeps track of the number of deleted lines and `Levels` uses this information to increase the level accordingly. Addition of these two pieces of advice results in 2 **CAB** changes (changed advice) and 10 **CM** changes. Again the defined advice is very selective. The two aspects in total define 10 pieces of advice, and a single piece of advice at most matches 3 joinpoints. For this example, all test are affected and all changes are covered by the tests.

Finally V_4 is the version resulting from application of *all* available aspects, including the two `Logic` aspects `NewBlocks` and `NextBlock`. Addition of these last two aspects results in 4 **CAB** and 7 **CM** changes. Again, all four tests are affected by these changes. Our prototype reports that 3 **CM** changes (i.e. changes for one method, but three different pieces of advice) are not covered by our tests. As this was surprising for us we double checked this case by adding `print`-statements to the respective method and confirmed that the respective changes were indeed not executed. The reason in this case is that aspect `NextBlock` contains `around`-advice which intercepts calls to `AspectTetris.getRandomBlock()` and does not call `proceed()`.

In a final case study we also compared versions V_0 and V_4 with each other. Figure 4.30 (a) summarizes all resulting changes, Figure 4.30 (b) gives an overview of the changes covered by our test suite. Note that there is only a single (uncovered) **AM** change, but multiple **CAB** and **CM** changes. The **CAB** changes show that `Tetris` provides a layered set of aspects, where aspects not only extend the base system, but also build on aspects and extend these aspects as well.

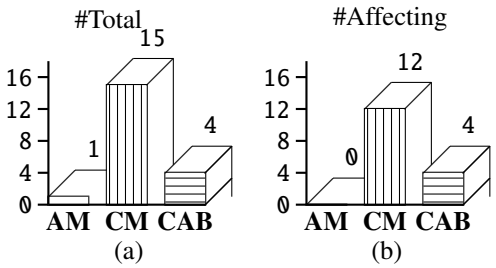


Figure 4.30: Total number of changes for the `Tetris` example (versions V_0 and V_4), (figure (a)) and of affecting changes (figure (b)).

To evaluate our aspect impact analysis, `Tetris` is not very helpful, as the information we can derive is similar to the information *ajdt* provides. This is however due to the coarse grained test suite and the fact that all tests in general are affected, i.e. we cannot give more specific information which functionality (represented by tests) is affected by the aspects. Similarly, the set of affecting changes is nearly the same for all tests.

Our analysis however gives—compared to *ajdt*—additional information on the quality of the tests in terms of coverage. We saw that the test cases we created were not sufficient to test all features added by the aspects, as the changes due to addition of the `Menu` aspect remained uncovered. As we intended to create a comprehensive test suite by using all features of `Tetris`, this indeed uncovered a feature we missed to test.

Besides the above, the `Tetris` program is interesting to study, as the aspects implement a set of optional featured which can be added to a system, similarly to the `ProdLine`-example. However, in contrast to `ProdLine`, `Tetris` does not use inter type declarations but instead uses very specific advice only affecting few joinpoints to modify and extend system behavior according to its needs. Consequently in this case study we see only one **AM** change which is due to an aspect-internal method (and not due to an inter type declaration).

Note that there is a qualitative difference in those two implementations. `ProdLine` injects

[15] Our prototype actually reports a **AM**, two **CM**, and a single **CC** change (for "change class"). The latter however only indicates the change of the generated default constructor. We will express such **CC** (and similarly **CAS** changes (change aspect)) as **CM** changes in the following.

new functionality in existing classes, and calls this functionality using advice. `Tetris` in contrast does not extend existing classes, but keeps all added functionality local to the aspect.

Auction System

We analyzed each single removable aspect of the auction system (cmp. Section 7.1.3) separately, by comparing a configuration with all aspects but the aspect to analyze with the configuration where all aspects are applied. This approach resulted in 8 different build configurations. It is interesting to note that all aspects except aspect `TestAspects` are rather specific, i.e. they only affect few joinpoints. Only the tracing aspect `TestAspects` affects a large amount of joinpoints, resulting in a total of 209 **CM** changes and a single **CSFI** change.

Aspect `AuctionUpdateNotificationAspect` results in 2 **AM** and 4 **CM** changes. These changes affect only a single of the 6 tests, test `testClose()`, which is also only affected by a single change. Analysis of the remaining changes in more detail showed that these changes can be related to dead code and method `joinAuctionForm`, which is not executed by the test suite. Aspect `CreditLog` results in 6 **CM** changes, but only one of these 6 changes affected the single test `testClose()`. All other tests remain unaffected, the remaining changes are uncovered. Again the low coverage of the test suite is responsible. Aspect `RefreshAuctionStateAspect` results in 3 **CM** changes. Only one of these changes affects two tests of the test suite. The remaining tests are unaffected, the other two changes uncovered.

A special case is aspect `SerializationAspect`. This aspect only contains a `declare parents ...implements` statement, but no methods or advice. Consequently there are no atomic changes and also no affected tests due to this aspect. This actually shows a weakness of our approach, as some tests crash due to missing serialization support although our method cannot report any changes. This is however a special case of the Java API—`Serializable` is a pure marker interface only used to mark classes a serializable, it contains no methods or constants. The only purpose of the interface is thus the type change, which is not covered by our analysis.

Aspect `DataManagementAspect` results in 2 **AM** and 2 **CM** changes. However, this aspect seems to affect system core functionality, as all tests are affected and all changes are covered by each single test. Aspect `MaintainAuctionIDAspect` results in 3 **AM** and 5 **CM** change. Two tests are affected by the 3 **AM** changes and 2 of the 5 **CM** changes. The remaining 4 tests are unaffected, 3 **CM** changes are uncovered. Finally aspect `ValidateUserAspect` results in a single **AM** changes, which does not affect any test. Analysis of the aspect showed that the relevant aspect method is unused. The calling advice in this aspect has been commented out, i.e. this is dead code.

To summarize this case study showed the weakness of the current test suite, as only a fraction of all changes resulting from the aspects is covered by our small test suite. By analyzing each aspect separately we got a good overview and understanding of how the aspects worked and which functionality they affected. For this case study we saw that most aspects were indeed rather specific and thus could be linked to only a subset of the tests. As our test suite did not use all the implemented functionality, it was not surprising that some untested changes remained. Note that this case study is also an example that our analysis is unsafe due to not covering type changes. This became apparent with the marker interface `Serializable`. While we are aware of this restriction and also discussed it in the theoretical part of this thesis, this is the only example we found where this restriction actually was relevant.

`AJHotDraw`

We analyzed version 0.2 and 0.3 of the `AJHotDraw` system using our prototype. Although version 0.1 introduces the invariants checking aspects in the test suite, we analyzed their impact in the context of version 0.2 for the sake of brevity. As stated in Chapter 7, version 0.2 of the

Test	AJHotDraw 0.2 less	build	impact	$\|AC(T)\|$	no_inv	impact	$\|AC(T)\|$
`TextFigureStorableTest`							
`testWriteRead()`	FAIL	PASS	✓	33	PASS	✓	8
`testWriteReadStorable()`	PASS	PASS	✓	33	PASS	✓	8
`testAlphaOmegaWrite()`	PASS	⊥	✗	n.a.	PASS	✗	n.a.
`ImageFigureStorableTest`							
`testWriteRead()`	CRASH	PASS	✓	15	PASS	✓	8
`testWriteReadStorable()`	PASS	PASS	✓	15	PASS	✓	8
`testAlphaOmegaWrite()`	PASS	⊥	✗	n.a.	PASS	✗	n.a.
`AttributeFigureStorableTest`							
`testWriteRead()`	FAIL	PASS	✓	33	PASS	✓	8
`testWriteReadStorable()`	PASS	PASS	✓	33	PASS	✓	8
`testAlphaOmegaWrite()`	PASS	⊥	✗	n.a.	PASS	✗	n.a.
`StandardDrawingStorableTest`							
`testWriteRead()`	FAIL	PASS	✓	44	PASS	✓	14
`testWriteReadStorable()`	PASS	PASS	✓	44	PASS	✓	14
`testAlphaOmegaWrite()`	PASS	⊥	✗	n.a.	PASS	✗	n.a.
`InvariantCheckTest`							
`testConstructor()`	FAIL	FAIL	✓	1	FAIL	✗	n.a.
`testCtorWithPublicCall()`	FAIL	FAIL	✓	1	FAIL	✗	n.a.
`testPublicCall()`	FAIL	FAIL	✓	1	FAIL	✗	n.a.
`testPkgCall()`	FAIL	FAIL	✓	1	FAIL	✗	n.a.

Table 4.13: JUnit test suite of `AJHotDraw`, results in different versions.

system aspectized figure persistence; version 0.3 additionally aspectized contract enforcement and an instance of the observer pattern.

`AJHotDraw` 0.2: To run our change impact analysis tool, we ran the available JUnit test suite with 16 JUnit tests on a version of `AJHotDraw` without any aspect applied, and on a version where four invariant checking aspects and the six persistence aspects were applied.

For these two versions, we executed the `AJHotDraw` JUnit tests. Table 4.13 gives an overview of all tests of this test suite including their results in the different program versions, and also the number of affecting changes per test. As can be seen, removal of the aspects causes several of the tests to FAIL, one test even to CRASH. This indicates that at least some of the removed aspects are not optional but provide necessary functionality for system correctness. Note however that not all tests are affected by the aspect-induced changes. Four tests (all `testAlphaOmegaWrite()` tests) do not execute any joinpoint adapted by advice (these tests also pass in both versions).

The aspects results in the set of changes shown in Figure 4.31 (a). Note the extremely high number of **CM** changes. This is due to the broad application of the invariants aspect (a single piece of advice adapts 326 different joinpoints). The remaining changes are due to addition of aspect methods (**AM** changes) and due to the inter type declaration of `read()` and `write()` methods on four figure classes by the persistence aspects. These methods have been removed from the base system previously and are now re-added to the base system by using inter type declarations. Note that in this example we also experienced several lookup changes, as the persistence aspects heavily rely on inter type declarations. As `AJHotDraw` does not compile when all persistence aspects are removed, build configuration *less* left an aspect in the system

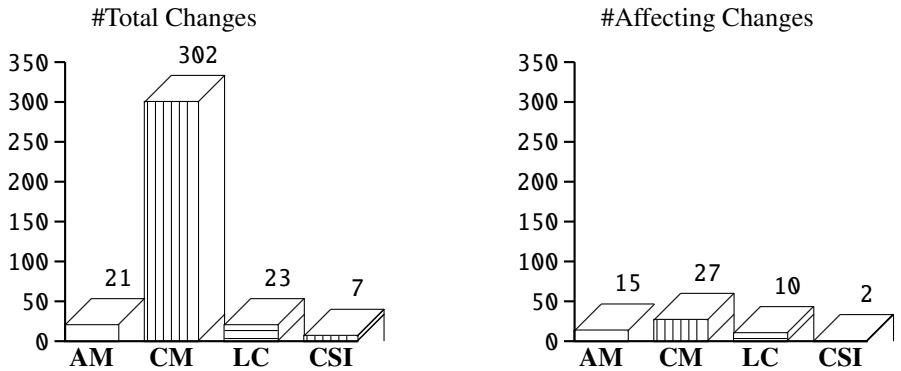


Figure 4.31: Changes for comparing configurations *less* and *build* for `AJHotDraw` 0.2.

which introduced `read()` and `write()` methods to class `AbstractFigure`, the base class of all figure classes. The introduction of figure-specific `read()` and `write()` methods then resulted in the observed lookup changes. Although only 8 methods are introduced in total, we see 20 lookup changes, as some of the four target classes also have subclasses which suffer from a changed lookup as well. The lack of these methods in configuration *less* is also the reason for the test failures.

Figure 4.31 (b) shows the set of changes actually covered be the test suite we executed. As can be seen, most **CM** changes remained uncovered. These are in general the changes due to the invariant checking aspects (the relevant code is simply not executed by the 16 tests provided by `AJHotDraw`; statement coverage was only 8%). However there are also 6 **AM** changes which are not covered. These changes are also relates to the invariants checking aspect.

As the number of changes resulting from aspect application in this example grows, we also added the number of affecting changes to Table 4.31. As can be seen, each persistence aspect is affected by at least 15 and at most 44 changes. Examining these changes in detail is already an increased effort, but still feasible. It is interesting to note that only a single change affects each invariants checking test. We investigated this small number of affecting changes further. The solution is rather simple—test `InvariantCheckTest` is not a test to check invariants in the system, as we assumed, but rather a test checking if the aspect itself works as intended. This test thus does only executed test classes, but no system code.

To get a better impression of the modifications due to the persistence concern, we also analyzed the effects of only applying the persistence aspects to the system, without also applying the invariants checking aspect. We again executed the test suite and this time the `testAlphaOmegaWrite` tests as well as the invariant tests were not affected by the aspects. The persistence tests were affected by 4 (or 6) **AM** and 4 (or 8) **LC** changes (depending on the respective test). All **AM** changes resulting from adding the persistence aspects affect at least one test. However, 10 of the **LC** changes remained uncovered, as can be seen by comparing Figures 4.32 (a) and 4.32 (b).

Note that all **LC** changes are actually correct, as the `AbstractFigure` class has a rather complex inheritance hierarchy. Note that some subclasses of `AbstractFigure` still have their own implementation of methods `read` and `write`, i.e. the refactoring of persistence is incomplete. However, only few figure classes (where the refactoring has been performed) are actually accessed be the tests.

`AJHotDraw` 0.3: Similar to the previous example we ran the available JUnit test suite on a version of `AJHotDraw` without any new aspect (but with the three invariant checking

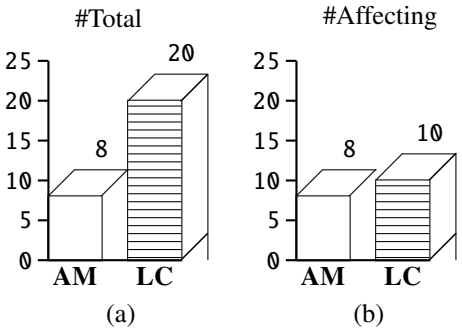


Figure 4.32: Changes for comparing configurations *less* and *no_inv* for `AJHotDraw` 0.2.

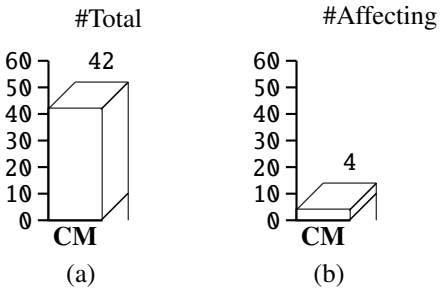


Figure 4.33: Changes for comparing configurations *no_new* and *build* for `AJHotDraw` 0.3.

aspects and the four persistence aspects) applied, and on a version including the new aspects, i.e. the concern checking aspect `CmdCheckViewRef` and the observer notification aspect `SelectionChangedNotification`.

For these two versions, we executed the `AJHotDraw` test suite now consisting of 21 JUnit tests. Table 4.14 gives an overview of all tests of this test suite including their results in the different program versions.

As can be seen, removal of the aspects causes several of the new tests to FAIL, one test even to CRASH. This indicates that at least some of the removed aspects are not optional but provide necessary functionality for system correctness.

The two new aspects result in the changes shown in Figure 4.33. Note that 38 of the **CM** changes are due to addition of aspect `CmdCheckViewRef`. However, only one of these changes is actually covered by the 21 tests of the test suite. Aspect `SelectionChangedNotification` induces the four remaining changes. The test suite covers three of these four changes.

The analysis of `AJHotDraw` shows two things.

- First, the set of changes derived from the aspect gave us a good impression which aspects affect which parts of the system. This is basically the information which is also available via the *ajdt* tool kit, however our analysis in the spirit of *Chianti* also links tests to changes. So we can additionally state which test is affected by which aspects. If we assume that each test has some semantical purpose, we can now derive which functionality is affected by which aspects. In this example, the `testReadWrite()`-tests were in general affected by the *respective* persistence aspects.

- Second, we got a good impression which aspect effects (in terms of adapted joinpoints) have in fact been covered by a test suite. As 12 of the 16 tests we analyzed are related to persistence, we assumed that these aspects are in fact well tested as our analysis confirmed. The invariants checking aspect however seems to be basically untested (at least by the 16 tests we examined), i.e. most **CM** changes remained untested. Finally our analysis also showed that the four `testAlphaOmegaWrite` tests did not execute the respective `read` and `write` methods in the different figure classes. Investigating this finding showed that this test is actually an inherited test and truly identical for all four test classes (no overridden functionality or data is accessed).

HSQLDB

In our final case study we analyze impact of aspects in the final version of the refactored `HSQLDB` system (see 7.2 on page 224). As `HSQLDB` is associated with a "real" JUnit test suite

Test	AJHotDraw 0.3 less	build	impact	$\|AC(T)\|$
TextFigureStorableTest				
testWriteRead()	PASS	⊥	✗	n.a.
testWriteReadStorable()	PASS	⊥	✗	n.a.
testAlphaOmegaWrite()	PASS	⊥	✗	n.a.
ImageFigureStorableTest				
testWriteRead()	PASS	⊥	✗	n.a.
testWriteReadStorable()	PASS	⊥	✗	n.a.
testAlphaOmegaWrite()	PASS	⊥	✗	n.a.
AttributeFigureStorableTest				
testWriteRead()	PASS	⊥	✗	n.a.
testWriteReadStorable()	PASS	⊥	✗	n.a.
testAlphaOmegaWrite()	PASS	⊥	✗	n.a.
StandardDrawingStorableTest				
testWriteRead()	PASS	⊥	✗	n.a.
testWriteReadStorable()	PASS	⊥	✗	n.a.
testAlphaOmegaWrite()	PASS	⊥	✗	n.a.
InvariantCheckTest				
testConstructor()	FAIL	⊥	✗	n.a.
testConstructorWithPublicCall()	FAIL	⊥	✗	n.a.
testPublicCall()	FAIL	⊥	✗	n.a.
testPkgCall()	FAIL	⊥	✗	n.a.
testCommandExecute()				
ChangeAttributeCommandTest	CRASH	PASS	✓	3
InsertImageCommandTest	FAIL	⊥	✗	n.a.
PasteCommandTest	FAIL	FAIL	✓	1
UndoCommandTest	FAIL	PASS	✓	2
UndoableCommandTest	FAIL	PASS	✓	2

Table 4.14: JUnit test suite of AJHotDraw, results in different versions.

(in contrast to the constructed tests for *abc*) and has a reasonable size (60 kLoC), it is a good candidate to evaluate our tool in a setting close to its intended usage.

Not all of the crosscutting concerns implemented as aspects in HSQLDB are optional. However, the system compiles without the classic development aspects Tracing and Profiling. The optimization aspect ValuePooling and enforcement of the *Swing thread safety* rule are also optional. We were also able to remove the general exception handling aspect *log and forget* without provoking compiler errors. Note that several of these concerns are implemented by multiple aspects. In the following we will now analyze the impact of each of these five concerns on the base system—i.e. the final HSQLDB version without the respective aspects applied—individually.

Impact of Tracing: The Tracing aspect is *the* classical optional orthogonal homogeneous crosscutting development concern. As such, it in general affects many parts of the system to analyze, however only as an observer, without modifying the I/O behavior of the system. To analyze the impact of the tracing aspect, we compared a build configuration where we removed tracing with the complete build configuration of our final version of HSQLDB.

The tracing aspect resulted in a total of 117 changes. Three changes are **AM** changes due to helper methods in aspect TracingAbstractAspect. The remaining changes are

Changes per Test	Tracing	Profiling	ValuePooling	Thread Safety	Exception Handling	Summary
Min	3	5	22	0	135	147
Max	42	6	61	0	536	585
Average	12.6	5.9	33.3	n.a.	299.3	315.4
Std. Derivation	10.6	0.25	11.3	n.a	134.3	140.0

Table 4.15: Overview Affecting Changes per Test for HSQLDB

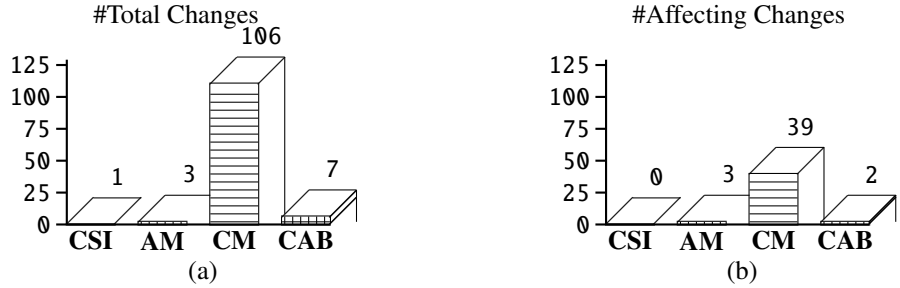


Figure 4.34: Changes for comparing configurations *without tracing* and the final HSQLDB version, (a) all changes, (b) affecting changes.

106 **CM** changes, 7 **CAB** changes and 1 **CSI** change due to joinpoints affected by tracing advice. Note that the tracing aspects however affect considerably more joinpoints (`TracingAbstractAspect` alone contains advice affecting 193 different joinpoints). However, in many cases several of these joinpoints are contained in a single method, so resulting in only one **CM** (or **CAB** etc.) change. Due to the method-level granularity we are thus able to abstract from the individual joinpoints.

The tracing aspect affects all tests but test `testHypersonic`. For the 38 affected tests, a minimum of 3 and a maximum of 42 changes affected each single test, with 12.6 changes on average and a standard derivation of 10.6 (see Table 4.15). Theses numbers show that it is feasible for a programmer to examine the impact of tracing on a given test (for example if this test fails), as a maximum of 42 changes can be examined manually. As none of the 38 affected tests changes the outcome by adding tracing, we however have some confidence—although no guarantee—that the tracing aspect does not change semantics of HSQLDB.

Note that a considerable amount of all changes—nearly two thirds—remains uncovered. Consequently we do not know if tracing actually works as expected in untested context. This again shows that the test suite of HSQLDB has only insufficient coverage. Coverage is especially low for the **CAB** changes (2/7). This can however be easily explained, as the uncovered pieces of advice are concerned with exception handling. However, the test cases simply do not throw these exceptions. As testing exception handlers is problematic in general this is not really surprising.

The change distribution for the tracing concern gives a good impression about the nature of the underlying aspect. We only see three **AM** changes, but more than 100 changes due to modified joinpoints, which affect nearly the whole system. Consequently, the tracing aspects heavily rely on advice, which is applied to large parts of the system. Examining the tracing aspects verifies this first impression.

Impact of `Profiling`: The `Profiling` aspect is very similar to the tracing aspect, although for this particular case study considerably more complex (comparable to `Logging`). As for `Tracing`, we expect many affected joinpoints, but no modified I/O behavior. However, the results of our experiment not quite matched these expectations. While semantics (as far as we can tell) remained unchanged, the profiling aspect was considerably more specific as we expected.

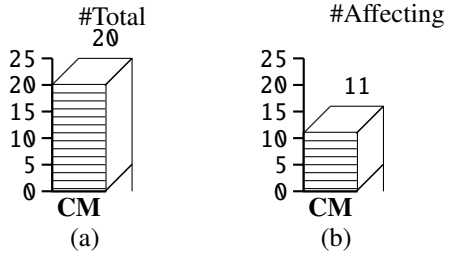


Figure 4.35: Changes for comparing configurations *no_profiling* and *build* for `HSQLDB`, (a) all changes, (b) affecting changes.

Similarly to the tracing experiment above, we compared a configuration of the final `HSQLDB` version where we removed the profiling aspects with the final `HSQLDB` version containing all aspects. Adding the profiling aspect in total results in 20 **CM** changes only, resulting from 30 affected joinpoints.

The profiling aspect only affects 16 of the 39 tests, i.e. 23 tests are not affected by the profiling aspect. This is actually surprising, as we expected it to be applied to more joinpoints. However analyzing the aspect in detail confirmed the results of our tool. Profiling in `HSQLDB` is rather specific. Although not even half the test suite is affected by profiling, only 55% of all changes (11 of 20) are covered by the test suite. For the affected tests, the number of affecting changes is between 5 and 6, so analyzing the impact of the profiling aspects on a given test can be done very efficiently for each affected test.

If we consider both the low change coverage and the high number of unaffected tests we can conclude that profiling primarily affects functionality which is not covered by the test suite of `HSQLDB`. Investigating the profiling aspect showed that profiling only affects tests which are subclasses of `TestBase`, which explicitly use the `HSQLDB Server` class. Profiling only affects database communication via this class, while other tests use other mechanisms to access the database, for example JDBC, and are thus unaffected.

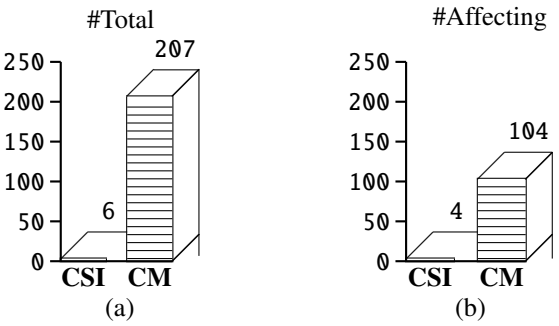


Figure 4.36: Changes for comparing configurations *without value pooling* and the final `HSQLDB` version, (a) all changes, (b) affecting changes.

Impact of `ValuePooling`: As the `ValuePooling` aspect is an optimization aspect, we do not expect any differences in the I/O behavior of the program, however we expect an impact on system runtime. This is however not in the scope of our analysis.

Adding aspect `ValuePooling` results in 213 changes, again most of them (207) **CM** changes. Figure 4.36 gives a detailed overview. As seen in the previous examples, coverage if these changes is again rather low, only 50.7% of all changes are actually covered by the test suite. In contrast to the precious examples, aspect `ValuePooling` however affects all tests of the test suite.

The number of changes affecting a single test is however still accessible for humans, with a minimum number of 22, a maximum number of 61 changes affecting a single test, and on average 33.3 changes per test for a standard derivation of 11.3 (see Table 4.15).

What we can see in this example is that value pooling is a truly crosscutting concern.

It affects more joinpoints—in this case always creation sites—than the tracing or profiling aspects we analyzed before. It is thus not surprising that the simple caching of immutable objects has significant impact on program performance (improvement between 5% and 15%, depending on the operation), as some benchmarks to evaluate performance of pooling showed [15].

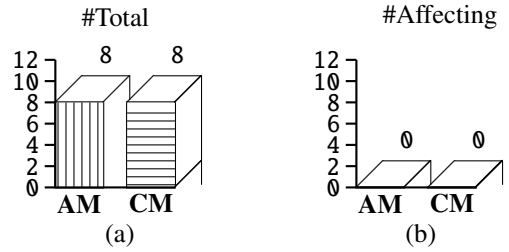


Figure 4.37: Changes for comparing configurations *without value pooling* and the final HSQLDB version, (a) all changes, (b) affecting changes.

Impact of *Swing Thread Safety*: Although Swing thread safety is optional in terms of compiler errors, it is not optional in terms of system correctness. Without this aspect, Swing's *single thread rule* is no longer enforced, potentially resulting in anomalies in the GUI (lost updates, GUI lockups, etc. due to race conditions).

In contrast to the aspects we analyzed up to now, the swing thread safety aspect is rather specific. Adding the aspect results in 8 **AM** and 8 **CM** changes. This is due to the fact that Laddad [33] provides a minimal pointcut in his book which captures all relevant joinpoints (calls to the swing API) to be wrapped in worker objects. This pointcut has been used during the refactoring of HSQLDB.

While pooling affected all tests, we now see the exact opposite—the swing thread safety aspect affects not even a single test. Consequently we also have no affecting changes. We also see that no change is covered by any test case (see Figure 4.37). Explaining this fact is rather simple. The swing thread safety aspect is an explicit GUI aspect. However the tests do not reference let alone test the GUI.

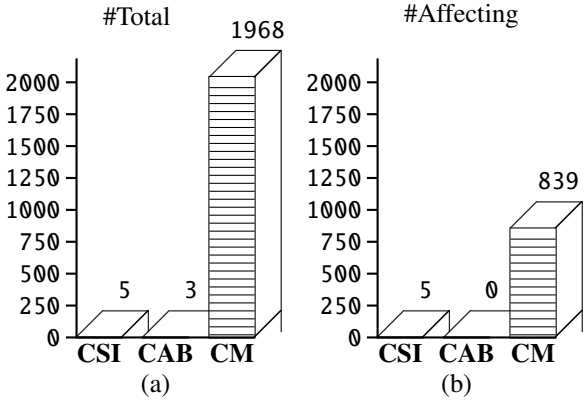


Figure 4.38: Changes for comparing configurations *without log'n'forget exception handling* and the final HSQLDB version, (a) all changes, (b) affecting changes.

Impact of *Exception Handling*: The general log and forget exception handling concern also applies to many joinpoints in the system. However, its removal will only result in lost exception messages (they are no longer logged), but not modify the control flow (exceptions are nevertheless caught by the respective handler).

Figure 4.38 gives an overview of all the changes resulting from the application of the very general *log and forget* error handling concern. This exception handling policy is obviously very common in HSQLDB, as the simple log and forget policy is applied to 1968 different methods in the system.

This very general concern affects all tests of the test suite. For the number of affecting changes per test, we see a minimum of 135, a maximum of 536, and 299.3 changes on average, with a standard derivation of 134.3 (see Table 4.15). Due to the low coverage of the HSQLDB test suite we however also see a considerable amount of uncovered changes. Similar to the

previous examples only 43% of all changes derived from the aspect are covered by the test suite.

For this example understanding aspect impact based on individual changes is clearly a hard task, as nearly every class and every method is affected. Analyzing 300 changes on average per test is clearly no longer feasible to do manually. In case of such general aspects, we instead used a different more efficient approach. A closer look at the relevant pieces of advice reveals that most joinpoint matches (2352) are due to only two pieces of advice, a piece of `before`- and a piece of `after`-advice. The remaining two pieces of advice in the exception handling aspect only adapt 35 joinpoints and are not directly related with exception handling but handle dumping of logged messages. The two relevant pieces of advice only access one parameter of their respective joinpoint context—the (caught) exceptions—which is then logged. As the advice is only logging data but not modifying it, we can deduce that the two pieces of advice do not modify I/O behavior.

Impact Analysis Summary: In a final experiment we analyzed the impact of adding all of the above aspects at once. For this experiment, all 39 unit test were affected, and our analysis took close to an hour (53:20), i.e. on average tracing each test with TPTP took 82 s.[16] While this does not seem to be very much (especially as this includes one large test comprising all batch tests), keep in mind that the whole JUnit test suite usually executes (untraced) in approximately 4 s.

Adding all aspects at once resulted in an impressive number of changes, Figure 4.39 gives an overview. Note that most changes are changes due to adapted joinpoints (**CM**, **CAB** or **CSFI** changes), and we only have 11 **AM** changes for this case study. This seems to be a characteristics of homogeneous aspects, as they are usually uniformly applied to many joinpoints. This behavior also meets the expectations we formulated earlier. Due to this fact all 39 tests were affected by the added aspects, and the number of changes affecting a given test is also very high—from 145 to 605 changes per test (see Table 4.15). Most of these changes are however due to the exception handling aspect. This high number of affecting changes demonstrates the importance of a well-structured presentation to the user, as otherwise the presented information is very hard to use. To present changes hierarchical per aspect is in this context important, as this allows to abstract from the exception handling aspect and to concentrate on other relevant changes.

Although the coverage of the `HSQLDB` test suite is relatively high compared to other Open Source systems (48% statement and 51% branch coverage), we nevertheless experience a high number of uncovered changes. In terms of covered changes, we only have a coverage of of 42%. This is clearly not satisfactory, i.e. our tool again demonstrated that the test suit is not sufficient and—more specifically—where and for which aspects coverage has to be increased.

We conducted this last experiment mainly for completeness reasons and also as a scalability test for our tool. While the tracing overhead is similar in the above cases, the large amount of changes is most interesting in this case. While the tool had no problems to deal with the large amount of changes, we noticed that for humans the information has to be categorized better, and optimized our user interface accordingly, to present changes in a hierarchical manner.

Summary and Discussion: We claim that our tool can help the programmer in determining the impact of the addition of a certain aspect to a given system. In the `HSQLDB` case study we saw that lifting the change impact analysis of *Chianti* can indeed provide considerable support for programmers in understanding the effects of an aspect, as we translate aspect impact in atomic changes for the program elements containing adapted joinpoints.

[16]Note however that this also includes a JUnit test which executes the whole batch test suite, a suite which is considerably larger than the remaining 38 tests together, but counted as a single test. Thus the average for the 38 unit tests is considerably lower.

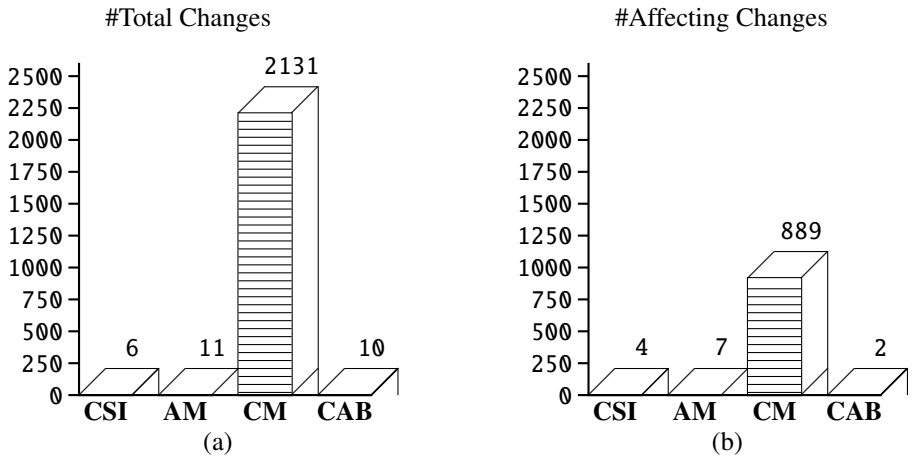


Figure 4.39: Changes for comparing configurations *no_aspects* and *build* for HSQLDB, (a) all changes, (b) affecting changes.

Furthermore—similarly to *Chianti*—we can associate aspect impact with tests, thus also giving a link to affected requirements, if we assume that tests can be related to a certain functionality to be tested. Our case study also showed that this works pretty well.

A very important property of our analysis results is that the programmer gets an overview of which changes have *not* been covered by any test. This means we show *in which context a certain aspect* remains untested. For the aspect, coverage criteria like branch or statement coverage are not sufficient. It seems a better idea to demand that each adapted joinpoint has to be executed at least once in order to call an individual aspect *covered* by a test suite. For HSQLDB we have seen that, for several configurations we examined, a lot of uncovered changes remained. For these changes the test suite has to be improved. Note however that change coverage is approximately in line with branch and statement coverage for the whole system.

We hoped that by using the development history of HSQLDB we would also be able to demonstrate the usefulness of our analysis results for debugging, in case of failing tests. While we have versions of HSQLDB where tests fail, this failure is however due to refactoring of the base code, and not primarily due to the addition of an aspect, and thus beyond the reach of our current tool. We were thus not able to demonstrate the usefulness of change coloring in the spirit of *JUnit/CIA* for aspects. However, we refer to the results for plain Java and argue that we are not aware of any reason why it shouldn't be possible to apply this technique in the context of AspectJ.

Threats to Validity: Note that all aspects we could analyze in the HSQLDB case study have two common properties: they are optional (at least in terms of syntactical correctness), and they are homogeneous aspects. It can thus be argued that the results we got for this case study are not applicable in general.

Note that the first property shows a restriction of the applicability of our tool. As our analysis requires call graphs and as we obtain these graphs using tracing, we have to be able to execute the system. This implies that the system is compilable. Consequently, we are currently only able to analyze the impact of aspects which can be removed from the system without provoking compiler errors.

However, as aspects have been promoted as *pluggable functionality*, this restriction is likely to be met when applying an aspect to a system. Even if an aspect is not easily removable, there is in general a predecessor version without the aspect which is compilable. Adding the aspect might have resulted in associated base code changes which made its removal im-

possible. While our current tool cannot compare these two versions, we already described in this chapter how it can be extended to deal with such a scenario. I.e. the described restriction is a restriction of our current implementation, not of our approach in general.

The fact that we could only analyze the effect of homogeneous aspects is a side effect of the implementation strategy used during the refactoring of `HSQLDB`. If an aspect was specific for a certain module, this aspect has been implemented as an inner aspect. AspectJ build files however only allow to remove files from a build configuration. As moving inner aspects to a separate file is a non-trivial refactoring, we refrained from doing so and restricted ourselves to the analysis of those aspects found in separate files.

Finally, this case study analyzed code which we refactored to aspects ourselves. While different people actually did the refactoring without taking any preconditions to make it usable as a subject for a case study (beside choosing a reasonable sized Open Source Java project associated with a test suite), it is nevertheless possible that the results of this case study are biased by this fact. However, we did our best to avoid unwanted influence.

4.5.9 Performance and Scalability

Discussing the runtime overhead of our techniques is an important facet to judge the usability of our tool in everyday work. From that perspective, our change impact analysis tool unfortunately is not yet ready to be used by practitioners as our method suffers from a considerable tracing overhead. Note however that this is a common restriction of dynamic analysis techniques, and not specific to our method. This section discusses performance of our tracers in more detail. Table 4.16 gives an overview of the runtimes of the different case studies we conducted using the different available tracers.[17]

The huge runtime overhead outlined by the numbers in the table is clearly unacceptable for interactive use of our tool. Analysis of the runtime behavior shows that nearly all the time[18] is spent in the construction of the dynamic call graphs. This can however be considerably improved. We experimented with JDI (Java Debugging Interface), the *JUnit/CIA* agent and the TPTP agent to trace test execution. While JDI allows us to easily access all relevant data (especially the runtime type for each object necessary to correctly match **LC** changes), it is probably the worst choice in terms of runtime.

We also experimented with the TPTP tracing agent, the agent provided by the Eclipse Test & Performance Tools Platform.[19] However, while TPTP allows to trace tests in a fraction of the time JDI needs (for example for `Bean`, TPTP only needs 4m 43s to produce the trace, compared to over 50 minutes with JDI), TPTP does not allow to access the runtime type of callee objects, which consequently does not allow to match **LC** changes correctly. For those versions where no runtime changes occur this might however be a more efficient alternative. Additionally TPTP is not able to produce trace files larger than 2 GB (Bug Id 149812), which considerably reduces its applicability. Nevertheless we consider TPTP as a lower bound for a JVMPI-based approach with materialized traces.

A third option is the JVMPI agent used by *JUnit/CIA*. Note that this agent in contrast to the two other tracers discussed before is not a tracer, but directly produces dynamic call graphs, thus avoiding the expensive materialization of large trace files. From our experiments we can conclude that it has in general the same magnitude of runtime compared to TPTP, but allows to access the necessary runtime type. Consequently this approach would be preferable to TPTP for our setup. Unfortunately the agent produced truncated call graphs for some of our tests, presumably due to our filtering approach and the renaming schema for inter type declarations

[17] Note that these are simple benchmark numbers taken by measuring the real execution time on a machine with an AMD Athlon 64 3200+ CPU and 2 GB of RAM running on Windows XP. Their only purpose is to show the magnitude of runtime we currently have to deal with.

[18] Depending on the number and runtime of the traced tests, for `Bean` for example more than 99.9%

[19] `http://www.eclipse.org/tptp/`

Versions	#Tests	JDI	TPTP	JUnit	Cache
`Telecom`					
basic → *timing*	1	6s	2.5s	0.5s	< 1s
timing → *billing*	2	4.5s	3s	0.9s	< 1s
basic → *billing*	1	8s	4.5s	0.5s	< 1s
`Bean`	1	50:17m	3:58m	0.3s	< 1s
`CertRevSim`					
`NullCheck`	1	×	4:40 h	5s	1s
others	1	×	×	n.a.	n.a.
`ProdLine`					
$V_0 \rightarrow V_1$	1	21s	5.5s	< 0.1s	2s
$V_0 \rightarrow V_2$	1	14s	5.3s	< 0.1s	3s
$V_0 \rightarrow V_3$	1	28s	6s	< 0.1s	3s
`Tetris`					
$V_0 \rightarrow V_1$	4	32:35m	28:38m	0.4s	1s
$V_1 \rightarrow V_2$	4	33:09m	29:48m	0.4s	1s
$V_2 \rightarrow V_3$	4	31:18m	31:18m	0.5s	1s
$V_3 \rightarrow V_4$	4	20:46m*	28:41m	0.5s	1s
$V_0 \rightarrow V_4$	4	21:23m*	29:35m	0.5s	1s
`AJHotDraw`					
0.2, *less* → *build*	16	2:29m	(55s)	0.4s	21s
0.2, *less* → *no_inv*	16	1:27m	(25s)	0.5s	13s
0.3, *no_new* → *build*	21	6:10m*	1:31m*	1.1s	30s
`HSQLDB`					
no_aspects → *all*	39	3:06:20h	12:03m	4.1s	2:43m
no_tracing → *all*	39	3:20:32h	14:17m	4.1s	1:23m
no_profiling → *all*	39	2:41:37h	8:18m	4.1s	1:22m
no_pooling → *all*	39	3:37:10h	9:43m	4.1s	1:46m
no_swing → *all*	39	1:51:33h	5:22m	4.1s	1:14m
no_exception → *all*	39	2:58:19h	10:17m	4.1s	2:24m

Table 4.16: Comparison of runtime for different tracers. The column "JUnit" indicates plain JUnit test runtime, "Cache" indicates runtime with pre-calculated call graphs.

and advice in the AspectJ compiler. Thus this agent is unfortunately not directly usable in the context of AspectJ programs (we thus also refrained from giving runtime numbers). However, as our experiments here and also with *JUnit/CIA* showed, for interactive use JVMPI is still to slow (although by magnitudes faster than JDI). Using call graphs generated by this agent will thus considerably reduce runtime, but not solve the problem completely.

For the `Telecom` example, runtime is just few seconds. Runtime for the `ProdLine` example is similar. However, `Bean` and `Tetris` suffer from considerably longer runtime due to the large tracing overhead necessary for the call graph construction (these are no short unit tests!).

For `Tetris` the tests replayed the four games we used for testing, i.e. test time equals play time. Note however that this is an interesting side effect of tracing, as CPU usage never exceeds 50%. This is also confirmed as both tracers nearly took the same time to trace the tests. Note however that this is a tracing side effect as the same plain JUnit test executes in few seconds. Note the two entries for `Tetris` marked with (*). For these two tests the different set of blocks resulted in a different sequence of blocks which resulted in a different test progression. Thus the player 'lost' considerably earlier. In these cases we terminated the game (i.e. the test) manually once the pit was full. For `Bean` the considerable runtime is

actually tracing overhead. The `Bean` test executed some statements in a static loop which is executed 100.000 times.

For the `CertRevSim` example we even failed completely to produce the required call graphs using the JDI or TPTP tracer (with the exception of `NullCheck`/TPTP).

The `HSQLDB` case study gives the most accurate overview of runtime numbers. The JDI tracer is clearly unsatisfactory due to its huge runtime overhead. Producing the call graphs for the 39 tests in both program versions (including the batch tests) took around 3.5 hours. The TPTP agent is clearly preferable in this context, as here the average runtime is around 10 minutes only. Note however that the runtime overhead compared to running plain JUnit tests is still huge (approximately 146.3 on average).

There are still other options we did not explore yet, for example byte code instrumentation (although this might be problematic as the AspectJ compiler also modifies the byte code) or using a modified virtual machine.

An important parameter for scalability of our approach is the number and execution time of test cases to examine, as the time spent for tracing of course directly depends on them. The JUnit test suite examined in the `HSQLDB` case study comprises 39 individual tests, and analysis of these tests already took close to an hour (using the TPTP tracer). If we reused pre-computed call graphs, analysis only took 1:49 minutes on average, which clearly shows that the tracing overhead is the problem.

To summarize, while we acknowledge that the tracer is currently the bottleneck of our prototype, we argue that this performance issue is not a problem of our analysis technique in general (i.e. not any different compared to any other dynamic analysis technique), as several more efficient ways to create traces/dynamic call graphs are known which will improve performance. We thus consider performance improvement in this area—although orthogonal to the aspect impact analysis presented here—as an important topic for future work.

Note that many similar techniques in the context of regression test selection or slicing based on analysis of control flow or program dependence graphs explicitly exclude the construction time of necessary base data structures when considering the costs of their analysis approach. We do not follow this approach of cost measurement as we do not think that call graphs are in general available for each build configuration. This would indeed paint a very nice picture of our approach. For example analysis of the tracing concern in `HSQLDB` took 53:20 minutes (using the TPTP tracing agent and including call graph construction), and 1:49 minutes on average when pre-computed call graphs were used. However, if the reader wants to follow that approach runtime in general was within a few minutes and thus our tool is usable for interactive use, as shown in Table 4.16, column "Cache".

From a theoretical point of view, we only have to traverse the abstract syntax trees of *aspects* in both program versions to derive the induced set of changes and then do a simple traversal of the call graphs in both the original and the edited program version to match nodes and edges in the graph with derived changes. These three algorithms are applied sequentially, and each algorithm is linear in the program size.

4.5.10 Summary

The `Telecom` example gave a first impression of how our tool worked. Especially the coverage of aspect influence is nicely demonstrated by this example.

The programs of the *abc* program suite confirm the observations made for `Telecom`: our tool allows to precisely uncover those joinpoints where aspect influence remains untested by the current test suite. Unfortunately, as the *abc* programs do not come with a suite of unit tests, we had to define our own tests or use available `main` methods, which have the characteristics of system regression test. Theses tests demonstrated the main weakness of our approach: the high overhead due to the construction of the dynamic call graphs for our current prototype.

However we do not consider this to be a principal restriction of this approach as discussed above.

The AJHotDraw case study showed that the dynamic impact tool is very useful to get an impression where aspects influence the system in terms of affected tests. From our experience our tool was very helpful and considerably supported program understanding for the given aspect-oriented system AJHotDraw. Again, it also gave a good impression if the available test suite is sufficient to test aspect influence. The AJHotDraw case study was also the first to demonstrate effects of **LC** changes.

Analyzing HSQLDB we were able to test the scalability of our tool. First, as this study only analyzed homogeneous aspects which affected a large number of joinpoints (with the exception of Swing thread safety), we collected experience how to deal with such large data sets and improved our user interface accordingly by providing a more hierarchical presentation. Second, HSQLDB again demonstrates that the tracing component is clearly the bottle neck for our analysis. If we use pre-computed call graphs, runtime drops dramatically, from an hour (using TPTP) to one and a half minute. Improving the call graph generation is thus an important topic for future work.

To summarize the experience gained from conducting the above case studies, our prototype was able to assist us as follows:

i The tool gave detailed feedback as to *if* and *where* a test suite has to be improved to cover new functionality in the aspect itself (aspect-internal **AM** changes), in introduced methods in the base code (inter type declarations), and also for joinpoints adapted by new aspects. This is especially visible for the HSQLDB case study,

ii The tool gave detailed feedback *where* aspects affect base functionality. While this is similar to the feedback *ajdt* provides, our analysis also offers a semantical analysis of lookup changes and also serves to link changes with tests, so also allowing to give *semantical information* about which functionality is affected.

The case studies however also showed that for a coarse grained suite of system regression tests this information is not helpful ("The system was affected."). For a detailed fine grained test suite however aspects can be precisely associated with the functionality tested by affected tests. This is for example visible in the AJHotDraw case study, the Auction System case study and to a certain degree also for HSQLDB.

iii Finally we found that the information provided by the tool considerably assisted in understanding the currently analyzed program by making aspect influence explicit, although aspect which affect a large set of joinpoints are problematic in this case.

Such aspects however tend to only observe but not modify system behavior. This is for example true for the tracing, profiling, and also exception handling concerns in HSQLDB. Value pooling in this case study actually modifies behavior by reusing existing objects instead of creating new ones, but this change in behavior is due to the restriction to immutable objects (apart from Date) also very local. The swing thread safety aspect modifies system behavior by creating worker objects, and is also considerably more specific. Here, analyzing aspect impact by using the (few) generated changes is again useful.

Given the above, our *Chianti*-based aspect impact analysis promises to considerably help programmers working with AspectJ. If these promises are actually fulfilled however would require a user study which is an interesting topic for future research.

For a discussion of related work on change impact analysis, we refer the reader to the previous sections in this chapter, for related work on program analysis of aspect-oriented construct in general, please refer to Sections 5.5 and 6.5.

Bibliography

[1] ANDY CLEMENT, A. C., AND KERSTEN, M. Aspect-Oriented Programming with AJDT. In *Proceedings of AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003* (July 2003).

[2] APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 432–441.

[3] BATES, S., AND HORWITZ, S. Incremental Program Testing using Program Dependence Graphs. In *Proc. of the ACM SIGPLAN-SIGACT Conf. on Principles of Programming Languages (POPL'93)* (Charleston, SC, 1993), pp. 384–396.

[4] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] BINKLEY, D. Semantics Guided Regression Test Cost Reduction. *IEEE Trans. on Software Engineering 23*, 8 (August 1997).

[6] BOHNER, S. A., AND ARNOLD, R. S. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[7] BREECH, B., DANALIS, A., SHINDO, S., AND POLLOCK, L. Online Impact Analysis via Dynamic Compilation Technology. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2004), IEEE Computer Society.

[8] CHEN, Y., ROSENBLUM, D., AND VO, K. TestTube: A System for Selective Regression Testing. In *Proc. of the 16th Int. Conf. on Software Engineering* (1994), pp. 211–220.

[9] CHESLEY, O., REN, X., AND RYDER, B. G. Crisp: A Debugging Tool for Java Programs. In *Proc. of the Int. Conf. on Software Maintentance* (September 2005).

[10] CHOI, J.-D., AND ZELLER, A. Isolating Failure-Inducing Thread Schedules. In *Proc. ACM SIGSOFT Int. Symp. on Softw. Testing and Analysis (ISSTA 2002)* (Rome, Italy, 2002), pp. 210–220.

[11] CLEVE, H., AND ZELLER, A. Locating Causes of Program Failures. In *Proc. 27th Int. Conf. on Softw. Engineering (ICSE 2005)* (St. Louis, MO, 2005).

[12] DALLMEIER, V., LINDIG, C., AND ZELLER, A. Lightweight Defect Localization for Java. In *Proc. 19th European Conf. on Object-Oriented Programming (ECOOP'05)* (Glasgow, Scotland, 2005).

[13] DEMILLO, R. A., PAN, H., AND SPAFFORD, E. H. Critical Slicing for Software Fault Localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 1996), ACM Press, pp. 121–134.

[14] DINIC, E. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Sov. Math. Dokl. 11* (1970), 1277–1280.

[15] EIBAUER, U. Studying the Effects of Aspect-Oriented Refactoring on Software Quality using HSQLDB as Example. Diplomarbeit (Master Thesis), Universität Passau, FMI, Passau, Germany, Innstraße 32, 94032 Passau, August 2006.

[16] EIDORFF, P. H., HENGLEIN, F., MOSSIN, C., NISS, H., SORENSEN, M. H., AND TOFTE, M. AnnoDomini: From Type Theory to Year 2000 Conversion. In *Proc. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January 1999), pp. 11–14.

[17] ELBAUM, S., KALLAKURI, P., MALISHEVSKY, A. G., ROTHERMEL, G., AND KANDURI, S. Understanding the Effects of Changes on the Cost-Effectiveness of Regression Testing Techniques. *Journal of Software Testing, Verification, and Reliability* (2003). To appear.

[18] ERLIKH, L. Leveraging legacy system dollars for e-business. *IT Professional 2*, 3 (2000), 17–23.

[19] ERNST, M. D. *Dynamically discovering likely Program Invariants*. PhD thesis, University of Washington, 2000.

[20] ERNST, M. D. Invited Talk Static and Dynamic Analysis: Synergy and Duality. In *PASTE '04: Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2004), ACM Press, pp. 35–35.

[21] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (1987), 319–349.

[22] FREEDMAN, D. P., AND WEINBERG, G. M. A Checklist for potential Side Effects of a Maintenance Change. In *Techniques of Program and System Maintenance*, G. Parikh, Ed. 1981, pp. 93–100.

[23] GALLAGHER, K., AND LYLE, J. R. Using Program Slicing in Software Maintenance. *IEEE Trans. on Software Engineering 17* (1991).

[24] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[25] GRAF, J. Eine feingranulare Multi-Versions Programmanalyse für AspectJ basierend auf dem Vergleich der Programmstruktur. Diplomarbeit, Universität Passau, FMI, Passau, Germany, Innstraße 32, 94032 Passau, March 2006.

[26] GUPTA, N., HE, H., ZHANG, X., AND GUPTA, R. Locating Faulty Code Using Failure-Inducing Chops. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)* (Long Beach, California, November 2005), pp. 263–272.

[27] HARROLD, M. J., JONES, J. A., LI, T., LIANG, D., ORSO, A., PENNINGS, M., SINHA, S., SPOON, S. A., AND GUJARATHI, A. Regression Test Selection for Java Software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), ACM Press, pp. 312–326.

[28] HARROLD, M. J., ROTHERMEL, G., WU, R., AND YI, L. An Empirical Investigation of Program Spectra. In *Proc. of the ACM SIGPLAN Workshop on Program Analysis for Softw. Tools and Engineering (PASTE'98)* (Montreal, Canada, 1998), pp. 83–90.

[29] JONES, J. A., AND HARROLD, M. J. Empirical Evaluation of the Tarantula automatic fault-localization Technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)* (Long Beach, California, November 2005), pp. 273–282.

[30] JONES, J. A., HARROLD, M. J., AND STASKO, J. Visualization of Test Information to Assist Fault Localization. In *Proc. Int. Conf. on Softw. Engineering (ICSE'02)* (Orlando, FL, 2002), pp. 467–477.

[31] KRINKE, J. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[32] KUNG, D. C., GAO, J., HSIA, P., WEN, F., TOYOSHIMA, Y., AND CHEN, C. Change Impact Identification in Object Oriented Software Maintenance. In *Proc. of the International Conf. on Software Maintenance* (1994), pp. 202–211.

[33] LADDAD, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[34] LARUS, J. Whole Program Paths. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (May 1999), pp. 1–11.

[35] LAW, J., AND ROTHERMEL, G. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 308–318.

[36] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug Isolation via Remote Program Sampling. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)* (San Diego, CA, 2003), pp. 141–154.

[37] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable Statistical Bug Isolation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)* (Chicago, IL, 2005).

[38] LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. Characteristics of Application Software Maintenance. *Commun. ACM 21*, 6 (1978), 466–471.

[39] LIU, C., YAN, X., FEI, L., HAN, J., AND MIDKIFF, S. P. SOBER: Statistical model-based Bug Localization. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM Press, pp. 286–295.

[40] LYLE, J., AND WEISER, M. Automatic Bug Location by Program Slicing. In *Proceedings of the Second International Conference on Computers and Applications* (Beijing (Peking), China, 1987), pp. 877–883.

[41] ORSO, A., APIWATTANAPONG, T., AND HARROLD, M. J. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ACM Press, pp. 128–137.

[42] ORSO, A., APIWATTANAPONG, T., LAW, J., ROTHERMEL, G., AND HARROLD, M. J. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In *Proc. of International Conf. on Sofware Engineering* (Edinburgh, Scotland, May 2004).

[43] RAMALINGAM, G. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (1994), 1467–1471.

[44] RAMALINGAM, G., FIELD, J., AND TIP, F. Aggregate Structure Identification and its Application to Program Analysis. In *Proc. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January 1999), pp. 119–132.

[45] REN, X., CHESLEY, O., AND RYDER, B. G. Crisp: A debugging tool for Java programs. *IEEE Transactions on Software Engineering* (April 2006). In press.

[46] REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 432–448.

[47] RENIERIS, M., AND REISS, S. Fault Localization with Nearest Neighbor Queries. In *In Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (Montreal, Quebec, Canada, October 2003), pp. 30–39.

[48] REPS, T., BALL, T., DAS, M., AND LARUS, J. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *Proc. of the 6th European Softw. Conf. (ESEC/FSE'97)* (1997), pp. 432–449. Springer-Verlag LNCS Vol. 1013.

[49] ROTHERMEL, G., AND HARROLD, M. J. A safe, efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol. 6*, 2 (1997), 173–210.

[50] RUTHRUFF, J. R., BURNETT, M., AND ROTHERMEL, G. An Empirical Study of Fault Localization for End-User Programmers. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 352–361.

[51] RYDER, B. G., AND TIP, F. Change Impact Analysis for Object-Oriented Programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2001), ACM Press, pp. 46–53.

[52] SAFF, D., AND ERNST, M. D. Reducing wasted Development Time via Continuous Testing. In *Fourteenth Int. Symp. on Softw. Reliability Engineering* (Denver, CO, November 17–20, 2003), pp. 281–292.

[53] SAFF, D., AND ERNST, M. D. Automatic Mock Object Creation for Test Factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Softw. Tools and Engineering (PASTE'04)* (Washington, DC, USA, June 7–8, 2004), pp. 49–51.

[54] SAFF, D., AND ERNST, M. D. Continuous Testing in Eclipse. In *Proc. of the 26th Int. Conf. on Softw. Engineering (ICSE'05)* (St. Louis, MO, USA, May 2005).

[55] STEVENS, W. P., MYERS, G. J., AND CONSTANTINE, L. L. Structured Design. *IBM Syst. J. 38*, 2-3 (1999), 231–256.

[56] STOERZER, M., RYDER, B. G., REN, X., AND TIP, F. Change Classification and its Applications to Program Development and Testing. Tech. Rep. DCS-TR-05-566, Department of Computer Science, Rutgers University, April 2005.

[57] STOERZER, M., RYDER, B. G., REN, X., AND TIP, F. Finding Failure-Inducing Changes using Change Classification. Tech. Rep. DCS-TR-582, Department of Computer Science, Rutgers University, September 2005.

[58] TEITELBAUM, T. Codesurfer. *SIGSOFT Softw. Eng. Notes 25*, 1 (2000), 99.

[59] TIP, F. A Survey of Program Slicing Techniques. *J. of Programming Languages 3*, 3 (1995), 121–189.

[60] TONELLA, P. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Trans. on Software Engineering 29*, 6 (2003), 495–509.

[61] VAN RIJSBERGEN, C. *Information Retrieval*. Butterworths, London, 1979.

[62] WEISER, M. *Program Slices: formal, psychological, and practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[63] ZECHMANN, H. Dynamische Impact Analyse für aspekt-orientierte Programme mit Hilfe des Call Graphen. Master's thesis, Universität Passau, October 2004.

[64] ZELLER, A. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99)* (Toulouse, France, 1999), pp. 253–267.

[65] ZELLER, A. Making Students Read and Review Code. In *ITiCSE '00: Proc. of the 5th annual SIGCSE/SIGCUE ITiCSE Conf. on Innovation and technology in computer science education* (2000), ACM Press, pp. 89–92.

[66] ZELLER, A. Isolating Cause-Effect Chains from Computer Programs. In *Proc. ACM SIGSOFT 10th Int. Symp. on the Foundations of Softw. Engineering (FSE 2002)* (Charleston, SC, 2002), pp. 1–10.

[67] ZELLER, A., AND HILDEBRANDT, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. on Softw. Eng. 28*, 2 (2002), 183–200.

5 System Evolution

One of the main goals of aspect-orientation is to improve the modularity of a system by allowing to encapsulate crosscutting concerns thus facilitating system maintenance. On the one hand this goal can be reached, as the crosscutting concern is indeed localized in the aspect, and additionally the tangled code is removed from the base system, thus improving internal cohesion of base modules. On the other hand, aspect and base are *tightly coupled* and furthermore this coupling is only *implicit* as it depends on the results of the pointcut expressions.

We briefly outlined problems of current aspect-oriented languages when reviewing available case studies in Section 2. This chapter specifically addresses problem "P3: System Evolution". In this chapter the reasons and implications of tight aspect-base coupling are examined and a tool approach to leverage the resulting problems is discussed. The approach presented in this chapter is an extension of own work presented in [19, 20].

5.1 Pointcuts and Coupling

An important quality criterion for software systems is their modularity. Modular systems tend to be easier to understand and maintain as they localize a single concern in a single module and optimally allow to independently change different modules. This is especially important as estimated 80% [11, 2] of the total cost during the lifetime of a software system stem from the maintenance phase. Rules of thumb to achieve a good modularity are to provide modules which have a *high internal cohesion* and are *loosely coupled* among each other.

A crosscutting concern per definition is not properly modularizable in a single place by using traditional programming techniques. Implementing such a concern using an aspect enhances the modularity of the system, as both the crosscutting concern is localized in the aspect and the cohesion of those modules where the tangled code is removed from is improved as well. However, this improvement of modularity is not without cost. A major problem of current aspect languages is that aspect and base potentially are tightly coupled. Even worse, this coupling is only implicit, as aspect influence is not directly visible in the affected base modules.

These problems in part have been discussed in literature, for example refer to [8]. In this thesis, the problem of code evolvability has been formulated in Section 2.2.2. We will further analyze the crosscutting mechanism in AspectJ-like languages in the following to give a detailed overview of the problem.

5.1.1 Pointcuts and Aspect-Base Coupling

The ability to reason about a program to select joinpoints has been identified as a core property of aspect-orientation and termed *quantification* in [3]. Recall that aspects in general provide two constructs to specify new behavior and where it should apply, called *advice* and *pointcuts* in AspectJ. Pointcuts are expressions allowing the programmer to specify *where* advice should be executed. So pointcuts implement this quantification property, as discussed in Section 1.2.2. The *aspect weaver* evaluates pointcut expressions and uses this information to combine advice with the base system to finally produce the executable system. If we follow [3] and accept quantification as a core property of aspect-orientation, then any aspect-oriented language has to provide a comparable mechanism.

We use the term *joinpoint selection language* to refer to the language used to formulate such quantified expressions. The availability of such a joinpoint selection language is actually observable when studying other aspect-oriented approaches beside AspectJ. As this chapter proposes an analysis-based approach to deal with this problem for AspectJ, we will only use the term joinpoint selection language if we explicitly refer to the general concept and in general use the term pointcut throughout this chapter.

Although there is a broad spectrum of different joinpoint selection languages, a pointcut expression can be seen abstractly as a *function* taking the (base) program as input and calculating a set of matched joinpoints as output. The main differences are *which joinpoints* can be selected and—more important for system evolution—*how* this function can be specified. The latter describes the expressiveness of a joinpoint selection language.

Optimally a pointcut expression by itself should transport its semantics, i.e. one would like to write a pointcut like *"update the observer whenever an observed subject changes"*. If today's systems would allow programmers to write such pointcuts, then indeed aspects relying on these pointcuts would be *semantically stable*, and also *easy to formulate*. For an observer aspect for example this means that the observer is notified of all relevant subject changes.

Unfortunately many—if not all—of today's main stream joinpoint selection languages only allow to select joinpoints *based on lexical, syntactical or structural properties* of the code. Pointcuts explicitly name elements in the code, e.g. methods, fields or types, to address joinpoints. I.e. the programmer has to specify that the observer has to be updated after a call to method `setPos(..)` in one of the observed objects.

As it is rather inconvenient to specify a larger set of joinpoints by explicitly "naming" each joinpoint, AspectJ introduced *wild cards* allowing to exploit naming conventions. So it is possible to formulate pointcuts like "update the observer whenever a `set*(..)` method is called on one of the observed subjects". However, using wild cards results in a new problem. Pointcuts using this mechanism exploit *naming conventions*. As such conventions are hard to check by a compiler, they are *never guaranteed*.

Naming and wild cards are natural things for programmers, thus these pointcut languages are relatively intuitive and convenient to use (for programmers!). However, there is also an important drawback: specifying joinpoints by using wild cards and exploiting naming conventions can easily result in unwanted or accidentally lost matches. So developing a *correct* pointcut is not trivial. For small programs a pointcut mismatch due to wild cards can easily be detected. However, aspects have been proposed for large or distributed system scenarios, where it is much harder to find spurious or missed matches. In general, the aspect programmer needs *global system knowledge* to assure that his pointcut works as expected.

The problem is that joinpoint selection based on lexical as well as syntactical or structural properties of a program do not allow to write pointcuts like "update the observer whenever an observed subject changes"—the programmer rather has to (lexically) enumerate all relevant joinpoints. Wild cards are just a (dangerous?) means to reduce the effort of writing the pointcut down. They do not provide additional robustness for pointcut definitions as even strict coherence to naming conventions does not guarantee that no additional joinpoints are

selected.

This enumeration of joinpoints in direct consequence results in aspects *tightly coupled with a specific version of the base system*. This is an important observation: while aspect-orientation allows to *improve internal cohesion of modules* it introduces additional modules which are potentially tightly coupled with the rest of the system.[1]

To reduce this tight coupling, *aspect inheritance* is a way to split advice and pointcut definitions. Advice is defined in the abstract aspect, and pointcuts are specified later in a concrete sub-aspect. However, while this approach allows to define reusable functionality in an aspect, the concrete aspect is still tightly coupled with the base system. Thus aspect-oriented systems *only provide half the solution* they promised to offer—they allow to localize crosscutting concerns in separate modules with a high internal cohesion. However for aspect-oriented systems there is always the problem of a high aspect-base coupling.

5.1.2 The Understanding of Pointcuts

In Section 1.2.2, we described pointcuts as a means to select joinpoints which should be adapted by advice. However it is worth to shed more light on the nature of joinpoints and also to examine the understanding of pointcuts by programmers, to understand the problems that pointcuts induce.

Pointcut Ideas, Expected and Actual Results

It is important to understand that there are three levels of semantics of a pointcut.

Pointcut Ideas: This is the idea a pointcut should capture, i.e. "update the observer, whenever an observed subject changes".

Expected Pointcut Results: This is the set of all joinpoints the programmer *expects* to be necessary to implement the original idea, e.g. "update the observer, after a `set*`-method has been called".

Actual Pointcut Results: This is the set of joinpoints actually selected by the defined pointcut, e.g. all setter methods, the method `settleDown()`, but not the method `changePosition(...)`.

For correct systems, the sets of joinpoints resulting from all three "pointcut definitions" have to be identical. However, current joinpoint selection languages face the problem that—as already outlined in the definition—differences among these sets can easily occur and jeopardize system semantics.

Programmers have clear expectations of the joinpoints selected by a certain pointcut definition, i.e. the expected pointcut results in general should be the set of joinpoints matching the original pointcut idea. However, here two problems show up.

Expected-Actual Mismatch: Due to misconceptions of the programmer, the formulated pointcut expression does not capture all necessary, and/or too many joinpoints, e.g. because naming conventions exploited by the pointcut are violated in the system.

Pointcut Idea-Expected Mismatch: Even if the actual result matches the intended pointcut results for a given program version, i.e. even if the expected pointcut results correctly formalize the intended pointcut, this matching is *not stable* when system evolution is considered. We will further discuss this problem in the following.

[1] Note that this observation is based on the deficiencies of current pointcut languages forcing programmers to enumerate relevant joinpoints.

While the expected-actual mismatch can be removed by disciplined program development (this is a conventional bug), the *pointcut idea-expected mismatch is problematic in general*, as here an informally stated idea has to be captured in a way that is valid for *any version of the underlying system*. Defining a joinpoint selection language which is capable to actually close this gap is a *very ambitious task*[2]; it can be doubted whether this is solvable in general.

Pointcuts are Part of the Program

While any programmer first develops a pointcut idea and then designs a pointcut expression he expects to implement this idea, programmers in general are not consciously aware of the above considerations.

A pointcut expression is not considered to be a *function* reasoning on the program as input, but rather as a *part of the program* itself. What programmers see however is that the semantics of one part of the program (the pointcut and consequently the aspect) depends on a lexically (and maybe also semantically) independent part of the system. For programs such behavior is *unintuitive*, especially for programmers used to well modularized systems. This is exactly what loosely coupled modules should avoid.

Narrowing the Gap—Structure-Based Pointcuts and Current Research

To actually write down a pointcut idea directly is a *very nice vision*—but not achievable in general. What researchers try to formulate today is to *narrow* the gap between original pointcut ideas and pointcut expressions one can actually write down in a program. A first step which can also be observed in the development of AspectJ was the propagation of so called *structure-based pointcuts*. For these pointcuts the main idea is to use types or method signatures instead of names or lexical positions to specify pointcuts, as types tend to transport some semantics.

While this approach might increase robustness, this is still no general solution as also types and signatures can change during system evolution. As it is also less intuitive (and often simply not feasible) to write pure structure based pointcuts, i.e. to describe joinpoints without naming elements in the code, lexical joinpoint selection has not been abandoned. As a result, many of today's languages—including AspectJ—have a mixture of lexical and structural selection constructs thus suffering from the described problems.

5.1.3 Evolution in AO Systems

While the tight coupling of aspects up to now has been discussed from a theoretical point of view, analyzing its concrete effects for evolution of aspect-oriented systems is necessary. In the above we compared pointcut expressions with a function using a given (base) program as input and producing the set of selected joinpoints as output. Although for functions in general one expects changed results for changed inputs, for pointcuts there are important differences. Even if one might expect changes in the set of selected joinpoints in this context, one does not expect changes in the *aspect semantics*. Or, in the context of the above, the actual pointcut results should optimally change to match the original pointcut idea. Unfortunately, the actual pointcut results are not cooperative in general, but instead show some in-between mixture of expected and unexpected additional and lost matches. We will justify this claim in the following.

Let us once again interpret a pointcut as a function. The set of matched joinpoints depends on *the whole program* as an input to this function, i.e. an aspect is potentially coupled with *all* modules of the system. Although this seems exaggerated, this actually is the case, if the

[2]This is close to the ultimate goal for each language developer: to reduce a language to two statements: "Do-What-I-Want" and "Remove Bugs". Note the second statement.

pointcuts are not restricted to specific parts of the system. In general a pointcut is an all-quantified expression over a program—and thus indeed its semantics depends on the program as a whole. Even worse, some pointcut languages, including AspectJ, allow to specify pointcut expressions which only result in the selection of a joinpoint if another joinpoint has been previously selected (cmp. `cflow`). Consequently, to evaluate the pointcut expression, one has to actually calculate a fixed point. This is clearly opposed to the statement that the result of a pointcut expression in general is "obvious" for the programmer.

For these complex functions, a change in any code artifact of the system can result in different results, manifested as additional or lost matched joinpoints, thus influencing semantics of the system as a whole. The same effects are possible by mere addition of new program elements; so there is even no closed world pointcuts reason on. For illustration consider the following scenario:

Example 5.1.1 (System Evolution Scenario) *A programmer has correctly specified a pointcut (i.e. pointcut idea, expected and actual pointcut results are the same). The corresponding aspect works as intended, all tests are successful. Afterward the base code evolves, e.g. by trivial changes like renaming some methods, changing some method signatures and adding new methods.*

If the programmer misses to update calls to changed methods, the compiler will issue a compile error. However, if we consider a pointcut referencing a method by its former name or by its former signature the set of joinpoints picked out by the—unchanged!—pointcut definition is silently altered*. As a consequence, we now have a mismatch in pointcut ideas and actual pointcut result. The pointcut expression is no longer a valid implementation of the original pointcut idea.*

In general there are several (trivial) non-local base code changes possibly modifying pointcut semantics in terms of actually selected joinpoints, for example:

Rename: Renaming classes, methods or fields influences semantics of `call`, `execution`, `get/set` and other pointcuts. Wild cards can only provide limited protection against these effects.

Move method/field/class: Pointcuts can pick out joinpoints by their lexical position, using `within` or `withincode`. Moving classes to another packages or methods to another class obviously changes matching semantics for such pointcuts.

Add/delete method/field/class: Pointcut semantics are also affected by adding or removing program elements. New elements can (and sometimes should) be matched by existing pointcuts, but in general pointcuts cannot anticipate all possible future additions. Removal of program elements naturally results in 'lost' joinpoints.

Signature Changes: `call`- and `execution`-pointcut designators allow to pick joinpoints based on method signatures including method visibility. Thus signature-based pointcut definitions—although propagated as the more robust mechanism—are nonetheless fragile.[3]

If a code artifact is changed, other artifacts depending on it in general have to be adapted.[4] (Automated) refactoring [4], as available in IBM's Eclipse IDE for Java or the Smalltalk Refactoring Browser [16] might be a way to avoid breaking pointcuts in some cases, but for AspectJ refactoring support is currently not available and—more important—are problematic

[3] This seems less relevant as public interfaces should be stable. However, aspects are not restricted to public interfaces.

[4] E.g. renaming a method requires to modify all calls accordingly.

in general as dynamic pointcut designators (like the `if`- or `cflow`-constructs) cannot be evaluated statically. Thus in general it is not feasible to determine if a specific dynamic match has been added or lost and thus if a pointcut has to adapted by a refactoring or not.

Additionally, automated refactoring requires that the user *explicitly requests* a refactoring, thus refactoring does not address system evolution in general. As an example just consider adding new methods, classes or packages due to new functionality. In these cases, a refactoring tool cannot help, as no existing code artifacts are modified and thus per definition no refactoring of an existing code artifact has been requested. But as we have seen above, addition of new code artifacts might result in additional adapted joinpoints and thus in changed aspect/program semantics. Finally, the new set of relevant joinpoints in terms of the pointcut idea in general is not formally specified, i.e. for the refactoring the set of joinpoints to select in general is not computable.

We have seen so far that aspects are tightly coupled to the base system due to deficiencies of current pointcut languages and that—as a result of this coupling—semantics of pointcut expressions can unexpectedly change due to system evolution. While similar problems are also true for traditional programming paradigms like object-oriented programming[5], for aspects there is an additional problem: Aspects influencing a given base class are *not directly visible* in the adapted class. As a result, a programmer modifying e.g. a class of the base system is not necessarily aware of all the aspects possibly selecting joinpoints in this class. Tool support lightens this problem [1], but in our opinion this does not solve the problems for *evolution of aspects, classes and their dependencies*, as here a tool has to keep track of *differences* between subsequent versions.

We briefly outlined the problem of aspect-oriented software evolution in Chapter 2, and identified this problem as one of the major problems of aspect-orientation today (P3: System Evolution). Good support for System Evolution is crucial for the long-term usability of any languages, as a major share of total costs of software development arise during software maintenance. For AspectJ and similar languages, on the languages level this problem is in general unsolved (and may not be solvable at all).

We refer to the problem of changed program semantics due to a mismatch in pointcut ideas and actual pointcut results[6] by using the term *fragile pointcut problem*. In this chapter we show how tool support can help to lighten this particular problem, be recapitulating and extending our own previously published results [19, 20].

5.2 Pointcut Delta Analysis

As in this thesis tool support is advocated to support aspect-oriented programming, in the following a delta analysis technique addressing the fragile pointcut problem is proposed. The approach presented here addresses the fragile pointcut problem for *current languages* and systems written in these languages. It is intended to ease maintenance and avoid rising costs for systems written in languages similar to AspectJ which suffer from fragile pointcut constructs. Although future pointcut languages might lighten the problem by narrowing the gap between intended pointcuts and pointcut constructs in aspect languages, completely closing the gap might not be possible. Thus the proposed analysis will still be helpful allowing to double-check programmer expectations and actual pointcut semantics.

The basic idea for the approach presented here is that programmers should be alerted of changes in the matching behavior of advice (i.e. the expected and actual pointcut results) if the underlying system changes. Unfortunately this approach is necessarily incomplete as there is still no information about the set of joinpoints to-be-matched to maintain aspect semantics. But focusing on actual changes can also help to check for expected changes in a pointcut set.

[5] Adding methods overriding a superclass implementation results in lookup changes.

[6] We will address this mismatch with the term *joinpoint mismatch* in the following.

While this is no perfect solution, it considerably improves the situation compared to manual inspection of matching behavior in the new program version.

5.2.1 Detecting Failures and their Reasons

Assume that due to joinpoint mismatches a new program version suffers from a bug. Although unintended semantical differences introduced into a system are (hopefully) revealed by rerunning a regression test suite (failing test), in general the programmer needs more information. Test failures do not explain failure reasons. Thus for a failing test, test results (e.g. an exception) have to be further analyzed to actually track down the bug.

Finding failure inducing code modifications is hard if changed pointcut semantics due to non-local base edits are responsible, especially as a trace—if available at all—not necessary points to the failure inducing edit. If a joinpoint mismatch is failure inducing, actually finding this mismatch is very hard when examining a single program version. A programmer has to be aware of affecting aspects *and their semantics* to reason about system correctness. In large systems doing this manually is infeasible.

For the remaining of this chapter we will use the program shown in Figure 5.1 as a running example to demonstrate our delta analysis. The figure shows both versions of example code. The underlined code is *added*, the code canceled out removed in the edited program version. Note that the two pieces of advice are marked up with a special kind of comment, which gives us a unique identifier for advice. We will discuss it's necessity in Section 5.2.3. We will compare those two versions using our *pointcut delta analysis* in the following.

The difference between these two versions includes moving a method (`update` from class `D` to class `C`), modification of pointcut `setField` and addition of a new piece of advice. Although this program is tiny, the resulting changes in advice matching behavior are already hard to see without support.

5.2.2 Calculating PC-Deltas

Once a problem is known, it is often half solved. This is especially valid for pointcuts unexpectedly changing their semantics due to e.g. base code edits. Consider the following scenario: we have two versions of a program: an original version $\mathscr{P}$ and an edited program version $\mathscr{P}'$. To detect semantical differences in program behavior due to changed pointcut semantics, we propose an analysis which detects changes in matching behavior (called *pointcut delta*) and also traces these differences back to their corresponding code modification(s).

To derive the pointcut delta the following analysis is used: Informally, we calculate the set of matched joinpoints for both versions of the program and compare the resulting sets, producing delta information for pointcut matching. This approach is possible for any AspectJ-like language where the set of matched pointcuts is statically computable.

For cases where joinpoint matching cannot be decided statically, the matching is conservatively approximated and the resulting match marked accordingly. As the weaver also needs this information to generate the woven program, static matching information in general is available for most if not all compiled languages. Dynamic pointcut expressions result in runtime checks associated with runtime penalties, which are avoided if joinpoint matching information is statically known. For purely dynamic approaches however this might not be true. This is clearly one of the limitations of the approach presented here. So, for a given aspect-oriented program $\mathscr{P}$, function

$$match : \mathscr{P} \rightarrow JP \times ADV \times Q$$

determines the set of all aspect-joinpoint relations, where

- JP is the set of all joinpoints in $\mathscr{P}$,

Listing 5.1: Original and edited program version. Code added in the edited program version is underlined, code canceled out is deleted in the edited program version.

```
1  aspect A {
2    pointcut dynamic(): within(C) && if(isTrue());
3    pointcut setField(): set(int *) && dynamic();
4
5    before(): setField() {
6      System.out.print("Changing field value");
7    } /*[161-0-1108388203184-8132904]*/
8
9    after (): call(* update()) && if(isTrue()) {
10     System.out.print("Field update done");
11   } /*[276-1-1108388538145-4535112]*/
12
13   private boolean isTrue() {
14     ... // some dynamic predicate
15   }
16 }
17
18 class C {
19   int x;
20   static void main(String[] args){
21     D d = new D();
22     d.setX(4);
23     d.setX(5);
24     d.update();
25   }
26   void setX(int x){
27     this.x = x;
28   }
29   void update(){
30     x = 2 * x;
31   }
32 }
33
34 class D extends C {
35   void setX(int x){
36     this.x = x;
37   }
38   void update(){
39     x = 2 * x;
40   } /* deleted in edited version */
41 }
```

- *ADV* is the set of all advice in $\mathscr{P}$ and
- Q is the quality of the matching relation, either *dynamic* or *static*.

5.2.3 Defining an Equality Relation for Pointcuts and Advice

To calculate the delta from $match(\mathscr{P})$ and $match(\mathscr{P}')$ it is necessary to identify *corresponding joinpoints and advice* in both versions of the program, $\mathscr{P}$ and $\mathscr{P}'$, respectively. More formally speaking we need *equality relations* defined for joinpoints (or better for joinpoint representations) and advice of both program versions.

However, while this is trivial for methods, both joinpoints and advice are unnamed constructs (at least for AspectJ) and thus matching is problematic. What is needed is an *identifying representation* for joinpoints and advice which is stable across different versions, comparable to a method signature.

The lexical position of a joinpoint/advice in the source code ("source handle") can be used to identify a joinpoint in a given system version. Unfortunately this is no longer true if subsequent versions are considered, as functionally irrelevant modifications like adding some blank lines or comments changes the joinpoint/advice source position and thus makes identification of corresponding items in both program versions impossible. Reordering of code artifacts in a file raises similar problems.

For advice this problem can be solved by (automatically) naming a new piece of advice once it is introduced in the system, as advice is a first class item in a program. This can be done by simply attaching an identifying comment to each advice body when a piece of advice is first encountered (as also visible in Figure 5.1). While naming as a standard solution reliably solves this issue for advice, joinpoints are more complicated as they are no first class code artifacts. A joinpoint is only implicitly defined by the program statements.

However, similar to method signatures, it is possible to identify joinpoints using *joinpoint signatures* composed of relevant code artifact signatures at the joinpoint. For example a `call`-joinpoint can be identified by the signature of caller method, called method and a counter; similarly a field set/get is identified by the accessed field and the, for example, method-signature the access is located in.

Joinpoint signatures consist of three parts: (i) the joinpoint type, (ii) a referenced program element and (iii) a containing program element.

Definition 5.2.1 (Joinpoint Signatures, Notation) *For a joinpoint jp*

- *env*(jp) *indicates the program item containing this joinpoint,*
- *ref*(jp) *indicates the program item referenced by this joinpoint,*
- *and kind*(jp) *indicates the joinpoint kind.*

For a detailed descriptions how joinpoint signatures are derived for the AspectJ joinpoint model refer to Table 5.1.

Example 5.2.1 (Joinpoint Signatures) *Consider the joinpoint jp represented by the call to* `D.setX(int)` *in* `C.main(String[])` *(line 23) in the original program version of Listing 5.1. The signature of this joinpoint is* $(\texttt{call}, \texttt{D.setX}(\texttt{int}), \texttt{C.main}(\texttt{String}[]))$. *Respectively, for this joinpoint,* $env(jp) = \texttt{C.main}(\texttt{String}[])$, $ref(jp) = \texttt{D.setX}(\texttt{int})$, *and* $kind(jp) = \texttt{call}$.

Note that this joinpoint identification scheme is a heuristic, as in general a single method can contain multiple calls to the same callee all forming different joinpoints. In this case joinpoint signatures are not able to distinguish these joinpoints (apart from a counter).

Joinpoint	Containing Element	Referenced Element
`call`	method/ctor signature, advice id, class/aspect name	method/ctor signature
`execution`	method/ctor signature	method/ctor signature
`get/set`	method/ctor signature, advice id, class/aspect name	qualified field name
`preinit.`	class/aspect name	qualified field name
`init.`	class/aspect name	qualified field name
`staticinit.`	class/aspect name	qualified field name
`handler`	method/ctor signature, advice id, class/aspect name	qualified exception type
`adviceexec.`	method/ctor signature, advice id, class/aspect name	advice id

Table 5.1: Overview of joinpoint signature generation schema.

Example 5.2.2 (Failure of Joinpoint Signatures) *Consider the edited program version shown in Listing 5.1. In method* `C.main()` *a second call to* `D.setX()` *(line 22) has been inserted* before *the call to* `setX()`*, also present in the original program versions. This additional call forms a new joinpoint which is assigned the* same signature *as the original call. As is has been inserted before the original call, the joinpoint signature will match the signature of the call in the original program.*

To deal with multiple joinpoints with the same signature, we augmented the joinpoint signatures with a trailing counter. This counter allows to keep track of the multiplicity of same-signature joinpoints. This allows us to detect changes in the number of same-signature joinpoints, although it is not possible to directly match respective joinpoints in two versions, as is also the case in the above example.

Although only a heuristic, this model of joinpoint identification using joinpoint signatures is descriptive enough for our purposes,

- as we keep track of the *number* of matched joinpoints by adding a counter to the respective joinpoint signature, thus enforcing unique identifiers and
- as the *pointcut language* of AspectJ itself is not able to directly distinguish such joinpoints either.

Thus in practice this identification schema works very well, even if this model cannot guarantee that indeed equal joinpoints are identified by the same signature in both versions. The relevant information for the user—concerning the above example—is that in method `C.main()` an additional joinpoint is matched, potentially affecting system semantics.

With these two notions of equality for advice and joinpoints across different program versions it is now straightforward to calculate the delta set for $match(\mathscr{P})$ and $match(\mathscr{P}')$ by using standard set operators, assuming that all matching information is statically known.

5.2.4 Dynamic Pointcut Designators

Up to now we did not explicitly consider *dynamic pointcut designators*. For these designators, the set of selected joinpoints can not be completely evaluated at compile time. Examples are the `if` or `cflow` pointcut designators. Statically one has to conservatively approximate these constructs by assuming `true` for each such predicate as evaluation of pointcut expressions requires runtime values.

For the delta analysis this results in the comparison of *supersets* rendering the derived information less reliable. To deal with this problem we refine the delta analysis to exploit the associated matching quality information (static/dynamic) and mark up resulting delta entries correspondingly. By adding this knowledge six different cases can be distinguished:

New matches: A new statically determined advice association appeared in $\mathscr{P}'$:

$$new_{static} = \{(jp, adv, +_{static}) \mid \exists(jp, adv, \texttt{static}) \in match(\mathscr{P}') \wedge \nexists(jp, adv, \bullet^{7}) \in match(\mathscr{P})\}$$

New potential matches: A new advice association has to be conservatively assumed in $\mathscr{P}'$, although evaluation is not possible at compile time:

$$new_{dynamic} = \{(jp, adv, +_{dynamic}) \mid \exists(jp, adv, \texttt{dynamic}) \in match(\mathscr{P}') \wedge \nexists(jp, adv, \bullet) \in match(\mathscr{P})\}$$

Lost matches: A statically determined advice association is no longer present in $\mathscr{P}'$:

$$lost_{static} = \{(jp, adv, -_{static}) \mid \exists(jp, adv, \texttt{static}) \in match(\mathscr{P}) \wedge \nexists(jp, adv, \bullet) \in match(\mathscr{P}')\}$$

Lost Potential matches: A conservatively assumed advice association is no longer present in $\mathscr{P}'$:

$$lost_{dynamic} = \{(jp, adv, -_{dynamic}) \mid \exists(jp, adv, \texttt{dynamic}) \in match(\mathscr{P}) \wedge \nexists(jp, adv, \bullet) \in match(\mathscr{P}')\}$$

Dynamic → Static: The set of associated advice did not change, but in contrast to $\mathscr{P}$ the responsible pointcut expression can be statically evaluated in $\mathscr{P}'$:

$$change_{d \to s} = \{(jp, adv, d \to s) \mid \exists(jp, adv, \texttt{dynamic}) \in match(\mathscr{P}) \wedge \exists(jp, adv, \texttt{static}) \in match(\mathscr{P}')\}$$

Static → Dynamic: The set of associated advice did not change, but in contrast to $\mathscr{P}$ pointcut evaluation needs conservative approximations in $\mathscr{P}'$:

$$change_{s \to d} = \{(jp, adv, s \to d) \mid \exists(jp, adv, \texttt{static}) \in match(\mathscr{P}) \wedge \exists(jp, adv, \texttt{dynamic}) \in match(\mathscr{P}')\}$$

Finally the pointcut delta is defined as the union of the classified delta sets, thereby also capturing dynamic pointcut designators:

$$pcDelta(\mathscr{P}, \mathscr{P}') = new_{static} \cup new_{dynamic} \cup change_{d \to s} \cup lost_{static} \cup lost_{dynamic} \cup change_{s \to d}$$

Note that in the above definitions we map triples associating joinpoints and advice (with a given certainty; either static or dynamic) to *delta triples*, which have a similar structure, but different semantics.

Note that the above assumes that *jp* and *adv* alone identify a tuple $(jp, adv, \bullet)$. This of course depends on the chosen joinpoint representation. As joinpoint signatures as proposed here include a counter this requirement is fulfilled in our case. Using these six categories, the derived matching delta is enriched with *confidence information*. Static information can be

[7] In the following, '•' will indicate any possible value for a tuple variable (wild card).

trusted, dynamic information still requires programmer investigation, but offers hints where to start.

Clearly a goal must be to reduce uncertain information as much as possible. Program analysis can be used to evaluate some dynamic expressions at compile time (i.e. by using partial evaluation, abstract interpretation or related techniques) so reducing the amount of spurious matches, but an exact calculation of matching information in general is not computable. As this is also a relevant problem for performance of AOP software, this is a current research topic [13, 18]. However, this is not in the scope of the work presented here.

5.3 Explaining Deltas

The benefit of calculating the delta set is that these sets tends to be *small* compared to the system's overall number of matched joinpoints, at least if we do not consider adding or removing aspects at the moment. If $pcDelta(\mathscr{P}, \mathscr{P}') = \emptyset$, the programmer can assume that an edit did not affect semantics of any static pointcut expression in terms of matched joinpoints (i.e. here the actual joinpoint result did not change). Note however that joinpoint mismatch due to dynamic pointcut expressions can occur and is not covered ($(jp, adv, dynamic) \in match(\mathscr{P}) \cap match(\mathscr{P}')$). If $pcDelta(\mathscr{P}, \mathscr{P}')$ contains differences, these differences can easily be traced back to the affected aspects, so the aspect programmer can be notified of this change. As a result, the delta is valuable information as unexpected matches can be found more easily.

The inverse problem is to find *expected but not experienced matches*. This corresponds to a change in the pointcut definitions (and thus the expected pointcut result). Unfortunately this is considerable harder to do automatically as here an analysis would need information about the expected pointcut results. These expected results would have to be checked against the actual matching behavior. However, although this can't be done automatically, for the programmer it is easier to check a small delta than the whole program in order to find out is expected matches are actually present. Thus our analysis also offers support in this case.

While the delta set alone is valuable, we refined our analysis to identify *causes for these deltas*, to allow a programmer to immediately see *why* a specific delta entry exists. Potential changes resulting in pointcut deltas are threefold:

1. Aspect evolution can add additional or remove some pieces of advice. This also includes addition or removal of a complete aspect (new expected pointcut result).
2. If a *pointcut itself* has been *modified*, we expect differences in its matching behavior[8] (new expected pointcut result).
3. *Base Code Edits*, or more precisely their effects on joinpoints are most problematic and most likely the reason for *unexpected* changes in the matching behavior, as outlined before (new actual pointcut result).

To explain why a joinpoint match is in the delta, we enriched each delta entry $(jp, adv, \bullet)$ with additional information explaining the reasons *why* this entry exists by associating relevant pointcuts, advice and joinpoints with atomic changes derived from comparing the two program versions $\mathscr{P}$ and $\mathscr{P}'$. In the following we assume that the set of atomic changes $\mathscr{A}$ has been calculated as described in Chapter 4.5.

5.3.1 New or Removed Advice

Most obvious, additional or lost matches can result from added or removed advice. Note that this also includes adding or removing a whole aspect. For each delta entry $(jp, adv, \bullet)$ we

[8] This also includes modification of anonymous pointcuts.

check if atomic changes exist which reference the advice *adv*.

Definition 5.3.1 (Advice Changes) *For* $(jp, adv, \bullet) \in pcDelta(\mathscr{P}, \mathscr{P}')$*, we calculate atomic changes associated with the advice adv as*

$$(adv, \Delta_{adv}) = \texttt{getAdvChanges}(jp, adv, \mathscr{A}(\mathscr{P}, \mathscr{P}')))$$

where

```
1   getAdvChanges((jp, adv, •), A(P, P')) =
2     case • = +static ∨ +dynamic then
3       (adv, {AAD(adv)} ∩ A(P, P'))
4     case • = −static ∨ −dynamic then
5       (adv, {DAD(adv)} ∩ A(P, P'))
6     otherwise
7       (adv, ∅)
```

Associating atomic changes with advice is straightforward. A tuple can be in the delta, if advice has either been added to (**AAD**(adv)) or removed from (**DAD**(adv)) the system. However, this is also the most simple case. More interesting are changes in pointcut definitions, which we will examine next.

5.3.2 Modified Pointcuts

Finding modified pointcut definitions only requires a simple analysis of textual differences in the respective code. The information which joinpoints and advice are linked by this pointcut expression is in general also derivable from the system, as this information is also needed by the weaver. The delta tuples associate joinpoints with adapting advice.

For AspectJ analyzing the source code of advice directly allows to collect all (transitively) referenced pointcut expressions.[9] We express these dependencies using two relations

$$reference(\mathscr{P}) \subset PC_{\mathscr{P}} \times PC_{\mathscr{P}} \qquad \text{and} \qquad bind(\mathscr{P}) \subset ADV_{\mathscr{P}} \times PC_{\mathscr{P}},$$

where $PC_{\mathscr{P}}$ is the set of pointcuts and $ADV_{\mathscr{P}}$ is the set of advice defined in $\mathscr{P}$. It is thus possible to compute all modified and referenced pointcut definitions for a given piece of advice in the delta set for two given program versions.

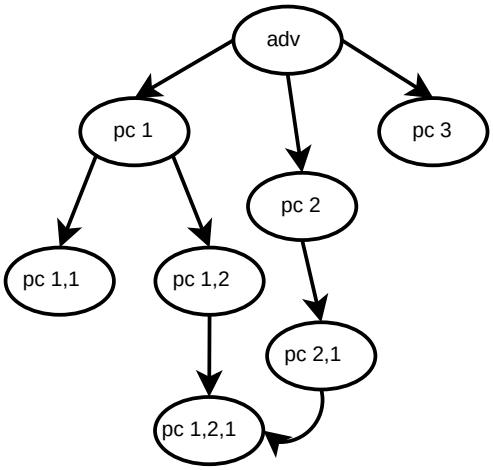
For the special case of hierarchical pointcut dependencies present in AspectJ, the presentation of the differences is best presented to the user in an annotated graph. The union of $reference(\mathscr{P})$ and $bind(\mathscr{P})$ defines a directed acyclic graph reflecting the syntactic dependencies of advice and pointcut definitions.

$$G(\mathscr{P}) = reference(\mathscr{P}) \cup bind(\mathscr{P})$$

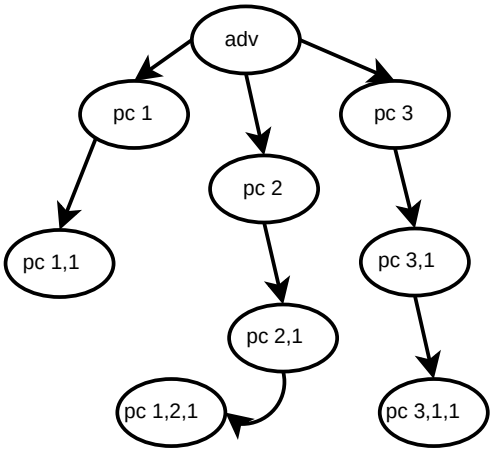
For each *adv* with a corresponding delta element $(jp, adv, \bullet)$ we calculate this graph. We then calculate the merged pointcut dependence graph.

$$\begin{aligned} G_{merged}(\mathscr{P}, \mathscr{P}') = & \\ & \{(a,b,+) | (a,b) \in G(\mathscr{P}') - G(\mathscr{P})\} \cup \\ & \{(a,b,-) | (a,b) \in G(\mathscr{P}) - G(\mathscr{P}')\} \cup \\ & \{(a,b,=) | (a,b) \in G(\mathscr{P}') \cap G(\mathscr{P})\} \end{aligned}$$

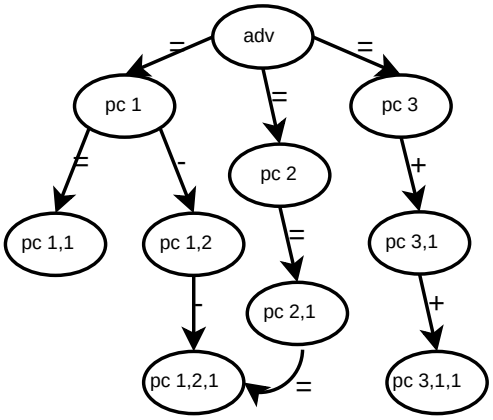
[9] For AspectJ, advice and pointcuts can again reference multiple other pointcuts (results are combined using logical operators not '!', or '||' and and '&&').



(a) Pointcut Dependences in $\mathscr{P}$



(a) Pointcut Dependences in $\mathscr{P}$



(c) Annotated Merged Pointcut Dependences

$$
\begin{aligned}
G_{merged}(\mathscr{P},\mathscr{P}') = \{ & \\
& (p_3, p_{3,1}, +), \quad (p_{3,1}, p_{3,1,1}, +), \\
& (p_1, p_{1,2}, -), \quad (p_{1,2}, p_{1,2,1}, -), \\
& (adv, p_1, =), \quad (adv, p_2, =), \\
& (adv, p_3, =), \quad (p_1, p_{1,1}, =), \\
& (p_2, p_{2,1}, =), \quad (p_{2,1}, p_{1,2,1}, =) \\
\} &
\end{aligned}
$$

(d) Set Representation of Merged Dependences

Figure 5.1: Analysis of Change in Pointcut dependences referenced by a given piece of advice. As can be seen in the example, the dependence graph in general is an acyclic directed graph. Figure (a) shows the dependences in the original program version $\mathscr{P}$, figure (b) the dependences in the edited program version. Figure (c) shows the merged dependence graph. Figure (d) finally shows the set representation of the merged dependences.

Example 5.3.1 (Structural Delta of Pointcut Dependences) *We illustrate this structural comparison with an example shown in Figure 5.1. While we do not show code matching this dependence of advice and pointcuts, this code can be easily generated, as the reader may verify. Note that the dependence graph for advice and pointcuts in general is an acyclic*[10] *rooted*[11] *graph. Sub-figure (a) shows syntactical pointcut dependences in the original program version, sub-figure (b) in the edited version and sub-figure (c) the merged dependence graph. Note that in sub-figure (c) the edges are labeled to show if a edge has been added (denoted by '+'), removed (denoted by '–') or is present in both program versions (denoted by '='). Figure (d) finally shows the resulting merged dependence graph (or rather the edge set)* $G_{merged}(\mathscr{P},\mathscr{P}')$ *in set representation.*

While $G_{merged}(\mathscr{P},\mathscr{P}')$ already gives the programmers hints to find out how a pointcut expression has been changed, we can further improve this information by also associating information about atomic changes referring to these pointcut definitions.

We will formulate the respective algorithms using functional pseudo code. Therefore we need the simple auxiliary function shown below, which transforms a set to a list (with arbitrary order).

```
1  listOf(∅, xs) = xs
2  listOf(S, xs) =
3    let s ∈ S and S' = S - {s}
4    in listOf(S', s:xs)
```

Definition 5.3.2 (Changes per Node) *For a referenced pointcut a with successors succ we derive the set of associated changes* Δ *as*

$$\texttt{changesForNode}(a, \texttt{listOf}(\{(a,b,\bullet)|(a,b,\bullet) \in G_{merged}(\mathscr{P},\mathscr{P}'), \mathscr{A}(\mathscr{P},\mathscr{P}'), \Delta),$$

where

```
1  changesForNode(a, [], 𝒜(𝒫,𝒫'), Δ) = (a,Δ)
2  changesForNode(a, (a,b,•):es, 𝒜(𝒫,𝒫'), Δ) =
3    case • ∈ {'-', '='} then
4      let Δ' = Δ ∪ {CAH(a), CP(a)} ∩ 𝒜(𝒫,𝒫')}
5      in changesForNode(a, es, 𝒜(𝒫,𝒫'), Δ')
6    otherwise
7      let Δ' = Δ ∪ {CAH(a), CP(a), AP(a)} ∩ 𝒜(𝒫,𝒫')}
8      in changesForNode(a, es, 𝒜(𝒫,𝒫'), Δ')
```

Function *changesForNode* augments each node in the structural delta $G_{merged}(\mathscr{P},\mathscr{P}')$ with potentially affecting changes. For example if we have a node a with a single outgoing edge $(a,b,+)$, the additional reference from a to b can only occur if a itself has been changed, thus we check if $\mathscr{A}(\mathscr{P},\mathscr{P}')$ contains a respective change (change pointcut $\mathbf{CP}(a)$ or change advice header $\mathbf{CAH}(a)$, as we do not know whether we deal with pointcut or advice). As the new reference can be due to the addition of a reference to a itself, we also check for add pointcut $\mathbf{AP}(a)$ changes in case of $+$.

We traverse the dependence graph using depth first search to capture all relevant changes. We want the derived set of changes to fulfill two properties: (i) no relevant change should be missing (i.e. we want a conservative solution), (ii) the associated sets should be as small as possible. To achieve this goal, we stop to traverse a path once we reach a removed node (changes further down are irrelevant). Function *getPCChanges* formalizes this strategy.

[10] Recursive pointcut definitions are not allowed in AspectJ.

[11] The respective piece of advice always is the root node.

Definition 5.3.3 (Pointcut Changes) *For a given merged annotated delta graph $G_{merged}(\mathscr{P},\mathscr{P}')$ and a piece of advice adv, we associate pointcut definitions reachable from adv with respective atomic changes by calling*

$$\texttt{getPCChanges}(G_{merged}(\mathscr{P},\mathscr{P}'),[adv],\Delta Set,\mathscr{A}(\mathscr{P},\mathscr{P}'))$$

where

```
1  getPCChanges(G_merged(P,P'), [], ΔSet, A(P,P')) = ΔSet
2  getPCChanges(G_merged(P,P'), a:as, ΔSet, A(P,P')) =
3    let succ = listOf({b|(a,b,•) ∈ G_merged(P,P') ∧ • ≠ '-'}
4    and edges = listOf({(a,b,•)|(a,b,•) ∈ G_merged(P,P') ∧ • ≠ '-'}
5    and ΔSet' = ΔSet ∪
6        {changesForNode(a, edges, ∅, A(P,P'))} in
7    getPCChanges(G_merged(P,P'), succ@as, ΔSet', A(P,P'))
```

and '@' denotes the list concatenation operator.

To illustrate the above concepts, we show how the two defined functions work in practice. Therefore we apply them to the annotated delta graph we showed in Figure 5.1.

Example 5.3.2 (Associating Atomic Changes) *In Figure 5.1 we derived $G_{merged}(\mathscr{P},\mathscr{P}')$ = $\{(p_3,p_{3,1},+),\ (p_{3,1},p_{3,1,1},+),\ (p_1,p_{1,2},-),\ (p_{1,2},p_{1,2,1},-),\ (adv,p_1,=),\ (adv,p_2,=),\ (adv,p_3,=),\ (p_1,p_{1,1},=),\ (p_2,p_{2,1},=),\ (p2,1,p_{1,2,1},=)\}$ there. We assume that $\mathscr{A}(\mathscr{P},\mathscr{P}') = \{\mathbf{CP}(p_1),\ \mathbf{CP}(p_{1,2}),\ \mathbf{DP}(p_{1,2}),\ \mathbf{CP}(p_{1,2,1}),\ \mathbf{CP}(p_3),\ \mathbf{AP}(p_{3,1}),\ \mathbf{CP}(p_{3,1}),\ \mathbf{AP}(p_{3,1,1}),\ \mathbf{CP}(p_{3,1,1})\}$ in the following. Calling getPCChanges($G_{merged}(\mathscr{P},\mathscr{P}')$, [adv], ΔSet, $\mathscr{A}(\mathscr{P},\mathscr{P}')$) then yields the following calculation[12]:*

$$
\begin{array}{ll}
adv: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[adv],\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[p_1,p_2,p_3], edges=[(adv,p_1,=),(adv,p_2,=),(adv,p_3,=)], \\ \Delta Set^1=\emptyset\cup changesForNode(adv,edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))=\{(adv,\emptyset)\} \end{array}\right) \\
p_1: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_1,p_2,p_3],\Delta Set^1,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[p_{1,1}], edges=[(p_1,p_{1,1},=)], \Delta Set^2=\Delta Set^1\cup \\ changesForNode(p_1,edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))=\Delta Set^1\cup\{(p_1,\{\mathbf{CP}(p_1)\})\} \end{array}\right) \\
p_{1,1}: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_{1,1},p_2,p_3],\Delta Set^2,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[], edges=[], \Delta Set^3=\Delta Set^2\cup \\ changesForNode(p_{1,1},[],\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))=\Delta Set^2\cup\{(p_{1,1},\emptyset)\}) \end{array}\right) \\
p_2: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_2,p_3],\Delta Set^3,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[p_{2,1}], edges=[(p_2,p_{2,1},=)], \Delta Set^4=\Delta Set^3\cup \\ changesForNode(p_2,edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))=\Delta Set^3\cup\{(p_2,\emptyset)\}) \end{array}\right) \\
p_{2,1}: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_{2,1},p_3],\Delta Set^4,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[p_{1,2,1}], edges=[(p_{2,1},p_{1,2,1},=)], \Delta Set^5=\Delta Set^4\cup \\ changesForNode(p_{2,1},edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))=\Delta Set^4\cup\{(p_{2,1},\emptyset)\}) \end{array}\right) \\
p_{1,2,1}: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_{1,2,1},p_3],\Delta Set^5,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[], edges=[], \Delta Set^6=\Delta Set^5\cup \\ changesForNode(p_{1,2,1},edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))= \\ \Delta Set^5\cup\{(p_{2,1},\{\mathbf{CP}(p_{1,2,1}\}))\}) \end{array}\right) \\
p_3: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_3],\Delta Set^6,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[p_{3,1}], edges=[(p_3,p_{3,1},+)], \Delta Set^7=\Delta Set^6\cup \\ changesForNode(p_3,edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))=\Delta Set^6\cup\{(p_3,\{\mathbf{CP}(p_3)\})\}) \end{array}\right) \\
p_{3,1}: & getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_{3_1}],\Delta Set^7,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \\
& \left(\begin{array}{l} succ=[p_{3,1,1}], edges=[(p_{3,1},p_{3,1,1},+)], \Delta Set^8=\Delta Set^7\cup \\ changesForNode(p_{3,1},edges,\emptyset,\mathscr{A}(\mathscr{P},\mathscr{P}'))= \\ \Delta Set^7\cup\{(p_{3,1},\{\mathbf{AP}(p_{3,1}),\mathbf{CP}(p_{3,1})\})\}) \end{array}\right)
\end{array}
$$

[12] We added a column showing the currently processed node in the call graph (Figure 5.1) to improve readability of the calculation. Steps shown in parenthesis show nested sub calculations.

$$p_{3,1,1}: \quad getPCChanges(G_{merged}(\mathscr{P},\mathscr{P}'),[p_{3_1,1}],\Delta Set^8,\mathscr{A}(\mathscr{P},\mathscr{P}')) = \left(\begin{array}{l} succ = [], edges = [], \Delta Set^9 = \Delta Set^8 \cup \\ changesForNode(p_{3,1,1}, edges, \emptyset, \mathscr{A}(\mathscr{P},\mathscr{P}')) = \\ \Delta Set^8 \cup \{(p_{3,1,1}, \{\mathbf{AP}(p_{3,1,1}), \mathbf{CP}(p_{3,1,1})\})\} \end{array}\right)$$

$$result: \quad \Delta Set = \left\{\begin{array}{l} (adv,\emptyset),(p_1,\{\mathbf{CP}(p_1)\}),(p_{1,1},\emptyset),(p_2,\emptyset),(p_{2,1},\emptyset), \\ (p_{1,2,1},\{\mathbf{CP}(p_{1,2,1})\}),(p_3,\{\mathbf{CP}(p_3)\}),(p_{3,1},\{\mathbf{AP}(p_{3,1}),\mathbf{CP}(p_{3,1})\}), \\ (p_{3,1,1},\{\mathbf{AP}(p_{3,1}),\mathbf{CP}(p_{3,1})\}) \end{array}\right\}$$

Note that for p_1, the set of successors does not contain $p_{1,2}$, as this association has been deleted in the edited program version $\mathscr{P}'$. We do not collect changes in removed subgraphs as any match defined by the now removed subgraph is lost due to the change of the root node $\mathbf{CP}(p_1)$.

However, note that (transitive) successors not traversed from p_1 are possibly traversed via another path in the graph. An example is node $p_{1,2,1}$ (and the associated change $\mathbf{CP}(p_{1,2,1})$*), which is traversed as a successor of $p_{2,1}$.*

5.3.3 Base Code Edits

Finally delta entries can also be due to modifications of the base code, as such edits can result in addition or removal of joinpoints matches by existing pointcut expressions.

- Additionally matched *new joinpoints* could be unexpected matches due to program extensions or rename/move operations and should be further examined.
- If a *joinpoint* has been *removed* from the program, this might be a lost match due to rename/move or deleted statements. This should be examined (also in the context of additional matches) to re-add the lost match if appropriate.
- If the *joinpoint is present in both versions*, the reason for a changed matching behavior must either be a pointcut modification or additional/removed advice (as captured by the first two cases).

In this context atomic changes allow us to capture these effects. To associate pointcut delta tuples with respective atomic changes, we exploit the information stored in the joinpoint signatures as described in Table 5.1.

A joinpoint signature references the program item a joinpoint is contained in and also the program item this joinpoint references. If for a delta tuple $(jp, adv, \bullet)$ the qualifier $\bullet$ is either $+$ or $-$ (either statically or dynamically), we can check if program items referenced by the joinpoint *jp* have either been added, changed or deleted. We formalize this idea in the following.

Definition 5.3.4 (Changes Affecting a Program Item) *Let p be a program item in $\mathscr{P}$, i.e. a method, initializer, etc. We then write* `getChanges`(p) *to refer to* add item, change item *and* delete item *changes directly affecting this program item.*

The above definition—on purpose—does not explicitly define the set of atomic changes affecting each given program item. We left this definition out as it is rather intuitive. For each program item there are in general changes indicating their addition, change or removal. For methods for example we have **AM**, **CM** and **DM** changes, for initializers **AI**, **CI**, and **DI** changes.

Definition 5.3.5 (Associated Base Code Changes) *Let $(jp, adv, \bullet) \in pcDelta$ be a pointcut delta tuple. Let $e = env(jp)$ be the joinpoint environment We then calculate the atomic changes associated with jp as*

$$(jp, \Delta) = \texttt{getChanges}(e)$$

5.3.4 Explained Delta Set

To summarize, a delta entry $(jp, adv, \bullet)$ is associated with changes affecting the bound advice, a structured delta of pointcut definitions by analyzing *adv* and its referenced pointcuts, and finally with information about new or removed joinpoints.

Thus the programmer gets detailed information *if and why* joinpoint matching behavior has changed. This information considerably helps when trying to find the reasons for failures due to changed pointcut semantics. We finally define function `explainDelta`, which collects all the above results.

Definition 5.3.6 (Explained pointcut Delta) *For each delta tuple* $(jp, adv, \bullet) \in pcDelta(\mathscr{P}, \mathscr{P}')$*, we define the associated* explained delta tuple *as*

$$\begin{aligned}&((jp, \texttt{getChanges}(env(jp))),\\ &(adv, \texttt{getAdvChanges}(jp, adv, \bullet), \mathscr{A}(\mathscr{P}, \mathscr{P}')),\\ &\texttt{getPCChanges}(G_{merged}(\mathscr{P}, \mathscr{P}'), [adv], \emptyset, \mathscr{A}(\mathscr{P}, \mathscr{P}')),\\ &\bullet).\end{aligned}$$

Informally, the explained delta tuple consists of four components. First, we associate the affected joinpoint with base code edits, second the affected advice with advice changes, third we add information about changed pointcut dependences and finally we also state the kind of joinpoint change.

With this structured association of atomic changes at hand, the programmer has considerably more information to examine and explain experienced changes in pointcut matching behavior.

5.3.5 Base Code Edits and Dynamic Predicates

Note that source code changes potentially change the value of dynamic predicates, i.e. even if a tuple $(jp, adv, dynamic)$ is present for both the original and the edited program version, we cannot be sure that the matching semantics did not change (compare e.g. AspectJ's pointcut designators `if`, `this` or `target`). As dynamic predicates are conservatively approximated, such effects are not visible and consequently our analysis is oblivious to such changes—a potential match is present for both program versions.

Although our analysis is incomplete in presence of dynamic pointcut constructs, it is nevertheless useful.

- For static joinpoint selection languages such effects are captured by the calculation of *match*, thus our analysis is complete in these cases.
- Finally, even if our tool misses these rare cases, in general our tool offers considerable support compared to the manual approach in the frequently occurring cases.

5.4 Case Studies

We implemented this pointcut delta analysis as part of the AOPA tool suite. An initial preliminary prototype has been implemented as a student project by Christian Koppen, and Jürgen Graf finally refined this prototype by also implementing a *Chianti*-like calculation of atomic changes and integrated this analysis as his master project [5]. He also considerably helped in gathering the data these case studies are based on.

We used the resulting *AJDiff*-plug-in to analyze the AspectJ example programs `Telecom` and `SpaceWar` as well as several of the programs available in the *abc* test suite. As for these

programs only a single version is available, we could only compare different build configurations. To counter this problem, we additionally performed two larger case studies using `AJHotDraw` [21], and `HSQLDB`.[13] For a more detailed discussion of our subjects refer to Chapter 7.

5.4.1 AspectJ Examples

Our first subjects for evaluation are the AspectJ example projects, more precisely `Telecom` (see also Section 6.2) and `SpaceWar`. These projects are small and easy to understand, thus the way how our tool derived results can be manually verified.

Unfortunately the CVS repository for the AspectJ examples does not contain different versions of the examples, thus our analysis can only be run against *different build configurations*. As expected we only found changes due to the addition or removal of advice and aspects.

`Telecom` Example—Comparing *basic* and *timing*.

We start with comparing the *basic* and the *timing* build configurations of the `Telecom` example. Compared to the minimal *basic* build configuration, *timing* adds class `Timer` and aspects `Timing` and `TimerLog`. Furthermore the test driver class `BasicSimulation` is replaced with class `TimingSimulation`. This addition resulted in the set of changes shown in Figure 5.2.

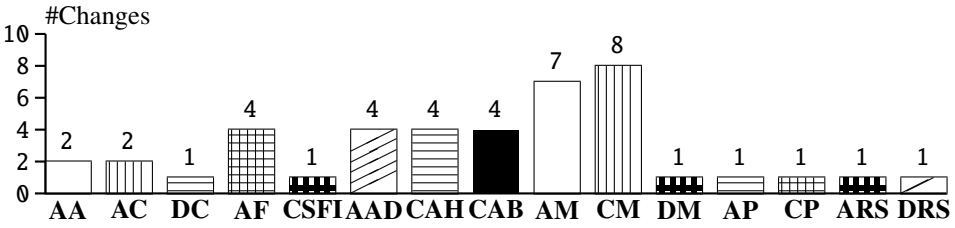


Figure 5.2: Atomic Changes for `Telecom` *basic* → *timing*.

Note that, compared to the set of changes we derived in Section 4.5.8, this set of changes is considerably more detailed, as we also capture more change categories—comparable to *Chianti*—and not only aspect induced changes necessary for the impact analysis proposed earlier. This especially includes changes derived from **AM**, **AAD**, and **AP** changes. Similarly to *Chianti*, we also generate a **CM** change for each **AM** change, if the body of the added method is non-empty. Similarly we also generate **CP** changes for **AP** changes and **CAB** and **CAH** changes for **AAD** changes. We could omit these changes for the impact analysis as they were irrelevant in their special aspect context. For example changes to initializers are irrelevant for the impact analysis, as fields added by an **AF** change, which are also referenced by e.g. a piece of advice, also resulted in a **CM** change which is directly captured when traversing the call graph. It is not possible to only *change* an initializer, without also introducing the field. For the purpose of a pointcut delta analysis however, such changes are relevant, as the change to the initializer might result in new or lost adapted joinpoints. Note however that the set of changes we derive for the pointcut delta analysis is no subset of the impact analysis changes, as we do not generate changes to model adapted joinpoints here. Such changes are only relevant for the impact analysis and not directly relevant here.

When examining the pointcut delta for the `Telecom` example we note four additional, statically adapted joinpoints in build configuration *timing* compared to *basic*, two in regular methods, and two within advice bodies. Figure 5.3 shows how changes are associated with this delta to explain it. In each case—not very surprising—our analysis found that the addition of a new piece of advice is the reason for the newly adapted joinpoint.

[13] http://hsqldb.org/

```
Aspect Timing Advice [timing-1]:
  Changed matches (new sure: 1)
  Reason: → AA(telecom.Timing<aspect> public)
    < AAD(telecom.Timing.[timing-1].after(telecom.Connection) public)

Aspect Timing Advice [timing-2]:
  Changed matches (new sure: 1)
  Reason: → AA(telecom.Timing<aspect> public)
    < AAD(telecom.Timing.[timing-2].after(telecom.Connection) public)

Aspect TimerLog Advice [timerlog-2]:
  Changed matches (new sure: 1)
  Specific reason at joinpoint telecom.Timing.after() while
    call to method void telecom.Timer.stop():
  → AA(telecom.Timing<aspect> public)
    < AAD(telecom.Timing.[timing-2].after(telecom.Connection) public)
    < CAB(telecom.Timing.[timing-2].after(telecom.Connection) public)
  Reason: → AA(telecom.TimerLog<aspect> public)
    < AAD(telecom.TimerLog.[timerlog-2].after(telecom.Timer) public)

Aspect TimerLog Advice [timerlog-1]:
  Changed matches (new sure: 1)
  Specific reason at joinpoint telecom.Timing.after() while
    call to method void telecom.Timer.start():
  → AA(telecom.Timing<aspect> public)
    < AAD(telecom.Timing.[timing-1].after(telecom.Connection) public)
    < CAB(telecom.Timing.[timing-1].after(telecom.Connection) public)
  Reason: AA(telecom.TimerLog<aspect> public)
    < AAD(telecom.TimerLog.[timerlog-1].after(telecom.Timer) public)
```

Figure 5.3: Explained Pointcut Delta for *basic* → *timing*.

The first two delta entries show the two additional matches for the two pieces of advice defined in `Timing`, which advise the call to `Connection.complete()` in method `Call.pickup()` and `Connection.drop()` in `Call.hangUp()`. The second two delta entries refer to advice defined by aspect `TimerLog`. Here, statements `timer.start()` in the first `after`-advice and `timer.stop()` in the second `after`-advice of `Timing` are advised by respective pieces of advice defined by aspect `TimerLog`. As for each delta entry a new piece of advice is found (none of the four pieces of advice is present when using the *basic* build configuration), the addition of the two aspects and the respective pieces of advice, modeled by the respective **AA** and **AAD** changes, are reported as reasons for the change in matching behavior. Note that for the two pieces of advice defined in `TimerLog` we also get the **CAB** change resulting from the addition of the target advice as additional potential delta reason. Figure 5.4 shows a screen shot of the Eclipse view presenting this output to the user.

`Telecom` Example—Comparing *basic* and *billing*.

The second example is a comparison of the *basic* and the *billing* build configurations of the `Telecom` example. Compared to the minimal *basic* build configuration, *billing* adds class `Timer` and aspects `Timing` and `Billing` (but *not* `TimerLog`). Furthermore the test driver class `BasicSimulation` is replaced with class `BillingSimulation`. These changes resulted in the set of changes shown in Figure 5.5.

We experience five additional, statically adapted joinpoints in build configuration *billing*

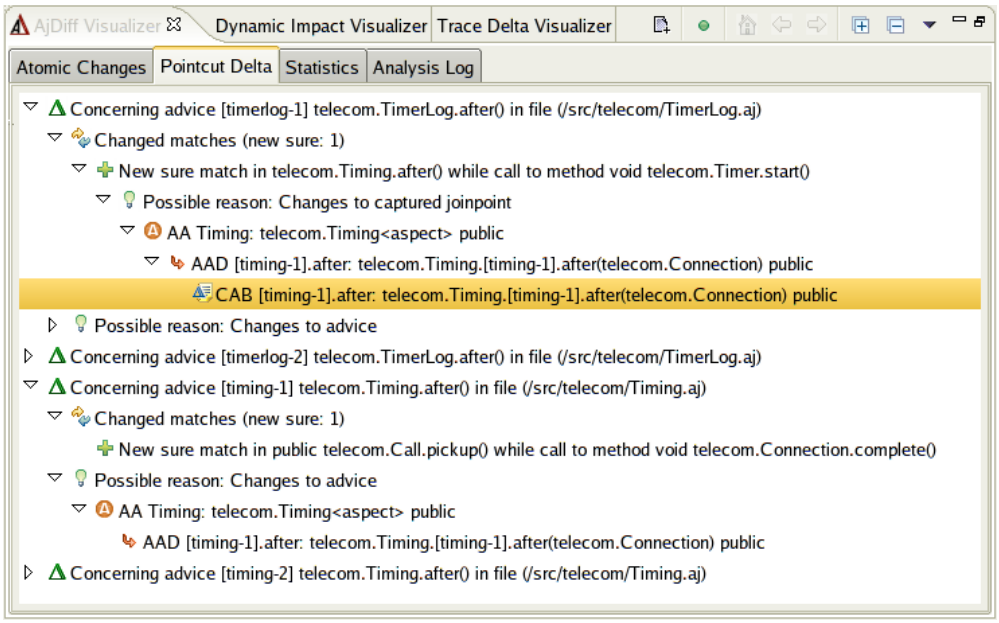


Figure 5.4: Screen Shot of AOPA presenting the Explained Pointcut Delta for the `Telecom` example comparing build configurations *basic* and *timing*.

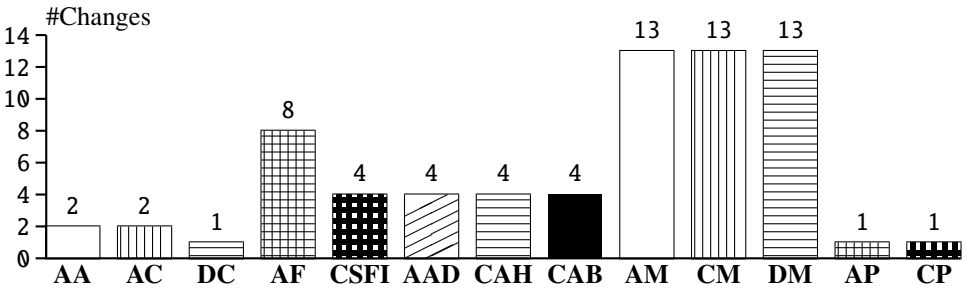


Figure 5.5: Atomic Changes for `Telecom`, *basic* → *billing*.

compared to *basic*, all of them in regular methods. Figure 5.6 shows how changes are associated to this delta to explain it. Similarly to our first example our analysis found that the addition of the two aspects and the new pieces of advice are the reason for the newly adapted joinpoints. The first two delta entries show the two additional matches for the two pieces of advice defined in `Timing`, which advise the call to `Connection.complete()` in method `Call.pickup()` and `Connection.drop()` in `Call.hangUp()`, as seen before. The second two delta entries show the new matches of the `Billing` advice. Note that for the first entry for `Billing` actually refers to two delta entries. The `after returning`-advice in `Billing` is bound to constructor calls for both class `Local` and class `LongDistance`. However, for both delta entries the same reason, i.e. the addition of aspect `Billing` and the `after returning`-advice is derived by our analysis. The second entry for `Billing` refers to the `after`-advice bound to pointcut `endTiming`. However, as a different (second!) piece of advice is bound to this pointcut, we also get a different tuple entry. Again for each delta entry a new piece of advice is found (none of the four pieces of advice is present when using the *basic* build configuration), thus—similarly to the first example—the addition of the two aspects and the

```
Aspect Billing Advice [billing-1]:
  Changed matches (new sure: 2)
  Reason: → AA(telecom.Billing<aspect> public)
    < AAD(telecom.Billing.[billing-1].afterReturning(Customer, Connection))

Aspect Billing Advice [billing-2]:
  Changed matches (new sure: 1)
  Reason: AA(telecom.Billing<aspect> public)
    < AAD(telecom.Billing.[billing-2].after(telecom.Connection) public)

Aspect Timing Advice [timing-1]:
  Changed matches (new sure: 1)
  Reason: AA(telecom.Timing<aspect> public)
    < AAD(telecom.Timing.[timing-1].after(telecom.Connection) public)

Aspect Timing Advice [timing-2]:
  Changed matches (new sure: 1)
  Reason: AA(telecom.Timing<aspect> public)
    < AAD(telecom.Timing.[timing-2].after(telecom.Connection) public)
```

Figure 5.6: Explained Pointcut Delta for *basic* → *billing*.

respective pieces of advice, modeled by the respective **AA** and **AAD** changes, is reported as reason for the change in matching behavior. Figure 5.7 shows a screen shot of the Eclipse

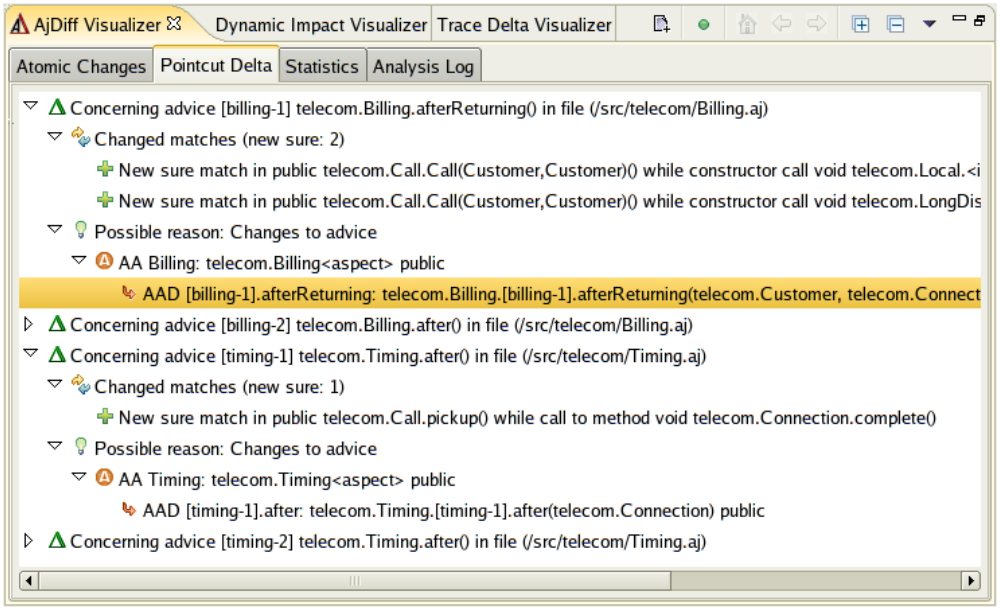


Figure 5.7: Screen shot of AOPA presenting the Explained Pointcut Delta for the `Telecom` example comparing build configurations *basic* and *billing*.

view presenting this output to the user.

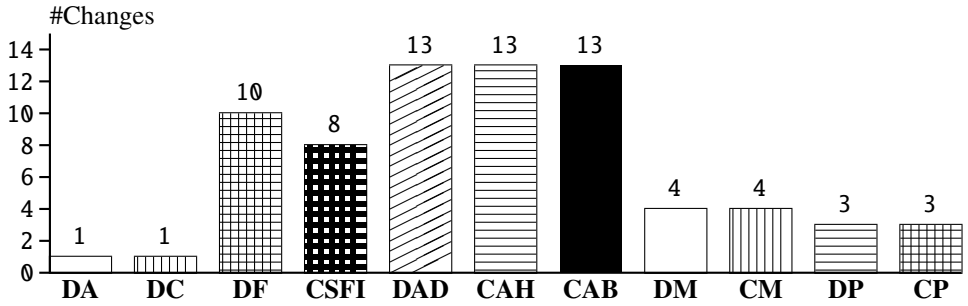


Figure 5.8: Atomic Changes for `SpaceWar`, *debug* → *demo*.

`SpaceWar` Example—Comparing *debug* and *demo*.

For the `SpaceWar` example we compare the two build configurations *debug* and *demo*. Note that in contrast to the previous example we now remove an aspect. However, as we still only have a single version, delta entries are still only due to addition and removal of aspects. Compared to build configuration *debug*, file `Debug.aj` is omitted in the *demo* build configuration. This file contains the definition of aspect `Debug` and of class `InfoWin` (a simple text window to display debug output). Figure 5.8 gives an overview of the resulting changes.

While we will not discuss the results for `SpaceWar` in detail, we want to point out that this example is the first where we experienced changes in matching behavior for dynamic joinpoints. Compared to the `Telecom` example, the only difference is that `SpaceWar` defines more pieces of advice applying at more joinpoints and thus has more entries in the delta set. However, again each entry is straightforward matched with removed advice.

5.4.2 The *abc* Test Suite

As the pointcut delta analysis is a static analysis technique, we could use the *abc* test suite to evaluate our prototype. We present the results in this section. However, for brevity and to avoid repetitions, we only point out interesting peculiarities for the different case studies and otherwise only present the remaining data, i.e. the resulting changes and explained deltas in interesting cases. Comparable to the `Telecom` example, the *abc* test suite is not available in different versions and thus in general deltas in general are due to addition or removal of aspects. For a more detailed discussion of the programs in the *abc* test suite, refer to Section 7.1.2.

The `Bean` Example

For this example we compared the two versions resulting from comparing a build with and without aspect `BoundPoint`. To make the non-aspectized version compilable, we also had to remove two lines in the `Demo.main(..)` method, which resulted in a respective **CM** change (i.e. we have an (artificial) base code edit in this case). Figure 5.9 gives an overview of the

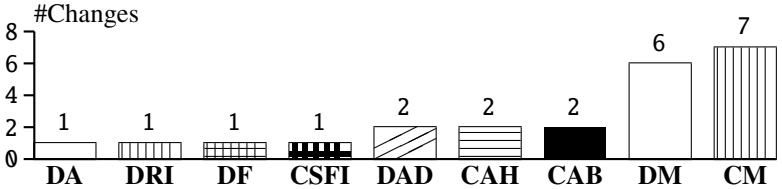


Figure 5.9: Atomic Changes for `Bean`, removal of `BoundPoint`.

resulting changes. Note the relative high number of **CM** and **DM** changes, which are due to the various inter type declarations to `Point` to add and remove property change listeners (there is one additional **CM** change due to the modification of `main(..)`).

```
Aspect BoundPoint Advice bean.BoundPoint.around.#1:
  Changed matches (lost sure: 2)
  Specific reason at joinpoint Demo.main(String[]) while
    call to method void Point.setY(int):
  → CM (Demo.main(String[]))
  Reason:
  → DA (bean.BoundPoint) < DAD (bean.BoundPoint.around.#1)

Aspect BoundPoint Advice bean.BoundPoint.around.#2:
  Changed matches (lost sure: 2)
  Specific reason at joinpoint Demo.main(String[]) while
    call to method void Point.setX(int):
  → CM (Demo.main(String[]))
  Reason:
  → DA (bean.BoundPoint) < DAD (bean.BoundPoint.around.#2)
```

Figure 5.10: Explained Pointcut Delta for *Bean*, removal of aspect `BoundPoint`.

For the `Bean` example removal of aspect `BoundPoint` results in 4 lost static matches, 2 of which can be associated with each removed piece of advice. However, considering Figure 5.10 also shows that for each advice, one joinpoint is associated with additional information. For this joinpoint, the **CM** change we introduced to make the non-aspectized version compilable is also reported as a potential change, as method `Demo.main(..)` also contains calls to `setX` and `setY` which are adapted by advice. In this case this is spurious information, however it gives a first impression how our analysis also captures base code edits (it is also possible that the two `set*` statements in `main(..)` had been deleted; in that case the **CM** change would have been the delta reason).

The `DCM` Example

For the `DCM` example we examined the effect of the removal of aspect `AllocFree`. Figure 5.11 shows the resulting changes. Note the high number of **DRI** changes. This is due to aspect `AllocFree` using the marker interface idiom to add a `finalize()` method to all classes in the system. Therefore it declares that all types implement the interface `DCM.handleGC.Finalize` and introduces a finalize method to this interface.

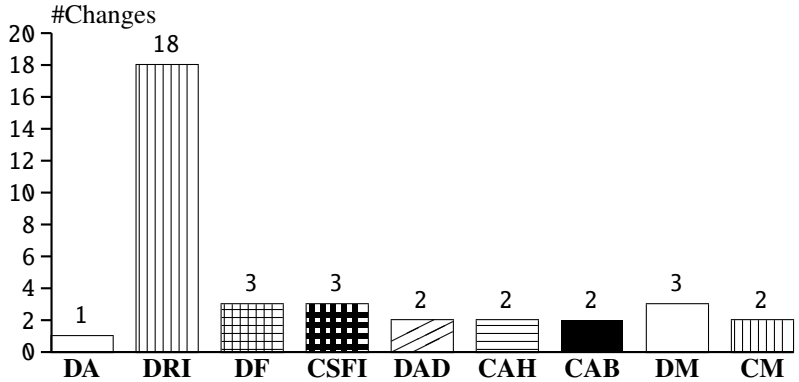


Figure 5.11: Atomic Changes for `DCM`, removal of aspect `AllocFree`.

```
Aspect EnforceCodingStandards Advice EnforceCodingStandards.around.#1:
  Changed matches (lost sure: 75)
  Specific reason at joinpoint EnforceCodingStandards.around() while
    call to method String StringBuffer.toString():
  Specific reason at joinpoint EnforceCodingStandards.around() while
    call to method StringBuffer StringBuffer.append(String):
  Specific reason at joinpoint EnforceCodingStandards.around() while
    call to method StringBuffer StringBuffer.append(String):
  Specific reason at joinpoint EnforceCodingStandards.around() while
    call to method Signature JoinPoint$StaticPart.getSignature():
  → DA (EnforceCodingStandards)  < DAD (EnforceCodingStandards.around.#1)
    < CAB (EnforceCodingStandards.around.#1)
  Reason:
  → DA (EnforceCodingStandards<aspect>)
    < DAD (EnforceCodingStandards.around.#1)
```

Figure 5.13: Explained Pointcut Delta for *NullCheck*, removal of aspect `EnforceCodingStandards`.

For this example removal of the aspect results in 41 delta matches. One of the two pieces of advice in `AllocFree` is associated with only 2 of those 41 delta tuples (this piece of advice only prints a summary). The second piece of advice is associated with 39 entries (these are the adapted constructor calls). For both pieces of advice the deletion of the aspect is identified as the delta reason.

The `NullCheck` Example

For the `NullCheck` example we removed aspect `EnforceCodingStandards` to generate the second version. This aspect defines a single piece of advice which is associated with 75 joinpoints. Figure 5.12 gives an overview of the resulting changes.

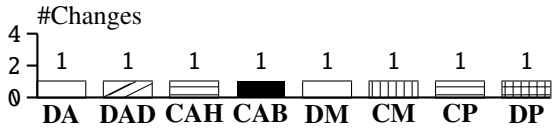


Figure 5.12: Atomic Changes for `NullCheck`, removal of aspect `EnforceCodingStandards`.

An interesting peculiarity is that this aspect also adapts 4 joinpoints within its own body. This is interesting as this case can lead to infinite recursion if calls within the advice actually return a `null` value (which is however not the case). For these 4 joinpoints within the advice body, we get a detailed feedback, as can be seen in Figure 5.13. For these joinpoints, the change of the advice body is additionally reported as a potential delta reason, as the body edit potentially removed the respective joinpoints. For all 75 joinpoints our analysis determines the removal of the aspect as a delta reason.

The `LawOfDemeter` Example

For the `LawOfDemeter` example we examined the effect of adding the whole `lawOfDemeter` package, which includes 5 files in total. Figure 5.14 shows the resulting changes. Note the relative high number of **AP** and **CP** changes. This is due to abstract class `Any`, which only defines a set of pointcuts. The intention is to provide named pointcuts for the relevant calls examined by the checker. The `LawOfDemeter` example is interesting as several of the pointcuts use dynamic pointcut designators. Figure 5.15 shows the results of the pointcut delta analysis.

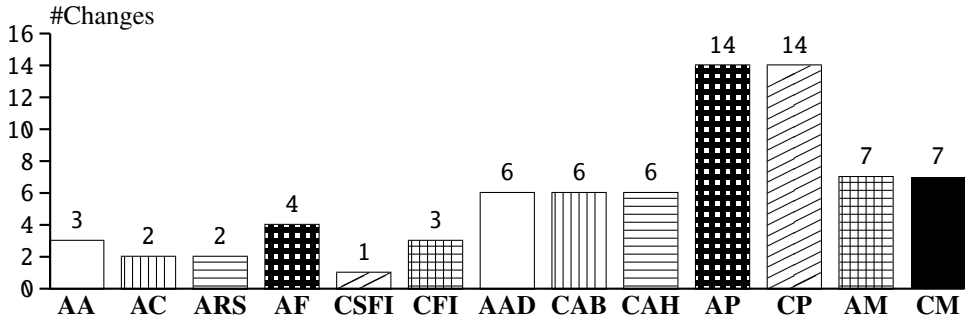


Figure 5.14: Atomic Changes for `LawOfDemeter`, addition of package `lawOfDemeter`.

```
Aspect Percflow Advice #1:
    Changed matches (new possible: 154)
    Reason: AA(lawOfDemeter.objectform.Percflow<aspect>)
      < AAD(lawOfDemeter.objectform.Percflow.#1.before())
Aspect Percflow Advice #2:
    Changed matches (new possible: 374)
    Reason: AA(lawOfDemeter.objectform.Percflow<aspect>)
      < AAD(lawOfDemeter.objectform.Percflow.#4.afterReturning(Object))

Aspect Check Advice #1:
    Changed matches (new sure: 33)
    Specific reason at joinpoint lawOfDemeter.objectform.Check.after()
      while get field java.io.PrintStream System.out:
        AA(lawOfDemeter.objectform.Check<aspect>)
          < AAD(lawOfDemeter.objectform.Check.#3.after())
            < CAB(lawOfDemeter.objectform.Check.#3.after())
    Reason: AA(lawOfDemeter.objectform.Check<aspect>)
      < AAD(lawOfDemeter.objectform.Check.#6.afterReturning(Object))
Aspect Check Advice #2:
    Changed matches (new possible: 97)
    Reason: AA(lawOfDemeter.objectform.Check<aspect>)
      < AAD(lawOfDemeter.objectform.Check.#2.after(Object, Object))
Aspect Check Advice #3:
    Changed matches (new sure: 2)
    Reason: AA(lawOfDemeter.objectform.Check<aspect>)
      < AAD(lawOfDemeter.objectform.Check.#3.after())

Aspect Pertarget Advice #1:
    Changed matches (new possible: 212)
    Reason: AA(lawOfDemeter.objectform.Pertarget<aspect>)
      < AAD(lawOfDemeter.objectform.Pertarget.#5.before(Object))
```

Figure 5.15: Explained Pointcut Delta for *LawOfDemeter*, addition of package `lawOfDemeter`.

Note that the law of Demeter checking concern is attached to a multitude of joinpoints. Several of these are also dynamic. The law of Demeter aspects use `cflow` as well as `if` and the type check designators `this` and `target`.

The piece of advice `lawOfDemeter.objectform.Check.#3.after()` is of special interest, as we here again see a statement within advice which is also advised by the same concern. While this might be intended in some cases here it is most likely a spurious match. For

Version	applied Aspects
0	NoPrinting, Undirected, Weighted
1	version 0 + Benchmark, ProgTime
2	version 1 + DFS, Cycle, CC
3	version 2 + MSTKruskal

Table 5.2: Analyzed configurations for ProdLine.

all other matches the addition of the aspects and the respective pieces of advice is identified as delta reason.

The ProdLine Example

The ProdLine example contains a larger set of optional aspects, thus we can derive a set of different build configurations to compare. We chose to add groups of aspects starting with a minimal configuration (version 0), then adding the development aspects Benchmark and ProgTime (version 1). This configuration is then extended by adding aspect DFS, Cycle and CC. The final setup contains all aspects. We thus yield the versions shown in Table 5.2.

We start with comparing version V_0 and version V_1. For version V_1, aspects Benchmark and ProgTime have been added to the system compared to version V_0. Aspect Benchmark uses inter type declarations to add several benchmarking methods to class graph. Aspect ProgTime only defines a main()-method for a profiled example execution of the system. None of the two aspects contains advice. Figure 5.16 shows the resulting set of atomic changes.

```
Aspect Weighted Advice #1:
    Changed matches (new sure: 1)
    Specific reason at joinpoint public GPL.ProgTime.main(String[]) while
    call to method void GPL.Graph.addAnEdge(GPL.Vertex, GPL.Vertex, int):
        AA(GPL.ProgTime<aspect> public)
        < AM(GPL.ProgTime.main<void>(java.lang.String[]))
        < CM(GPL.ProgTime.main<void>(java.lang.String[]))
```

Figure 5.17: Explained Pointcut Delta for *ProdLine*, comparing versions V_0 and V_1.

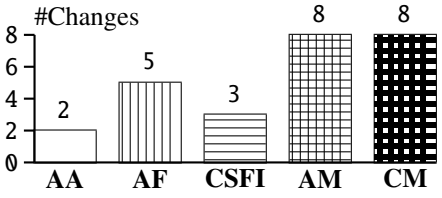


Figure 5.16: Atomic Changes for ProdLine, comparing versions V_0 and V_1.

It is interesting to note that we nevertheless observe an additional match in the pointcut delta, as can be seen in Figure 5.17. This is actually a witness for (intended?) aspect interference. Aspect Weighted contains advice which intercepts calls to method addEdge(..) (located in method addAnEdge(..)) which originally adds an unweighted edge (although the interface contains a weight parameter[14]) and instead adds a weighted edge. Our pointcut delta analysis correctly determines that the **CM** change for the new method main(..) which contains this call to addEdge is the reason for this new match.

[14]Note that this can be solved better, by providing a (legacy) interface which adds unweighted edges using some default weight (using advice) and provide an explicit interface for weighted edges. The current implementation forces users to specify an edge weight even when using unweighted graphs.

We continue by examining version V_1 and V_2. For version V_2, we add aspects `DFS`, `CC` and `Cycle`. All three aspects encapsulate an optional implementation of the respective algorithm by using inter type declarations and advice. Figure 5.18 shows the resulting set of atomic

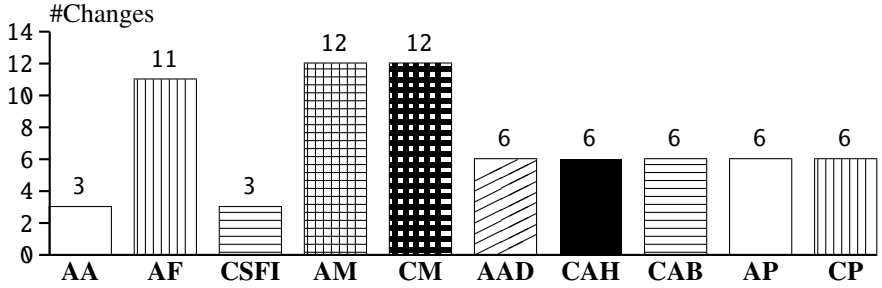


Figure 5.18: Atomic Changes for `ProdLine`, comparing versions V_1 and V_2.

changes. The resulting pointcut delta is again rather small (in total only 5[15] delta entries), and each delta entry is due to the addition of the respective aspect/piece of advice.

We finally examine version V_2 and V_3. For version V_3, we added the optional aspect `MSTKruskal`. Figure 5.19 shows the resulting set of atomic changes. Addition of the single aspect `MSTKruskal` only results in three delta entries, however one of them is an additional match of aspect `Weighted`, as this aspect now also captures a call to method `adjustAdorns()` in the newly introduced constructor `Graph.Kruskal()`.[16]

The `Tetris` Example

We finally examine the `Tetris` example. For this example, we compared the versions with no aspects applied (V_0), all feature aspects applied (V_1) and finally all aspects applied ($V2$)

Version V_1 adds 6 aspects in total to the system, where each aspect contains several pieces of advice. Figure 5.20 gives an overview of the resulting changes.

While most of the resulting changes are straight forward new matches due to the addition of the adapting piece of advice, several of the resulting additional matches in the pointcut delta are more interesting. Figure 5.21 shows those delta entries.

We start with the discussion of the piece of advice defined by the `Levels` aspect. This aspect increases the speed of the dropping blocks depending on the amount of lines already deleted by the player. As the `Counter`-aspect keeps track of this number, aspect `Levels` directly references joinpoints of the `Counter` aspect which are identified by our analysis. Our tool thus shows the dependence of aspect `Levels` on aspect `Counter`. Note that this is no syntactic dependence—aspect `Levels` can be applied to the system without compiler errors, even without aspect `Counter` applied. It just does not work in this case.

A similar situation becomes apparent for aspects `NewBlocks` and `NextBlock`. Aspect `NextBlock` modifies the system by adding a preview of the next block. To do so, aspect `NextBlock` overrides the creation of the next block, caches and displays the next block

[15]Note that there were two additional spurious delta entries due to a bug in *ajdt*, which we reported. The bug also resulted in displaying erroneous matching information in the Eclipse editor.

[16]Unfortunately, due to the above mentioned *ajdt*-bug we did not get accurate matching information in this case but were only able to identify the delta rather than explain it.

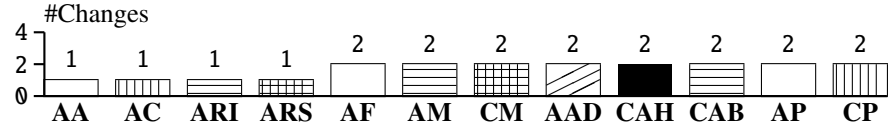


Figure 5.19: Atomic Changes for `ProdLine`, comparing versions V_2 and V_3.

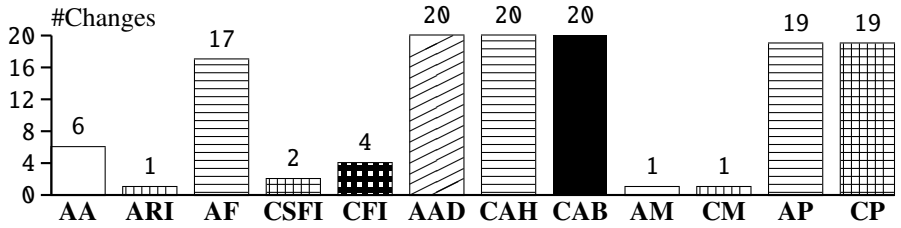


Figure 5.20: Atomic Changes for `Tetris` comparing version $V_0 \rightarrow V_1$.

and—when the base system demands a new block—returns the cached block instead. As a consequence block creation is now called from two locations: the base system and aspect `NextBlock`. Now aspect `NewBlocks` adds two new types of blocks to the game. As a consequence this aspect has to make sure that both locations where new blocks are created call the new behavior introduced by the aspect, resulting in the respective additional joinpoint matches. For the base system, finding the reason for these new matches is straight forward (the respective **AAD** change), however we also have matches in `NextBlock`. As the respective pieces of advice have also been added, addition of them is also part of the delta reason. Note that this shows the interaction of aspects `NextBlock` and `NewBlocks`.

We finally also add the development aspects defined by the system. Aspect `DesignCheck` only contains two `declare warning` statements and thus does not result in any changes or delta entries. Aspect `TestAspect` contains a pointcut definition and a single piece of advice. As both the resulting changes and also the explanation are straight forward, we will not describe the results in detail here.

5.4.3 Auction System

We compared the build configuration of the auction system (see Section 7.1.3) where only aspect `AuctionSystemExceptionHandler` has been applied with the configuration where all 9 available aspects have been applied.

As only a single version of the auction system is available, all delta entries could be explained with the addition or removal of the respective aspect/piece of advice.

Summarized, the `AuctionUpdateNotificationAgent` results in 6 delta entries, aspect `CreditLog` in 8 delta entries, the `RefreshAuctionStateAspect` in 3 delta entries, the `DataManagementAspect` in 2 delta entries, the `TestAspect` in 210 delta entries and finally the `MaintainAuctionIDAspect` in 5 delta entries. All delta entries are static additions of adapted joinpoints and can be directly related to the respective added aspect/piece of advice.

While the results of this case study do not provide any additional insights we report the results for completeness reasons. Note however that the results we got here confirm that most aspects (except the `TestAspect`) are very specific aspects with only few adapted joinpoints.

5.4.4 `AJHotDraw`

We compared the three available versions of `AJHotDraw` with each other. As version 0.1 only contains aspects in the tests, 0.2 only adds a persistence aspect without advice, and version 0.3 only adds an Observer aspect the results of this study are unfortunately less interesting as we expected. As `AJHotDraw` represents the result of refactoring the `JHotDraw` system, the three versions are only separated by the changes necessary to encapsulate a crosscutting concern into an aspect, but not by changes due to system evolution. `AJHotDraw` is the largest publicly available aspect-oriented system, so we nevertheless present the result of our analysis here.

We first discuss the changes between version 0.1 and 0.2, as presented in Figure 5.22. The change profile gives an interesting overview of what happened in between these two

```
Aspect Levels Advice #1:
    Changed matches (new sure: 3)
    Specific reason at joinpoint Counter while
    set field int Counter.totalLines:
        AA(Counter<aspect>)
        < AF(Counter.totalLines<int> protected)
    Specific reason at joinpoint Counter.after() while
    set field int Counter.totalLines:
        AA(Counter<aspect>)
        < AAD(Counter.#1.after()) < CAB(Counter.#1.after())
    Specific reason at joinpoint Counter.before() while
    set field int Counter.totalLines:
        AA(Counter<aspect>)
        < AAD(Counter.#2.before()) < CAB(Counter.#2.before())
    Reason: AA(Levels<aspect>)
        < AAD(Levels.#1.after(int))
...

Aspect NewBlocks Advice #1:
    Changed matches (new sure: 4)
    Specific reason at joinpoint NextBlock.after() while
    call to method int[][] Logic.Blocks.getBlock(int):
        AA(NextBlock<aspect>)
        < AAD(NextBlock.#1.after())
        < CAB(NextBlock.#1.after())
    Specific reason at joinpoint NextBlock.around() while
    call to method int[][] Logic.Blocks.getBlock(int):
        AA(NextBlock<aspect>)
        < AAD(NextBlock.#2.around(AroundClosure))
        < CAB(NextBlock.#2.around(AroundClosure))
    Specific reason at joinpoint NextBlock.around() while
    call to method int[][] Logic.Blocks.getBlock(int):
        AA(NextBlock<aspect>)
        < AAD(NextBlock.#2.around(AroundClosure))
        < CAB(NextBlock.#2.around(AroundClosure))
    Reason: AA(NewBlocks<aspect>)
        < AAD(NewBlocks.#1.around(int, AroundClosure))

Aspect NewBlocks Advice #2:
    Changed matches (new sure: 3)
    Specific reason at joinpoint NextBlock.after() while
    get field int Logic.Blocks.NUMBEROFTYPES:
        AA(NextBlock<aspect>)
        < AAD(NextBlock.#1.after())
        < CAB(NextBlock.#1.after())
    Specific reason at joinpoint NextBlock.around() while
    get field int Logic.Blocks.NUMBEROFTYPES:
        AA(NextBlock<aspect>)
        < AAD(NextBlock.#2.around(AroundClosure))
        < CAB(NextBlock.#2.around(AroundClosure))
    Reason: AA(NewBlocks<aspect>)
        < AAD(NewBlocks.#2.around(AroundClosure))
```

Figure 5.21: Explained Pointcut Delta for *Tetris*, comparing versions V_0 and V_1.

versions. Compared to version 0.1, six aspects have been added to the system in version 0.2 (represented by the 6 **AA** changes). For selected subclasses of `AbstractFigure`, methods `read(..)` and `write(..)` implementing persistent storage for an object of the class have been removed from the class these methods originally were defined in and instead have been moved to an respective persistence aspect (represented by 10 **DM** changes). The aspect then re-introduces these methods (10 **AM** change). Furthermore aspect `FigureEquivalence` has been modified, the introduced method `equivalent()` has been moved from `Storable` to `Figure` (1 **DM** and 1 **AM** change) to avoid an AspectJ bug. In this context, the signatures of the `make()` methods also changes, resulting in 5 more **AM** and 5 more **DM** changes.

Of the remaining 51 changes, 28 are due to the 16 **AM** and **DM** changes, respectively (2 added methods are abstract), but the remaining 33 **CM** changes are adaptations of the base system to match the new system structure. Moving the 10 `read`-methods to the aspects results in 10 delta entries, as these methods were adapted by the invariants checking aspect. It is interesting to note that the new inter type methods in the aspect are not captured by pointcut `methodsWithInvariant`, i.e. here relevant joinpoints have been indeed lost.

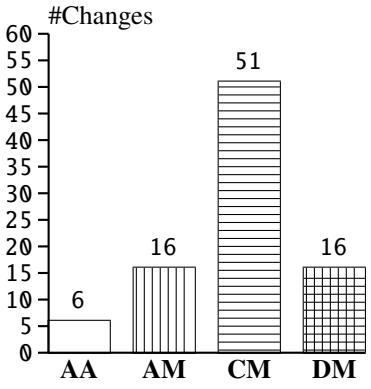


Figure 5.22: Atomic Changes for `AJHotDraw`, comparing versions 0.1 and 0.2.

Comparing version 0.2 and 0.3 yields more interesting results. Version 0.3 adds 8 new test classes with several new test methods and fields, two interfaces and 5 aspects total, where three aspects contain advice (`CmdCheckViewRef` (1), `FigureSelectionSubjectRole` (2) and `SelectionChangeNotification` (2)). Again there is a considerable amount of inter type declarations, although the high number of **AM** changes is mostly due to added test methods. Figure 5.23 gives an overview of the resulting changes.

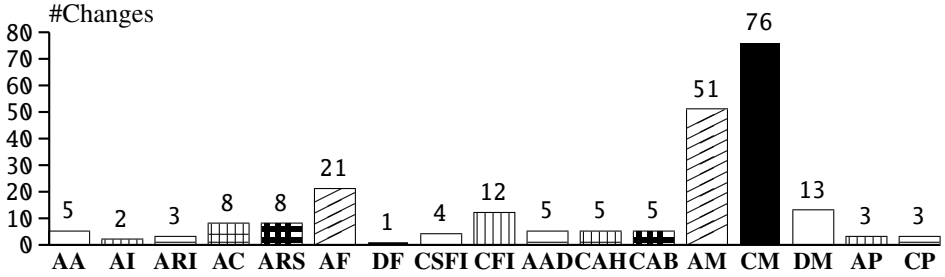


Figure 5.23: Atomic Changes for `AJHotDraw`, comparing version 0.2 and 0.3.

We again experience a high number of **AM**, **CM**, and **DM** changes. Most of these changes result from added and evolving base classes, but a considerable amount of them is also due to introduction of the observer concern (11 **AM** changes). The whole observer administration (i.e. the listener collection plus method to add and remove listeners, etc.) has been removed from the base system and instead implemented in aspects which reintroduce the respective methods to the base system using inter type declarations. The observer concern also defines and uses a common interface (`FigureSelectionListener`) as a common type for listeners and uses AspectJ's `declare parents ...implements` construct to attach the interface to the respective listener classes, represented by the 3 **ARI** changes. Actually there are 6 `declare parents ...implements` statements, although only 3 have an effect (the remaining three target classes implement interface `DrawingEditor` which is declared to be a sub-interface of `FigureSelectionListener` by the aspect—i.e. these are superfluous

declarations).

We finally discuss the pointcut delta for comparing versions 0.2 and 0.3 of AJHotDraw. The explained delta is shown in Figure 5.24. The single piece of advice declared in aspect

```
Aspect CmdCheckViewRef Advice #1:
  Changed matches (new sure: 18)
  Specific reason at joinpoint method contrib.zoom.ZoomCommand.execute():
    CM(contrib.zoom.ZoomCommand.execute<void>())
  ... (repeated with respective CM change for each delta entry) ...
  Specific reason at joinpoint method util.UndoCommand.execute():
    CM(util.UndoCommand.execute<void>())
  Reason: AA(cmdcontracts.CmdCheckViewRef<aspect>)
    < AAD(cmdcontracts.CmdCheckViewRef.#1.before(AbstractCommand))

Aspect SelectionChangedNotification Advice #1:
  Changed matches (new sure: 2)
  Specific reason at joinpoint method
    StandardDrawingView.toggleSelection(Figure):
    CM(StandardDrawingView.toggleSelection<void>(Figure))
  Specific reason at joinpoint method StandardDrawingView.clearSelection():
    CM(StandardDrawingView.clearSelection<void>())
  Reason: AA(SelectionChangedNotification<aspect>)
    < AAD(SelectionChangedNotification.#1.after(StandardDrawingView))
Aspect SelectionChangedNotification Advice #2:
  Changed matches (new sure: 2)
  Specific reason at joinpoint StandardDrawingView.removeFromSelection(Figure)
  while call to method void Figure.invalidate():
    CM(StandardDrawingView.removeFromSelection<void>(Figure))
  Reason: AA(SelectionChangedNotification<aspect>)
    < AAD(SelectionChangedNotification.#2.after(StandardDrawingView))

Aspect FigureSelectionSubjectRole Advice #1:
  Changed matches (new sure: 1)
  Specific reason at joinpoint method
    StandardDrawingView.readObject(java.io.ObjectInputStream):
    CM(StandardDrawingView.readObject<void>(java.io.ObjectInputStream))
  Reason: AA(FigureSelectionSubjectRole<aspect>)
    < AAD(FigureSelectionSubjectRole.#1.after(StandardDrawingView))
Aspect FigureSelectionSubjectRole Advice #2:
  Changed matches (new sure: 1)
  Specific reason at joinpoint constructor
    StandardDrawingView.StandardDrawingView(DrawingEditor,int,int)():
    CM(StandardDrawingView.StandardDrawingView<init>(...))
  Reason: AA(FigureSelectionSubjectRole<aspect>)
    < AAD(FigureSelectionSubjectRole.#2.after(StandardDrawingView))
```

Figure 5.24: Explained Pointcut Delta for AJHotDraw, versions 0.2 and 0.3.

CmdCheckViewRef matches 18 joinpoints. As general reason the addition of this new piece of advice is reported by our analysis. However, for each delta entry we also find a **CM** change in the method containing the adapted joinpoint, which is also reported.

For the two observer aspects the reported results are similar. However it is interesting to note that both aspects define very specific advice—each piece of advice only matches 1 respective 2 joinpoints. As we expected more matches for an observer, we further investigated and discovered that the observer updating, i.e. calls to fireSelectionChanged() and even

the method itself have been moved to the aspect. This method is now called by advice which is bound to `Figure.invalidate()` and other relevant methods. These methods represent a suitable hook for the observer. As they are directly called by the base system, relatively few adapted joinpoints suffice.

Observations: Although `AJHotDraw` yields less interesting results than expected—given that it is the largest publicly available AspectJ system—our study nevertheless showed one interesting thing: Examining the pure changes between two versions can give interesting support for programmers in understanding changes between two versions.

Note that we were not familiar with the `AJHotDraw` code base when starting this study. However, by reviewing the changes, we were in general able to give a good estimate about the nature and underlying purpose of the changes. While it is too early to state that there are certain *change profiles* matching certain development tasks, we nevertheless noticed that inter type declarations usually result in many **AM**, and **CM** changes (and also **AF** changes). For refactorings involving inter type declarations we additionally observed **DM** (and **DF**) changes. Note that we observed changes to the base system in that case. Adding optional aspects in contrast results in relative few changes (equally distributed **AM**, **CM**, **AF**, and **AAD**, **CAH**, and **CAB** changes), but no changes in the base code (which seems reasonable). It might be interesting to further study if we can actually derive change profiles characteristic for a certain development task.

5.4.5 HSQLDB

`HSQLDB` as described in Section 7.2 is by far the most interesting program to study. For this system we have several different program versions with an increasing number of aspects applied to the system as crosscutting implementations are refactored into aspects. Additionally, some of these aspects still evolved after introduction to the system.

Beside the multiple available versions, a test suite of both JUnit unit tests and also standard regression tests is available which also allows to execute the system in a controlled way. As these are JUnit tests, we also have test results, i.e. tests either PASS, FAIL or CRASH. Some of the available versions include failing tests.

Version $V_1 \leftrightarrow V_2$: Exception Handling

We start with a discussion of the comparison of versions V_1 and V_2 of `HSQLDB`. Note that the available versions have a considerable editing distance. Due to the considerable size of the resulting pointcut delta, we will not show the complete results but only show the set of derived changes and discuss interesting delta entries in the following.

For version V_2, exception handling has been refactored into (in part inner, in part global) aspects. Figure 5.25 gives an overview of the resulting set of changes. Note the high number of **CM** changes. Theses changes are usually due to the removal of crosscutting code (in this case the exception handlers) from the original base code. The change profile also shows, that a considerable amount of aspects has been added (11), however that in general only few pieces of advice per aspect (16) exist. Often exception handlers have been moved to *inner aspects* with just a single piece of advice (depending on the number of different exceptions handlers within a given class).

For this large edit, the pointcut delta consists of 93 new matched, 2409 new potential matches, 649 lost matches, and 36 potentially lost matches. Note that the number of joinpoints affected by a single piece of advice varies greatly. Some pieces of advice only affect a single joinpoint, however there is also single piece of advice affecting 1720 joinpoints. There are two interesting cases to discuss. First, for a piece of advice defined in aspect `org.hsqldb.aspects.TracingAbstractAspect` a potentially matched joinpoint is

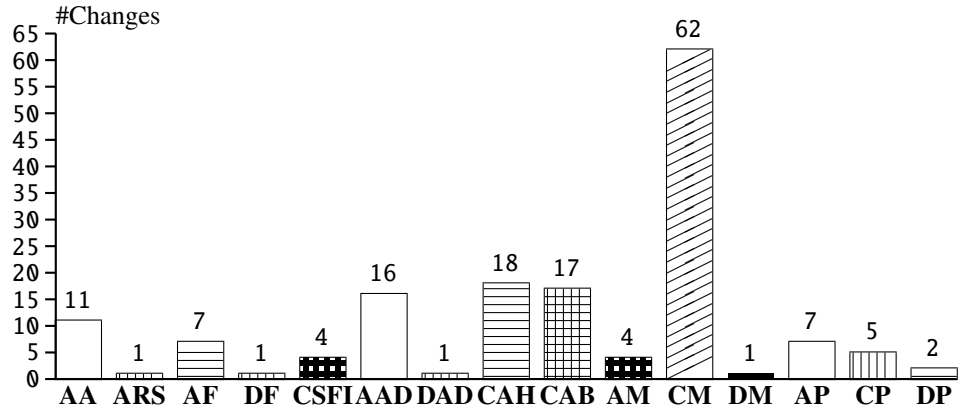


Figure 5.25: Atomic Changes for HSQLDB, comparing versions V_1 and V_2 (refactoring exception handling).

no longer affected, and 9 joinpoints are additionally affected. Investigating this change in matching behavior shows that a refactored exception handler had been adapted by tracing advice. Our tool correctly identifies the respective **CM** change as reason for this lost match. The 9 new matches in turn are associated with **CAB** changes for new pieces of advice. There, new log statements are adapted by the new tracing aspect. Second, there is also a piece of advice with 684 lost matches. This is due to the deletion of this piece of advice. This is interesting as we did not expect lost matches when we conducted this case study. However, examining the respective piece of advice showed that this piece of advice has been used to discover exception handlers, and thus became obsolete after refactoring exceptions. The reminder of the delta entries above can be directly associated with the addition or removal of the respective piece of advice.

The comparison of the two versions of HSQLDB showed two things. First, that the benefit of our delta analysis depends on the size of the delta. For this case study, we experienced a delta with 3187 entries—too much to check manually. However, 3094 of these delta entries are concentrated in just two aspects, 3012 of them even in just 3 pieces of advice, and can be classified as the prototypical example for aspect-orientation—(exception) logging. The remaining aspects are considerably more specialized and it is also feasible to examine the delta manually.

From this observation we can derive a recommendation how to examine pointcut deltas. We suggest that the programmer should examine the delta *per aspect* or even *per advice*. The resulting delta sets—with the few exceptions outlined above—can be checked manually. Furthermore not all delta entries are in general interesting. If we add aspects, we assume that we match additional joinpoints. However it might be interesting if—as a side effect of edits (as experienced in this case study)—previously matched joinpoints are lost. Such information can be easily generated by our tool by applying respective filters to the joinpoints delta. For example it seems reasonable to only show deltas associated with a certain aspect. Another possible filter in this context is "*show delta entries which can not be associated with base code edits only*".

Versions $V_2 \leftrightarrow V_3 \leftrightarrow V_4$: Value Pooling

We proceed with a discussion of the comparison of versions V_2, V_3, and V_4 of HSQLDB. As is typical for this case study, the available versions again have a considerable editing distance, thus we will again only report aggregated data and interesting peculiarities.

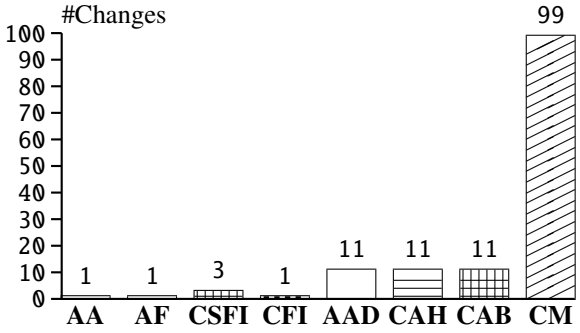


Figure 5.26: Atomic Changes for HSQLDB, comparing version V_2 and V_3 (refactoring value pooling).

For version V_3, value pooling has been refactored into a single inner aspect of class `ValuePool`. Figure 5.26 gives an overview of the resulting set of changes. Note again the high number of **CM** changes. Theses changes are due to replacing direct access to the value pool with ordinary constructors. For example statements like `ValuePool.getDouble(d)` for some `double`-value `d` are replaced with explicit constructor calls (`new Double(d)`). These constructor calls are then advised by `around`-advice from aspect `ValuePoolingAspect`. In total there are 11 pieces of advice for relevant types of non-mutable data objects.

The delta consists of a total of 447 entries. Of these, 437 entries are expected additional matches due to the refactoring described above. However, 10 additional matches remain. These additional matches show up in aspect `ExceptionHandlingAspect`. The adapting aspect counts the direct references to class `ValuePool`, i.e. this is simply *debug code* placed inside this (exception handling?) aspect, although the aspect itself has nothing to do with exception handling (this piece of advice is the only code in this aspect, it is otherwise empty). Indeed the complete aspect has been deleted in a later version.

Note that rerunning the `HSQLDB` test suite reveals a test failure for version V_3. This bug has been removed in version V_4 by completely removing variable `GrantConstants.INTEGER_ALL`. Advising this interface constant initialization (as done by aspect `ValuePoolingAspect`) resulted all tests to CRASH. However, although the introduction of this bug is related to the introduction of aspect `ValuePoolingAspect`, it is not a consequence of an faulty pointcut definition but rather due to a bug in AspectJ itself. Removing the offending line however still resulted in test `testDoubleNaN` to FAIL. This failure however was due to a malformed replacement of base code and not related to a pointcut delta. The derived set of changes was nevertheless useful in finding the failure inducing statement, however this might also have been accomplished by using a simple Unix `diff` tool.

In total the comparison of versions V_3 and V_4 results in 15 changes, 11 of them **CM** changes. The new version also introduces a piece of advice to move the resetting of the `ValuePool` to the aspect. Removal of constant `GrantConstants.INTEGER_ALL` and introduction of the piece of advice resetting the value pool resulted in two delta entries. We only mention this comparison for completeness reasons, as it does not give any additional insights.

Analyzing the introduction of value pooling demonstrated that our tool can be used to discover unexpectedly matched joinpoints, in this case superfluous matches due to debug code. While the addition of the `ValuePoolingAspect` aspect lead to a multitude of matches (437), all these matches were expected. For the remaining 10 unexpected matches our tool allowed to easily trace them back to the respective inducing code edit.

Versions $V_4 \leftrightarrow V_5$: Intermezzo

We call the comparison of these two versions of `HSQLDB` intermezzo, as no new concern has been refactored, but existing aspect code has been improved/reconfigured. Most changes are in aspect `TracingAbstractAspect` which is now better configurable as constants to set the desired level of tracing have been added. Beside these modifications a new aspect `UtilAspect` has been added to the system which includes two pieces of advice for debugging/refactoring help. Figure 5.27 gives an overview of the resulting changes.

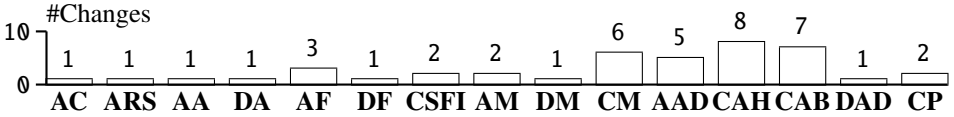


Figure 5.27: Atomic Changes for HSQLDB, comparing version V_4 and V_5 (intermezzo).

Nevertheless there are 126 pointcut delta entries due to the introduction/change of tracing constants (98 delta entries), and the addition of aspect `UtilAspect` (17 delta entries). The latter aspect now also contains the piece of debugging advice from the (now removed) aspect `ExceptionHandlingAspect` (10 lost matches), although deactivated (by adding `if(false)` to the pointcut). A single delta entry is interesting in this context. For aspect `ExceptionHandlingAbstractAspect` a piece of advice is attached to a call to method `TestUtil.print()` in `TestSelf.test()`. This lost match is due to the removal of the respective statement (and thus the joinpoints) as a side effect of base evolution.

As a simple experiment we reactivated the aspect in the context of the `UtilAspect` aspect. As a result, the 10 lost matches disappeared, i.e. our tool was able to match the joinpoint matches to this piece of advice although the advice has been moved to a different aspect.

The comparison of these two versions is interesting, as we could see a delta entry which results from a side effect of base code edits. Additionally, the small experiment showed that our tool is also robust with respect to moved advice.

Versions $V_5 \leftrightarrow V_6$: Caching

The `Caching` aspect is very similar to the pooling aspect we have seen before. However, in contrast to class `ValuePool` there was no respective class responsible for caching, but instead the caches were local to the respective classes using caching.

In order to properly refactor the caching concern in this code, the original base code had to be refactored with traditional means before. After this refactoring aspect `Caching` was introduced by removing the original maps used for caching from classes `Function` and `GranteeManager`. Figure 5.27 shows all resulting changes.

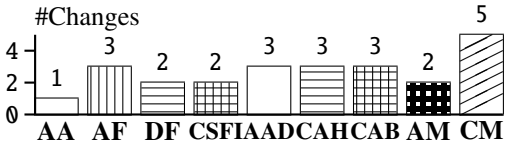


Figure 5.28: Atomic Changes for HSQLDB, comparing version V_5 and V_6 (refactoring caching).

The resulting pointcut delta is rather small—the new aspect `Caching` only adapts two joinpoints (exactly those were the caching is needed). However, interestingly there are 3 additional delta entries for aspect `ExceptionHandlingAbstract-Aspect`. Two of these delta entries can be easily dismissed, as they only show the relocation of an exception handler to the new method `getmMethod()` due to the necessary traditional refactoring before introducing the aspect. This new method `getmMethod()` however is also affected by a piece of advice bound to the execution of methods throwing an exception. While in this particular case the new advice is only logging the thrown exception (and thus the additional match is unproblematic) this might not be the case in general. This again shows that unexpected pointcut deltas due to base code edits can occur.

Although we only experienced relatively few changes in the comparison of these two versions, our tool again uncovered unexpected pointcut deltas due to base code edits.

Versions $V_6 \leftrightarrow V_7$: Authorization

We finally examine the refactoring of the authorization concern. While authentication happens exactly once in HSQLDB (the user has to log in, thus acquiring an authentication token),

authorization was implemented by statements scattered throughout the system. Beside authorization, the new version also considerably changes aspect `UtilAspect` and introduced aspect `TriggerFiring` to encapsulate scattered database trigger activation statements. Finally tracing has been reconfigures again by changing the value of a configuration constant. Similar to the value pool refactoring, removing the original authorization code from the system resulted in many **CM** changes. However here we also experienced a high number of **AM** and **DM** changes. This is due to the fact that many classes contained relatively simple `check..(..)`-methods to invoke authentication checks. These methods have been moved to the respective aspects (thus the **AM** changes), or in part they became superfluous and were simply deleted. The high number of aspects and advice can is due to implementing authorization not as a global aspect but as a set of local inner aspects. Figure 5.29 gives an overview of all resulting changes.

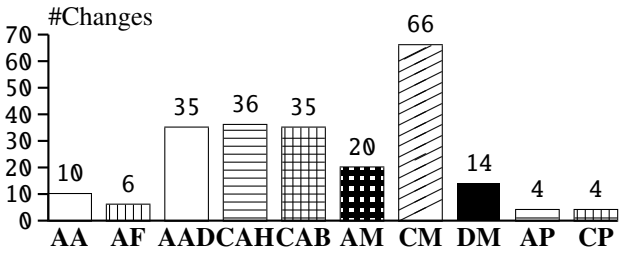


Figure 5.29: Atomic Changes for `HSQLDB`, comparing version V_6 and V_7 (refactoring caching).

The pointcut delta comprises 75 delta entries. 17 of these matches are lost matches due to modification of the advice header in aspect `UtilAspect` (added `&& if(false)` in the pointcut—and associated with the relevant **CAH** change), trigger activation affects 14 joinpoints (associated with the relevant **AA** and **AAD** changes), and the remaining delta entries are due to the newly added authorization aspects (and also associated with them). Although the delta set is not small, explaining the delta for this analysis is straightforward, as no aspect interference or base code edit side effects occur.

In contrast to the previous example, we have a large amount of changes in this example. These changes are also associated with a larger set of delta entries (75). Nevertheless we did not experience any unexpected delta entries in this case. Thus the last two examples demonstrate that unexpected entries do not depend on the size of the edit.

`HSQLDB`—Summary

We will briefly summarize the observations made during the `HSQLDB` case study. Comparing the different versions of `HSQLDB` resulted in both small and large change sets, i.e. we observed a considerable difference in the editing distance for the different versions we analyzed. The size of the delta set is similarly distributed, from a minimum of 5 delta entries up to 3187 entries. Although size of the edit in terms of changes and size of the delta set approximately correspond with each other in this case study, this may not be the case in general, just consider adding a new tracing advice. However, for a refactoring project, this correspondence might be valid, we usually have a change due to removed base code and also a delta entry as a joinpoint at the former code location is now matched.

It is interesting to note that we also experienced several delta entries due to base code edits (lost joinpoints) and also delta entries referring to joinpoints within advice. This demonstrates that the fragile pointcut problem is indeed relevant, and also shows that even aspects themselves are not safe, as they can also be targets of other aspects.

Finally we want to point out that `HSQLDB` is no toy example, but a real world database application. Nevertheless our analysis approach scales well, and also the results presented in this case study were accessible and informative for humans, even if several thousand delta entries were reported. This was achieved through the hierarchical organization of the delta entries which easily allowed to distinguish between expected and unexpected delta entries.

While the `HSQLDB` refactoring project yielded several other versions (in total 17), we

refrain from discussing all of them in detail here. For the following versions pointcut deltas could be explained straight forward by our prototype. The first seven versions are actually the most interesting ones for the purpose of this case study.

5.4.6 Case Studies—Conclusions

With the limited available data for case studies we could not show that the resulting tool is actually useful for developers, as for this one would actually need a user case study. However we already had serious trouble to find at least some objects to study let alone independent programmers who are willing to use our research prototype for their every day work.

Nevertheless the above data shows that the tool is useful to uncover and explain changes in the set of selected joinpoints for two program versions. We also showed that such changes actually occur in code. Especially as only few version pairs exhibit changes due to base code edits, our tool promises to be useful as these changes can be automatically detected and thus captured, thus potentially avoiding subtle program flaws.

We finally also got a good impression about the scalability of our approach. First, the performance of our tool is not a critical factor. Most time is currently spent in recompiling the two versions we compare. However this is only a technical restriction as our prototype in this way accesses the abstract syntax tree and the joinpoint model from the AspectJ compiler (these data structures are unfortunately partly discarded after compilation). Second, scalability is also an issue for the presented results. The case studies have shown that large editing distances can result in a considerable size of the pointcut delta. This obviously reduces the benefits of our tool. However, with appropriate filtering and focusing on relevant aspects we can deal with large delta sets.

5.5 Related Work

Compared to the prior work reported in [19, 20] the delta analysis proposed here has been considerably extended as we improved the delta analysis by adding the explanation of the resulting deltas using atomic changes. We also considerably extended our case studies, including new systems and the explanation of delta entries using atomic changes.

The AspectJ development tools `ajdt` [1] mainly visualize relations between aspects and base, but the most recent version also added support for pointcut deltas (called Crosscutting Comparison view). However the crosscutting view has no fine grained analysis on the level of atomic changes to explain the delta entries.

Our approach relies on a good approximation of dynamic pointcut designators. An approach to better approximate the `cflow` pointcut is presented in [18]. Partial evaluation [13] may also be useful to better approximate dynamic joinpoints.

Beside the work mentioned above we see our work related to many other efforts to improve program understanding, especially the work about Delta Debugging, Change Impact Analysis and the development of new pointcut languages.

5.5.1 Change Impact Analysis and Delta Debugging

The goal of Change Impact Analysis is to provide techniques to allow programmers to analyze the effects of changes they made. Examples are the work presented in [7, 14] or [15, 17]. In the latter work the edit between two program versions is decomposed in a set of Atomic Changes which we also use to explain delta entries.

Delta Debugging as introduced in [22] also focuses on finding failure inducing inputs or edits. However, this approach does not reveal any syntactical or semantical dependencies of the different program constructs as derived by our delta analysis. Second, Delta Debugging

relies on *executing* intermediate versions. This however might not be possible for software under development. Our approach statically analyzes both versions.

5.5.2 Improved Pointcut Languages

The improvement of the pointcut definition mechanism is an important research topic today. Several approaches have been proposed to attack the fragile pointcut problem using improved pointcut languages.

To reduce coupling, AspectJ [10] invented *abstract aspects*. These aspects can contain abstract pointcuts which are defined by inheriting aspects. Thus all the advice code is encapsulated in the abstract aspect and can be reused. The aspect can be applied to a concrete problem by inheriting from the abstract aspect and defining the pointcuts for the concrete base system. Unfortunately, although coupling is reduced, pointcuts in the concrete aspect still are fragile.

[6] proposes a logic pointcut language. In this language, a program is represented as a set of facts and pointcuts are defined in a Prolog like language as a query over these facts. Although this language is Turing-complete its expressions could be evaluated by our tool to acquire the necessary matching information. However, if a expression cannot be completely evaluated at compile time we would again have to conservatively approximate. However, as joinpoints are picked in a more semantical way pointcuts tend to be less fragile.

An approach in-between these two extremes proposes *declarative pointcuts*, a set of *descriptive pointcut designators* which allows to specify joinpoints by their (semantic) properties [9]. This approach reduces the necessity to reference names or source locations and thus considerably lightens the problem with fragile pointcuts. Unfortunately, although research produced first results [12] these pointcut designators are currently not widely available.

While we consider the improvement of pointcut languages important research, these languages will only lighten the problem *in the future* when the emerging constructs will become part of mainstream languages. However, by then we assume that there is a considerable amount of code written in e.g. AspectJ where evolution suffers from the problems outlined above - even if the goal of system evolution is the renewal of the pointcut definitions with new, more declarative constructs. Additionally, even if new constructs are available the old constructs will be kept for compatibility reasons for some time. For this code our approach is valuable.

5.6 Conclusions

This chapter in detail examined currently available joinpoint selection mechanisms used in state-of-the-art aspect-oriented languages. We showed that these mechanisms all suffer from the fragile pointcut problem. Furthermore we claim that a purely language-based solution is a very ambitious goal, although improvement of pointcut languages is a research topic and might well reduce the impact of the fragile pointcut problem one day.

To deal with currently available languages and software written in these languages we introduced an alternative tool-based approach: pointcut delta analysis is used to deal with changes in the set of actually selected joinpoints in an aspect-oriented program.

We showed that the calculated delta set together with associated responsible code changes can considerably help to reveal unexpected changes in the matching behavior of pointcuts by reporting the results of our case studies using our implementation. Thus the main contributions of this chapter are:

- A detailed analysis of the fragile pointcut problem as a major problem for evolution of aspect-oriented programs and aspect-orientation in general.

- The introduction of a pointcut delta analysis allowing to derive and explain differences in the set of selected joinpoints.
- Finally we also provided an implementation of our analysis as an Eclipse plug-in extending `ajdt` and examined the benefits of our tool in several case studies.

To conclude, although we only have few data points to evaluate our tool, the results are promising and suggest that our tool might well help to avoid introduction of bugs into an aspect-oriented system due to accidentally matched or lost joinpoint deltas during system evolution. Compared with the younger Crosscutting Comparison View of *ajdt* our tool provides considerably more detailed feedback by automatically explaining delta entries.

Bibliography

[1] ANDY CLEMENT, A. C., AND KERSTEN, M. Aspect-Oriented Programming with AJDT. In *Proceedings of AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003* (July 2003).

[2] ERLIKH, L. Leveraging legacy system dollars for e-business. *IT Professional 2*, 3 (2000), 17–23.

[3] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. Tech. rep., RIACS, 2000.

[4] FOWLER, M. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] GRAF, J. Eine feingranulare Multi-Versions Programmanalyse für AspectJ basierend auf dem Vergleich der Programmstruktur. Diplomarbeit, Universität Passau, FMI, Passau, Germany, Innstraße 32, 94032 Passau, March 2006.

[6] GYBELS, K., AND BRICHAU, J. Arranging Language Features for more robust Pattern-based Crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 60–69.

[7] HARROLD, M. J., JONES, J. A., LI, T., LIANG, D., ORSO, A., PENNINGS, M., SINHA, S., SPOON, S. A., AND GUJARATHI, A. Regression Test Selection for Java Software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), ACM Press, pp. 312–326.

[8] KERSTEN, M., AND MURPHY, G. C. Atlas: a Case Study in building a web-based Learning Environment using Aspect-Oriented Programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1999), ACM Press, pp. 340–352.

[9] KICZALES, G. The Fun has just begun. Keynote Speach AOSD 2003, Boston, March 2003.

[10] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An Overview of AspectJ. *Lecture Notes in Computer Science (LNCS) 2072* (2001), 327–355.

[11] LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. Characteristics of Application Software Maintenance. *Commun. ACM 21*, 6 (1978), 466–471.

[12] MASUHARA, H., AND KAWAUCHI, K. Dataflow pointcut in aspect-oriented programming. In *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings* (2003), vol. 2895 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag GmbH, pp. 105–121.

[13] MASUHARA, H., KICZALES, G., AND DUTCHYN, C. Compilation Semantics of Aspect-Oriented Programs. In *Proc of workshop Foundations Of Aspect-Oriented Languages (FOAL) held in conjunction with AOSD 2002*. 2002.

[14] ORSO, A., APIWATTANAPONG, T., LAW, J., ROTHERMEL, G., AND HARROLD, M. J. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In *Proc. of the Int. Conf. on Softw. Engineering (ICSE'04)* (Edinburgh, Scotland, 2004), pp. 491–500.

[15] REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 432–448.

[16] ROBERTS, D., BRANT, J., AND JOHNSON, R. A refactoring tool for smalltalk. *Theor. Pract. Object Syst. 3*, 4 (1997), 253–263.

[17] RYDER, B. G., AND TIP, F. Change Impact Analysis for Object-Oriented Programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2001), ACM Press, pp. 46–53.

[18] SERENI, D., AND DE MOOR, O. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development* (2003), ACM Press, pp. 30–39.

[19] STOERZER, M., AND GRAF, J. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 653–656.

[20] STOERZER, M., AND KOPPEN, C. Pcdiff: Attacking the fragile pointcut problem. In *Proceedings of European International Workshop on Aspect Software (EIWAS'04), Berlin, Germany* (2004).

[21] VAN DEURSEN, A., MARTIN, M., AND MOONEN, L. AJHotDraw: A Showcase for Refactoring to Aspects. In *In Proceedings of AOSD'05 Workshop on Linking Aspect Technology and Evolution* (2005).

[22] ZELLER, A. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99)* (Toulouse, France, 1999), pp. 253–267.

6

Static Analysis of Aspect Interference

One interesting idea of aspect-orientation is to implement standard crosscutting concerns as reusable aspects, which are then made available in aspect libraries. This allows aspects to provide pluggable functionality which can be added to a system on-demand.

Aspect interference is a major issue in this context, as it can easily break the functionality of independently working aspects, finally breaking the complete system. In this chapter we examine a special case of this problem which we term the *advice precedence problem*. We show that undefined advice precedence can easily jeopardize system correctness and provide an interference criterion to check for precedence related issues. We used static analysis techniques to implement this criterion and its usefulness by discussing the `Telecom` example. The interference criterion defined in this chapter has been published in [24].

6.1 The Advice Precedence Problem

Recently there was an interesting discussion on the AspectJ mailing list illustrating a major AOP problem: *aspect interference*. An AspectJ user had the following problem migrating his system to a new version of the AspectJ compiler[1]:

> "What I am seeing ... is that my aspects that previously worked for transaction control and database connections are no longer working. ... I can not stress enough that the only change was my migration from 1.2 variants of AspectJ and AJDT to the newest versions when this started to occur."

What changed in between these two compiler versions? In the absence of explicit user-defined aspect ordering advice precedence for two pieces of advice can be *undefined*. The above problem could be tracked down to a change in *compiler-specific precedence rules*. The new compiler chose a different order in cases where advice order was undefined, finally resulting in a program failure as advice is not commutative in general. In the same thread, AspectJ developers state that no guarantee can be given on any order picked by the compiler for undefined precedence, even that the order can change arbitrarily among different compiler versions or even for different compiler runs.

[1] [aspectj-users] AJDT 1.3 and aspectj; thread started by Ronald R. DiFrango on Oct. 10th, 2005

The advice-precedence related problem reported on the mailing list is a special case of a problem known as the *aspect interference problem* in AOP research [6]. We will term this above mentioned special case *the advice precedence problem* here. While one could argue that programmers should not rely on a particular order picked by a compiler, the problem is more substantial.

First, undefined semantics—while very common for languages like 'C'—have been explicitly avoided when creating Java. The Java Language Specification [12] explicitly defines the semantics of the complete language, thereby removing any degrees of freedom for the compiler. This is also one of the enablers of Java's platform independence. Note that AspectJ—as a Java extension—now considerably jeopardizes these efforts by reintroducing degrees of freedom for a compiler.

Second, in team developed projects different programmers potentially develop and test aspects *independently* of each other. As a consequence these programmers in general are not aware of other aspects let alone whether or not their aspects interfere. Third, due to the design of pointcuts as quantified expressions over a program, *evolution* of the base program may result in introduction of aspect interference some time *after* aspects have been applied and tested.

Example 6.1.1 *For example assume a new method is added to the system where now two pieces of advice from two formerly non-conflicting aspects apply. Each system modification thus always requires to check whether applied aspects interfere with each other in the new version. Doing this manually is tedious and error prone, as conflicts are not obvious.*

Finally, manually maintaining aspect precedence among a large number of aspects can be hard, as a multitude of potential conflicts has to be examined by the programmer. Most aspects however might not conflict at all. Thus the problem can easily be neglected as an irrelevant effort ("Aspects don't interfere."), leaving system semantics undefined in case of aspect conflicts.

We argue that *tool support* is necessary to make programmers aware of interfering aspects at compile time. In this chapter we analyze the problem of undefined aspect precedence in detail and define *advice order related aspect interference*. Further on we propose an analysis technique to automatically determine a set of potentially interfering aspects based on static analysis of a system. Although we use the AspectJ programming model—focusing on advice, pointcuts and (statically resolvable) joinpoint matching—our approach is applicable to languages based on a comparable programming model as well.

6.2 Illustrating the Problem—The `Telecom` Example

We will use the simple `Telecom` application which is part of the AspectJ distribution to illustrate the advice precedence problem in this paper. The `Telecom` application models a telecommunication administration system and keeps track of calls made by customers. The base application is extended with two aspects, `Timing` and `Billing`. The `Timing` aspect keeps track of the duration of a phone call, while the `Billing` aspect uses this information to calculate the amount of money that customers are charged. In the original example, aspect `Billing` contains an AspectJ statement to explicitly define that `Billing` has higher precedence than `Timing`: `declare precedence: Billing, Timing;` However, we will remove this statement to demonstrate interference problems.

Figure 6.1 shows the `Timing` and `Billing` aspects[2] of the `Telecom` example. Consider the definition of the pointcut named `endTiming` in `Timing` (line 07), which uses the `call` keyword of AspectJ to select joinpoints representing calls to the `drop()` method of the

[2] As the base code is not necessary to understand the example and also publicly available, we omit the base code here.

Listing 6.1: Timing and Billing aspects of the Telecom example. The Telecom-example is part of the AspectJ distribution, we only show relevant parts of the system here.

```
1  public aspect Timing {
2    private Timer Connection.timer = new Timer();
3    public Timer getTimer(Connection conn) {
4      return conn.timer;
5    }
6
7    pointcut endTiming(Connection c):
8      call(void Connection.drop()) && target(c);
9
10   after (Connection c): call(void Connection.complete())
11     && target(c) {
12     getTimer(c).start();
13   }
14   after(Connection c): endTiming(c) {
15     getTimer(c).stop();
16     c.getCaller().totalConnectTime
17       += getTimer(c).getTime();
18     c.getReceiver().totalConnectTime
19       += getTimer(c).getTime();
20   }
21   ...
22 }
23 public aspect Billing {
24   /* declare precedence: Billing, Timing; */
25   private Customer Connection.payer;
26   public Customer getPayer(Connection conn){
27     return conn.payer;
28   }
29   public abstract long Connection.callRate();
30   public long Customer.totalCharge = 0;
31   public void Customer.addCharge(long charge) {
32     totalCharge += charge;
33   }
34
35   after(Customer cust) returning (Connection conn):
36     args(cust, ..) && call(Connection+.new(..)) {
37     conn.payer = cust;
38   }
39   after(Connection conn): Timing.endTiming(conn) {
40     long time = Timing.aspectOf().getTimer(conn).getTime();
41     long rate = conn.callRate();
42     long cost = rate * time;
43     getPayer(conn).addCharge(cost);
44   }
45   ...
46 }
```

Listing 6.2: `Timer` class.

```
47 public class Timer {
48   public long startTime, stopTime;
49
50   public void start() {
51     startTime = System.currentTimeMillis();
52     stopTime = startTime;
53   }
54
55   public void stop() {
56     stopTime = System.currentTimeMillis();
57   }
58
59   public long getTime() {
60     return stopTime - startTime;
61   }
62 }
```

`Connection`-class. In the following *c* is the `Connection`-object `drop()` is called on. The `target` keyword is used to expose *c* to advice bound to this pointcut.

There are two pieces of `after`-advice defined in the `Timing` and `Billing` aspects referencing pointcut `endTiming` (lines 14 and 39). These pieces of `after`-advice are executed *immediately after* the call to `drop()` returns. As *c* is exposed by the pointcut, both pieces of advice can access *c* to perform their calculations. Beside pointcuts and advice, the `Telecom` example also uses *inter type declarations*. The declaration `Timer Connection.timer = new Timer()` (line 2) for example adds and initializes a field `timer` in the `Connection` class.

Ending a phone call is modeled by calling `hangUp()` on a `Customer`-object which finally results in a call to `drop()` on the `Connection`-object. The pointcut `endTiming` binds the `after`-advice of both the `Timing` and `Billing` aspects (lines 14 and 39 in Figure 6.1) to the joinpoint representing the `drop()` call. Observe that `Timing` saves the end time of the phone call (`getTimer(c).stop()`, line 15) finally used by `Billing` to calculate the amount of money the caller is charged (`getTimer(conn).getTime()`, line 40). The `Timer`-class is shown in Listing 6.2.

The `declare precedence` statement in the original example (line 24) guarantees that the advice defined in the `Timing` aspect is executed *before* the advice defined in the `Billing` aspect. However, if this statement is missing, the compiler is free to choose the opposite order (advice precedence according to the language specification is undefined). In this case the `Billing`-advice will always receive 0 when calling `getTime()` on the shared `Timer`-object, as the observant reader may verify. System functionality is broken; here the order in which both actions are performed is obviously relevant.

For this example dependence between the two aspects is easy to see and the necessary explicit ordering is easy to add. This is not the case in general. In large, team-developed projects non-trivially interfering aspects might be developed by different programmers, making it hard to even notice interference. The resulting errors are cumbersome to detect, as the interference may be well-hidden. The missing ordering might also fail to produce a failure if the compiler by chance chooses the "right" order. In this case even thorough testing fails to detect a potential problem. Note that this is not a weakness of testing, but rather a principal problem of changing semantics due to a different compiler run—in the first version there simply *is* no

problem a test could reveal. This however might change with the next compiler version or even with the next compilation of the system, if the new compiler chooses the "wrong" order.

Studying the example allows to derive a general pattern helping to define order-dependent advice interference in general. Observe that the `Timing` aspects calculates a value (the stop time for a phone call) which is subsequently used by the `Billing` aspect. Abstracting from this observation we can state that two pieces of advice interfere if one of them reads data the other one writes to. We will use this observation to define aspect interference more formally in the following.

6.3 Advice Interference

In the `Telecom` example introduced in Section 6.2 `Billing` uses information written by `Timing`. A "read from relation" is also well-known from transaction serialization theory for databases. In this context, two transactions T_1 and T_2 *conflict* if they both access a common data object and at least one of them modifies this object. If two transactions conflict, they have to be *serialized* to maintain database consistency.

Analysis of data flow between two pieces of advice is the first cornerstone of our analysis. Beside data flow, multiple pieces of advice at a single joinpoint can also interfere if they potentially prevent execution of subsequent advice. This is the case, e.g if advice throws an exception or `around`-advice does not call `proceed`. Analysis of control dependences thus is the second cornerstone of our analysis. We will declare advice interference based on these two properties. If two pieces of advice conflict, aspect precedence has to be explicitly declared to avoid undefined system semantics.

6.3.1 Data Flow Interference

To formulate the data flow interference criterion, we need to know which parts of the *system state* are used by a piece of advice. In this context the system state is an abstraction capturing a mapping from all variables known in the program to their values. Read and write access to the system state is captured by the *def*- and *use*-sets, where we capture all accesses in the control flow of a given piece of advice.

Definition 6.3.1 (*def*() and *use*()-sets) *Let 'm' be a method or a piece of advice. Let 'decl' be the unique source location of a variable declaration. Let N_t be the set of call targets for all call sites in 'm' and A_t the set of all pieces of advice adapting a joinpoint in m's lexical scope. Then def() and use() are defined as follows:*

$$\begin{aligned} def(m) = {} & \{(a, decl(a)) \mid a \text{ appears as l-value in } m\} \\ & \cup \{(a.x, decl(a)) \mid a.x \text{ appears as l-value in } m\} \\ & \cup \bigcup_{n_t \in N_t} def(n_t) \cup \bigcup_{a_t \in A_t} def(a_t) \end{aligned}$$

$$\begin{aligned} use(m) = {} & \{(a, decl(a)) \mid a \text{ appears as r-value in } m\} \\ & \cup \{(a.x, decl(a)) \mid a.x \text{ appears as r-value in } m\} \\ & \cup \bigcup_{n_t \in N_t} use(n_t) \cup \bigcup_{a_t \in A_t} use(a_t) \end{aligned}$$

Traditionally the *def*- and *use*-sets of a method (and similarly advice) are defined as the set of memory locations defined (or written) and used (or read) by a method, respectively. We define *def*() and *use*()-sets semi-formally based on the statements contained in a method or advice. Note that we assume that nested statements have been resolved previously for simplification,

which is always possible using simple syntactical transformations and is done on-the-fly by our analysis.[3]

Definition 6.3.1 states that all identifiers (both qualified and unqualified, i.e. including the package (and type) name or not) used as l-values are added to the *def*()-set, while all identifiers used in expressions or method calls as parameters are added to the *use*()-set (as they are implicitly used as r-values). Note that we add the *def*()- and *use*()-sets of called methods and attached advice as well, as we have to analyze data access in the complete control flow of potentially conflicting advice, including subsequently called methods. Note further that we associate each identifier with its unique declaration site to be able to unambiguously identify it even if called methods or nested blocks reuse names.

The recursion in Definition 6.3.1 terminates for methods where either no call site or attached advice are present or all called methods/pieces of advice have been previously analyzed. Recursive methods are handled using fixpoint iteration.

Example 6.3.1 (*def*() and *use*()-sets) *Consider the following code:*

```
1  class C {
2    private int x;
3    public static void main(String[] args) {
4      C c = new C();
5      c.x = 42;
6      int y = -1;
7      c.foo(y);
8    }
9    void foo(int z) {
10     int q = z * x;
11     x = q;
12   }
13 }
```

The system state consists of a mapping for the variables `c`*,* `c.x`*,* `y`*,* `z`*,* `q` *and* $\texttt{this}_{foo}\texttt{.x}$*. Then* $def(\texttt{foo}) = \{(\texttt{q, 10}), (\texttt{this}_{foo}\texttt{.x, 9})\}$ *and* $use(\texttt{foo}) = \{(\texttt{z, 9}), (\texttt{this}_{foo}\texttt{.x, 9}), (\texttt{q, 10})\}$ *and* $def(\texttt{main}) = \{(\texttt{c, 4}), (\texttt{c.x, 4}), (\texttt{y, 6})\} \cup def(\texttt{foo})$ *and* $use(\texttt{main}) = \{(\texttt{c, 4}), (\texttt{y, 6})\} \cup use(\texttt{foo})$.

Note that implicit assignments to formal parameters are missing here, as we do not need them for the conflict analysis. For each variable, we gave the line number of its definition as its declaration handle. Remember that for a field access o.x, we add the declaration handle of the qualifier o. For `this` *and formal parameter variables, we state the line number of the defining method as the respective declaration handle in this example. In general the declaration handle is more complex and also comprises file name and column.*

Analyzing all pieces of advice is not necessary, as only a small set of available pieces of advice is relevant. We define advice data dependence for relevant advice in the following.

Definition 6.3.2 (Relevant Advice) *Two pieces of advice* a_1 *and* a_2 *are* relevant*, if they are defined in different aspects, apply at at least one common joinpoint and are either of the same kind or at least one of them is* `around`*-advice.*

If the above criteria are not met, then application order for two pieces of advice is defined, either by program control flow (different joinpoints) or the language semantics (advice of same aspect, non-conflicting kind).

[3] For example we transform $x = a.b.c$ to $\$1 = a.b; x = \$1.c$ or $f(g(x))$ to $\$1 = g(x); f(\$1)$ where \$1 is a new auxiliary variable.

Example 6.3.2 (Relevant Advice) *Assume we have two (not explicitly ordered) aspects A_1 and A_2. A_1 defines* `before`*-advice bf_{1_1} and* `before`*-advice bf_{1_2}, advice A_2* `after`*-advice af_{2_1} and* `around`*-advice ar_{2_2}. Further assume we have 3 joinpoints, jp_1, jp_2 and jp_3 matched be the 4 pieces of advice as follows: $bf_{1_1} \rightarrow \{jp_1, jp_3\}$, $bf_{1_2} \rightarrow \{jp_1, jp_2\}$, $af_{2_1} \rightarrow \{jp_2\}$ and $ar_{2_2} \rightarrow \{jp_3\}$.*

Then each joinpoint is affected by two pieces of advice, however for jp_1 both adapting pieces of advice are defined in aspect A_1, thus their order is defined and no conflict occurs. For joinpoints jp_2 advice from different aspects applies, however both pieces of advice have different kind (`before` *and* `after`*), thus this joinpoint is not relevant for analysis either. For joinpoint jp_3 however, the two applying pieces of advice bf_{1_1} and ar_{2_2} are* relevant *for analysis, as they are defined in different unordered aspects (A_1 and A_2), and ar_{2_2} is* `around`*-advice.*

For relevant pieces of advice we define data dependence as follows.

Definition 6.3.3 (Advice Data Dependence) *Let a_1 and a_2 be two relevant pieces of advice. Let '$objects(r, decl(r))$' denote the objects a reference 'r' may refer to, '$formals(a)$' be the formal parameters of a piece of advice 'a', and '$actuals($*`proceed`*$)$' the actual parameters of the call to* `proceed`*. Then a_1 is* data dependent *on a_2 if*

$$(r_1.x, decl(r_1)) \in def(a_2) \wedge (r_2.x, decl(r_2)) \in (def(a_1) \cup use(a_1))) \\ \Rightarrow objects(r_1, decl(r_1)) \cap objects(r_2, decl(r_2)) \neq \emptyset$$

or, if a_2 is `around`*-advice,*

$$formals(a_2) \not\equiv actuals(\texttt{proceed}) \vee formals(a_2) \cap def(a_2) \neq \emptyset \vee \\ a_2 \text{ returns a different value than } \texttt{proceed}(\ldots).$$

The first property checks if one advice reads data from the other, similarly to the criterion stated for transactions. The second criterion however handles the special case of `around`-advice. Such advice can easily modify or completely redefine the actual parameters of the `proceed`-call, thus changing values reaching e.g. a called method. Similarly it can also access and modify the return value. The second property explicitly captures these cases. If a `proceed`-parameter is a reference variable, (transitive) modifications of accessible fields or method calls on the underlying object are captured by the first criterion.

Example 6.3.3 (Advice Data Dependence) *We expand the code shown in Example 6.3.1 with two pieces of advice:*

```
14 aspect A1 {
15   void around(C c, int y):
16     call(void C.foo(int)) && args(y) && this(c) {
17
18     proceed(c.x * y); // bound to C.foo(int)
19   }
20 }
21 aspect A2 {
22   before(C c):
23     call(void C.foo(int)) && this(c) {
24     c.x = c.x * 2;
25   }
26 }
```

We check the criterion of Definition 6.3.3 on this example. As the reader may verify, the two pieces of advice are relevant. For the `before`*-advice we establish* $def(\texttt{before}_{\texttt{A2}}) = \{$`(c.x, 28)`$\}$*, for the* `around`*-advice use*$(\texttt{around}_{\texttt{A1}}) = \{$`(y, 23), (c.x, 21)`$\}$.

Running a pointer analysis on this code now reveals that $objects((\texttt{c},21)) \cap objects((\texttt{c},28)) = (\texttt{new C}(),4)$*, i.e. the* `before`*-advice actually changes a value (field* `x`*) read by the* `around`*-advice. The criterion uncovers this data dependence.*

Note that the second part of the criterion—handling `proceed`*—also shows a problem as* `proceed` *changes the (accessible) call parameter before forwarding it (i.e.* $formals(\texttt{around}_{\texttt{A1}}) \not\equiv actuals(\texttt{proceed})$*), thus also revealing a data dependence.*

6.3.2 Control Flow Interference

Additionally to data flow between advice we also have to examine the *control flow*. Advice can modify control flow such that execution of pieces of advice with lower precedence applying at the same joinpoint is prevented (e.g. by throwing an exception). In this case program semantics again depend on advice precedence; their order thus has to be explicitly stated. We define control dependence and finally advice interference as follows.

Definition 6.3.4 (Advice Control Dependence) *Let a_1 and a_2 be two relevant pieces of advice. a_1 is* control dependent *on a_2, if a_2 explicitly throws an exception*[4] *which is not handled in the advice body of a_2 or if a_2 is* `around`*-advice and either an exception thrown by* `proceed` *is caught or* `proceed` *is not called* exactly once *in its control flow.*

Note that demanding that `proceed` is called *at least once* is not sufficient as multiple calls to `proceed` subsequently result in multiple executions of respective methods and advice so likely changing system semantics.

Definition 6.3.5 (Advice Interference) *Two relevant pieces of advice a_1 and a_2 interfere, if a_1 is* data or control dependent *on a_2 or vice versa.*

Note that advice interference is restricted by the advice kind. The order of `before` and `after` advice is trivially determined. However, if one of the two pieces of advice is `around`-advice or both pieces of advice are of the same kind, then advice precedence may be undefined and in this case can be picked arbitrarily by the compiler, potentially affecting program semantics. Finally we define aspect interference based on conflicting advice.

Definition 6.3.6 (Aspect Conflict) *Let A_1 and A_2 be two aspects. Then A_1 and A_2* conflict*, if two pieces of advice $a_1 \in A_1, a_2 \in A_2$ exist such that a_1 and a_2 interfere and precedence of A_1 and A_2 is undefined.*

Example 6.3.4 (Telecom) *We apply Definition 6.3.6 to the* `Telecom` *example from Section 6.2. Let a_1 be the* `after`*-advice in* `Timing`*, a_2 the* `after`*-advice in* `Billing`*. As we removed the* `declare precedence` *statement, precedence of these two aspects is undeclared. These two pieces of advice are* relevant *as they are both bound to joinpoints selected by pointcut* `endTiming`*, are of the same kind (both* `after`*-advice) and are defined in different aspects,* `Timing` *and* `Billing`*, respectively.*

As the reader may verify, both a_1 and a_2 access the same `Timer`*-object o_{tim} associated with the current connection through their respective call to* `getTimer(Connection)`*. a_1 calls o_{tim}.*`stop()`*, thus setting o_{tim}.*`stopTime`*, i.e. o_{tim}.*`stopTime` $\in def(a_1)$*. a_2 in turn calls o_{tim}.*`getTime()`*, thus reading o_{tim}.*`stopTime`*, i.e. o_{tim}.*`stopTime` $\in use(a_2)$*. So there is a data dependence between a_1 and a_2 on o_{tim}.*`stopTime`*. As a consequence a_1 and a_2 interfere and our criterion discovers that aspects* `Timing` *and* `Billing` *have to be explicitly ordered, as we saw in the original version of the* `Telecom` *example.*

[4] Note that we consider explicitly thrown exceptions only. `RuntimeException`s due to programming errors (e.g. `NullPointerException`s) are ignored in this context.

Note that this criterion, if it succeeds, does not state that the two aspects are independent of each other in general. There might of course be data flow between them. However such data flow does not depend on advice precedence as long as both pieces of advice do not apply at the same joinpoint. Joinpoints are reached subsequently as program execution proceeds,[5] so advice execution order is determined by program control flow. If aspect precedence however is relevant for program semantics, Definition 6.3.6 provides a sufficient criterion to check if an order has to be established for two given pieces of advice.

Note further that although detecting interference among pieces of advice can considerably help to avoid problems, this will not prevent programmers to add two *semantically incompatible aspects* to a system. Consider for example a tracing and an encryption aspect (taken from [20]). The tracing aspect is logging relevant data to be able to easily understand system failures, while the encryption aspect's job is to assert that no data leaves the system unencrypted. Obviously there is a conflict here—we end up with an unencrypted log if logging has a higher precedence than encryption (thus breaking the encryption aspect) or with an encrypted log (hampering logging). Both solutions are not satisfactory. However, if both aspects access common joinpoints, our analysis will at least detect the conflict (as the same data is accessed by both aspects and the encrypting aspects modifies it). If no common joinpoints exist, the system will at least always show the same behavior and not depend on the compiler used, easing debugging in this case.

6.3.3 Criterion is Sufficient

We finally prove that our interference criterion is sufficient, given that undeclared runtime exceptions can be ignored.

Proposition 6.3.1 *Let A_1 and A_2 be two aspects. If these aspects do not conflict according to Definition 6.3.6, then the execution order of contained advice is either determined or all execution orders are equivalent.*

Proof: Proof by contradiction. We assume our criterion holds, but the compiler has freedom in advice execution order and different orders produce different results.

Case 1: Aspect precedence for A_1 and A_2 is determined. In this case language semantics determine advice order as well, in contradiction to the assumption.

Case 2: Aspect precedence is undefined, but advice order is defined nevertheless. Let $a_1 \in A_1$ and $a_2 \in A_2$ be two pieces of advice applying at the same joinpoint *jp* (otherwise execution order is determined by program control flow). Their order is defined if either a_1 or a_2 is `after`-advice and the other advice is `before`-advice. All these cases are in contradiction to the assumption, as the compiler cannot choose execution order.

Case 3: Assume now A_1 and A_2 contain two pieces of advice a_1 and a_2, where the compiler may choose execution order. Execution of advice changes the system state S creating a new state S', denoted by $S' = exec(a_1, S)$. If two pieces of advice a_1 and a_2 are commutative, then $S'' = exec(a_1, exec(a_2, S))$ $= exec(a_2, exec(a_1, S))$ holds. We now show that our criterion guarantees that a_1 and a_2 commute.

We first discuss possible changes of advice context by `around`-advice. Modifications of `proceed`-parameters and return value could result in different values passed to subsequently executed advice. This however contradicts property (2) in Definition 6.3.3.

As direct changes of exposed context are not possible, indirect modifications of reachable system state have to be considered. Property (1) from Definition 6.3.3 guarantees that two pieces of advice do not read data from each other, the parts of the system state defined by one advice is disjoint with the part of the system state accessed by the other.

Data flow could also occur transitively via a third piece of advice a. In this case however, both a_1 and a_2 would conflict with this piece of advice, finally resulting in a not necessarily total but sufficient order of all three pieces of advice. If these conflicts are eliminated, the following orders are possible:

[5] Note that we do not explicitly handle multi-threaded programs here. Synchronization in this case is similar to synchronization of traditional Java code.

$a_1 \sim a_2 < a$; $a_1 < a < a_2$; $a_2 < a < a_1$ and $a < a_1 \sim a_2$, where $\sim$ denotes an undefined order. For any of these orders however data flow is determined as the reader may verify. Except advice, no other statements can be executed in-between applying pieces of advice, i.e. no other indirect data flow can occur.

Property (2) in Definition 6.3.5 guarantees that both pieces of advice are actually executed, as neither a_1 nor a_2 may throw an exception or skip a call to `proceed` preventing execution of subsequent advice.

This property also guarantees that `proceed` is executed exactly once, as multiple calls to `proceed` might result in multiple executions of advice with lower precedence thus also potentially changing the resulting system state.

To summarize, both pieces of advice get the same input, and produce the same output as they are both executed exactly once and do not modify common parts of the system state if our criterion holds, i.e. a_1 and a_2 commute. This is again in contradiction to the assumption as system semantics do not depend on advice execution order. Thus the assumption is false, A_1 and A_2 do not conflict. □

6.4 Checking For Aspect Conflicts

In this section we discuss how our criterion has been implemented. Unfortunately not all necessary information can be calculated statically, thus we had to both approximate unknown information and use heuristics.

6.4.1 Finding Relevant Advice

The first step in our analysis is to find relevant pieces of advice, as here potentially conflicts can occur. Joinpoints where multiple pieces of advice apply can be derived from the matching information calculated by the AspectJ weaver. To create the executable system, the weaver evaluates pointcut definitions and uses this information to map advice to adapted joinpoints. We use this mapping for our analysis. Our criterion assumes that we can evaluate advice-joinpoint matching statically. However, as AspectJ also knows dynamic pointcut constructs (`if`, `cflow`, `this`) we have to deal with this uncertain information. Fortunately, the weaver also reports whether a pointcut contains dynamic constructs, i.e. if the matching is only a conservative approximation or not. So we can use this information to create conservative approximations if necessary.

Not all joinpoints where multiple pieces of advice apply are relevant for our analysis. The system might already contain rules to explicitly define advice precedence. For AspectJ, two pieces of advice are ordered, if (1) they are defined in the same aspect, (2) their defining aspects are related by inheritance or (3) a `declare precedence` statement lists aspects (with decreasing precedence). Deriving information about inheritance relations and `declare precedence` statements from the system is straightforward.

To check our interference criterion we then examine control dependences and data flow for each pair of advice defined in potentially conflicting (i.e. unordered) aspects applying at the same joinpoint, if the advice kind is relevant for a conflict.

6.4.2 Basic Data Structures

To implement our analysis we need two basic data structures: the intra-procedural control flow graphs and the advice-aware call graph.

Advice-Aware Call Graph Construction

Call graphs describe the calling relations among methods–and in this context also advice–for a program. Each method and piece of advice is modeled as a node, and an edge from a node n_1 to a node n_2 indicates that a call site or joinpoint, respectively, in n_1 (potentially)

invokes n_2. Creating a call graph for a language like AspectJ raises two important issues. First, dynamic binding has to be approximated and second implicit "advice calls" have to be added in the call graph to model advice execution at adapted joinpoints. While the first problem is a standard problem for object-oriented languages [25], modeling of advice application is more interesting. Here we insert explicit call edges from the method containing an adapted

```
class C {
  void zip() { foo(); }
  void foo() {}
}
aspect A1 {
  around(): call(* foo()) {
    proceed();
  }
}
aspect A2 {
  around(): call(* foo()) {
    proceed();
  }
}
```

(a)

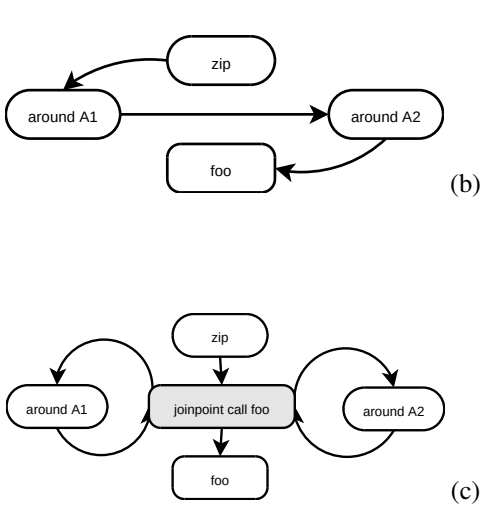


Figure 6.1: **Call Graph Modeling for Aspect-Oriented Programs** with `around`-advice. Figure **(a)** shows the source code, figure **(b)** the resulting call graph if we assume that advice order is determined. Figure **(c)** shows a conservative approximation of call graph modeling if advice precedence is unknown.

joinpoint to attached pieces of advice. While this approach is straightforward for `before` and `after`-advice, `around`-advice is more challenging, as it actually *wraps* the adapted joinpoint, which is only reached if `proceed` is called. Several pieces of `around`-advice thus result in a hierarchy of wrappers, their order implied by advice precedence.

Creating this wrapper hierarchy however faces two problems. First, advice *precedence can be undefined* and second advice application itself can be uncertain due to *dynamic pointcuts*, i.e. pointcut definitions which depend on runtime values and thus cannot be decided statically. To deal with both problems we explicitly model the respective adapted joinpoint and add edges from the containing method to its joinpoint. We assume that each advice is called from the adapted joinpoint in turn, thus flattening any wrapper hierarchy to avoid a combinatorial explosion[6] which would result if all possible precedence orders had to be considered.

Example 6.4.1 (Call Graph Modeling) *Consider the simple example program shown in Figure 6.1. Sub-figure (a) shows the source code of a simple example involving two pieces of* `around`*-advice. Sub-figure (b) shows the call graph if we know that* `A1` *has higher precedence than* `A2`*. As we apply* `around`*-advice, the original joinpoint—although not modeled explicitly in this call graph—is replaced by the "call" to the* `around` *advice in* `A1`*. The* `around`*-advice in* `A2` *is called next, as it has lower precedence than the previous advice. We thus add a respective edge to the call graph. After unfolding the wrapper hierarchy we finally add the original call target* `foo()`*.*

However, if either aspect application is uncertain due to dynamic joinpoints or advice order is unknown, this modeling of the call graph is incorrect, as paths which actually occur are potentially lost (just assume the opposite precedence order as shown in Figure 6.1 (b)). To

[6]For a detailed discussion of the possible combinatorial explosion refer to Section 6.4.4.

deal with this problem, we explicitly model the joinpoint as a special node in the call graph in these cases, as shown in Figure 6.1 (c). Advice applied at this joinpoint is now attached to this joinpoint node, and a call to `proceed` *in the* `around`*-advice links back to this joinpoint instead of an advice or method node. The original call target* `foo()` *is also attached to the new joinpoint node.*

With this construction, all feasible paths are represented in the call graph, no paths are lost, however at the cost that now infeasible paths can occur. As this construction is only necessary if precedence order is undeclared or dynamic pointcuts exist, the additional imprecision can be tolerated. We will define call graph construction more formally in the following.

Definition 6.4.1 (Notation) *Let $\mathscr{M}$ be the set of all methods, and $\mathscr{A}$ the set of all pieces of advice defined in a given program. Let JP be a set of joinpoints. Let $A = \{a_1, ...a_n\} \subseteq \mathscr{A}$ be a set of pieces of advice. Let*

$$match : JP \times \mathbb{P}(\mathscr{A}) \rightarrow \mathbb{P}(\mathscr{A})$$

be a function selecting all pieces of advice in $\mathscr{A}$ adapting the joinpoint $jp \in JP$. Let

$$\sigma : JP \times \mathbb{P}(\mathscr{A}) \rightarrow \mathbb{P}(\mathscr{A})$$

be a function selecting all pieces of advice in $match(jp,A)$ statically *matching the respective joinpoint from JP. Let*

$$\pi(jp, \{a_1, \ldots, a_n\}) = \begin{cases} \bot, \textit{if advice precedence is undeclared} \\ [a_{k_1}, \ldots, a_{k_n}], k_i \in \{1, \ldots, n\}, i \neq j \Rightarrow k_i \neq k_j, \textit{else} \end{cases}$$

be a function returning a list of advice ordered by advice precedence for a given set of advice applying at a joinpoint jp. If the corresponding joinpoint is apparent from the context we will drop this parameter for σ and π. Finally, if $jp \in JP$ represents a `call`*-joinpoint,*

$$targets : JP \rightarrow \mathbb{P}(\mathscr{M})$$

is a function to derive the set of all potentially invoked methods at this call site (for non-`call` *joinpoints $targets(jp) = \emptyset$).*

In the above definitions, $\mathbb{P}$ denotes the *power set* for the given argument. With the functions σ and π we have the necessary means to express if we have to explicitly model the joinpoint node in the call graph or not. For all pieces of advice not selected by σ we cannot say statically if this piece of advice actually applies at runtime. If $\pi(jp,A) = \bot$ for a given set of advice A applying at a joinpoint jp, then precedence order is undefined. In each case we explicitly model the joinpoint as a separate node, if `around`-advice is involved. The function *targets* finally serves to express all potential call targets for (virtual) call sites.

Definition 6.4.2 (Advice-Aware Call Graph) *The advice-aware call graph is defined as a graph $G = (V,E)$ where V contains all methods $\mathscr{M}$ and pieces of advice $\mathscr{A}$ defined in the system, as well as additional joinpoint nodes (i.e. $V \supseteq \mathscr{M} \cup \mathscr{A}$, as we may add additionally joinpoint nodes) and $E \subseteq V \times V$ contains an edge (m,n) if m potentially calls (explicitly for methods or implicitly for advice) n.*

We first define a helper function to state how an ordered list of advice (including `around`-advice) is processed at a given joinpoint, if the set of applying advice is statically known and advice order is defined. We will use the term *executable* in order to address methods and advice in the following, if a distinction is not important.

Definition 6.4.3 (Wrapping Hierarchy) *Let $e \in V$ be an executable containing joinpoint jp adapted by a set of advice $A = \{a_1, ..a_n\}$. Let $\pi(\sigma(A)) = [a_{k_1}, ..a_{k_n}]$. Then $(E, e') =$* `createWrappers`*$(jp,$*`true`*$, [a_{k_1}, ..a_{k_n}], e, E)$, where*

```
createWrappers(jp, addCall, [], current, E) =
    case addCall then (E ∪ {(current,t)|t ∈ targets(jp)}, current)
    otherwise (E, current)
createWrappers(jp, addCall, a:as, current, E) =
   (let E' = E ∪ (current, a)
    and current' =
       case (kind(a) == around) then a
       otherwise current
    and (addCall', as') =
       case (proceed ∈ a) then (true, as)
       otherwise (false, [])
    in createWrappers(jp, addCall', as', current', E'))
```

The function `createWrappers` intuitively takes a list of advice applying at a single joinpoint which is ordered by decreasing precedence, and creates edges accordingly, thereby respecting the wrapping hierarchy resulting from `around`-advice. Therefore, we have to model that advice with lower precedence is only executed if the `around`-advice actually calls `proceed`. This can be interpreted such that advice is moved from the original joinpoint to the `proceed` statement. We model this changing of the joinpoint by changing the `current` node in `createWrappers` in case of `around`-advice. The parameter `addCall` serves to model the case that `around`-advice does not contain a `proceed`-statement. In this case original calls are also suppressed, which is modeled by the case distinction in the base case. The function `createWrappers` finally returns the resulting edge set and the innermost node of the wrapping hierarchy.

What remains is the case that the applying advice set cannot be determined statically and/or advice precedence is undefined. To formulate this case, we need a helper function `back`(jp, A, $\tilde{E}$), to model edges from advice back to a joinpoint node in case a wrapper hierarchy cannot be constructed.

Definition 6.4.4 (Back Edges) *For a given joinpoint jp and a given set of advice A,* `back`*(jp, A,* $\tilde{E}$*) =* $\tilde{E}'$ *calculates the set of necessary back-edges as follows:*

```
back(jp, ∅, Ẽ) = Ẽ
back(jp, A, Ẽ) =
   (let a ∈ A and A' = A − {a}
    in case proceed ∈ a then back(jp, A', Ẽ ∪ {(a,jp)})
       otherwise back(jp, A', Ẽ))
```

We finally define function `createEdges` to model application of a given set of advice A at a joinpoint jp, independent of defined advice precedence.

Definition 6.4.5 (Create Edges) *Let* $e \in V$ *be an executable containing joinpoint jp adapted by a set of advice* $A = \{a_1, ..a_n\}$*. Then we calculate* $(V,E,e') =$ `createEdges`(jp,A,V,E,e)*, where*

```
createEdges(jp, A, V, E, current) =
   case (σ(jp, A) = A ∧ π(jp,A) ≠ ⊥) then
      (V, createWrappers(jp, true, π(jp,A), current, E))
   otherwise
      (let V' = V ∪ {jp}
       and E' = E ∪ {(current, jp)}
           ∪ {(jp,t)|t ∈ targets(jp)} ∪ {(jp,a)|a ∈ A} ∪ back(jp,A,∅)
       in (V', E', jp))
```

Function `createEdges` generalizes function `createWrappers` by also defining how a joinpoint is processed if either the set of actually applying advice is statically unknown or advice precedence is undefined. In this case, an explicit node is added to the call graph modeling the current joinpoint—as outlined in the introductory example—and all pieces of advice as well as call targets (if any) are connected to this new node. Finally, if we deal with `around`-advice and a `proceed`-statement is contained, we also add back-edges from the advice to the new joinpoint node. If creation of a wrapper hierarchy is possible, we simply call `createWrappers`.

With function `createEdges` at hand, we can now define call graph construction for a given executable e.

Definition 6.4.6 (Call Graph Construction) *To construct G, we process each $e \in V$ in turn. Let $joinpoints_e$ be a list of adapted joinpoints and call sites in the lexical scope of e, ordered by their **lexical appearance**. To process e and calculate successor nodes and respective edges, we call $processExecutable(joinpoints_e, V, E, e) = (V', E')$, where*

```
1 processExecutable([], V, E, current) = (V, E)
2 processExecutable(j:js, V, E, current =
3    (let as = match(j, 𝒜)
4     and (V', E', e') = createEdges(j, as, V, E, current)
5     in (case j is execution-joinpoint then
6           processExecutable(js, V', E', e')
7         otherwise
8           processExecutable(js, V', E', current)))
```

The function *processExecutable* defines how all points of interest for call graph construction—i.e. call sites and joinpoints with attached advice—in an executable e are processed. Intuitively *processExecutable* calls `createEdges` on each call site/adapted joinpoint, thus capturing all potentially invoked executables.

The case distinction is necessary, as there is the special case of `execution`-joinpoints. Here, attached `around`-advice not only wraps a single statement, but the complete body of e. To model this case, we connect any executable invoked within e to the innermost node in the wrapper hierarchy resulting for the execution joinpoint, if applicable. Note that due to the joinpoint ordering the single `execution` joinpoint is always handled before all other joinpoints contained in a method.

Example 6.4.2 *As an example we construct the call graph for the code shown in Example 6.1. Here, V contains methods `zip()` and `foo()` as well as the two pieces of `around`-advice which we will label a_{A1} and a_{A2}, where the index denotes the defining aspect (i.e. $\mathscr{A} = \{a_{A1}, a_{A2}\}$). We start our analysis with the `zip()` method. This method only contains a single adapted joinpoint—the call to `foo()` which we label jp. We thus call $processExecutable([jp], V, E, \texttt{zip}())$. To calculate the result, we establish $match(jp, \mathscr{A}) = \mathscr{A}$ (all pieces of advice in the system match this joinpoint in this example). The joinpoint jp is a `call`-joinpoint (and* no `execution`*-joinpoint), thus $target(jp) = \{\texttt{foo}()\}$ is relevant but we don't have to process the special case of execution advice. So next, we calculate $createEdges(jp, \mathscr{A}, V, E, \texttt{zip}())$.*

Case 1: aspect order undefined. *We first assume aspect order is undefined. As a consequence $\pi(jp, \mathscr{A}) = \bot$. In this case the "otherwise" case in Definition 6.4.5 is relevant. Thus $V' = V \cup \{jp\}$ and $E' = E \cup \{(\texttt{zip}, jp)\} \cup \{(jp, \texttt{foo}())\} \cup \{(jp, a_{A1}), (jp, a_{A2})\} \cup \{(a_{A1}, jp), (a_{A2}, jp)\}$ ($back(jp, \mathscr{A}, \emptyset) = \{(a_{A1}, jp), (a_{A2}, jp)\}$, as both pieces of advice are `around`-advice and contain `proceed`-statements), i.e. $V = \{\texttt{zip}(), \texttt{foo}(), a_{A1}, a_{A2}, jp\}$ and $E = \{(\texttt{zip}, jp), (jp, \texttt{foo}()), (jp, a_{A1}), (jp, a_{A2}), (a_{A1}, jp), (a_{A2}, jp)\}$.*

Case 2: $A1$ **has higher precedence than** $A2$. *Now* $\pi(jp, \mathscr{A}) = [a_{A1}, a_{A2}]$*, i.e. createEdges forwards calculation of edges to function createWrappers shown in Definition 6.4.3. There we have the following calculation:*

$$\begin{array}{l} createWrappers(jp, \texttt{true}, [a_{A1}, a_{A2}], \texttt{zip}(), \{\}) = \\ createWrappers(jp, \texttt{true}, [a_{A2}], a_{A1}, \{(\texttt{zip}(), a_{A1})\}) = \\ createWrappers(jp, \texttt{true}, [], a_{A2}, \{(a_{A1}, a_{A2}), (\texttt{zip}(), a_{A1})\}) = \\ (\{(a_{A1}, a_{A2}), (\texttt{zip}(), a_{A1})\} \cup \{(a_{A2}, \texttt{foo}())\}, a_{A2}), \end{array}$$

i.e. $V = \{\texttt{zip}(), \texttt{foo}(), a_{A1}, a_{A2}\}$ *and* $E = \{(a_{A1}, a_{A2}), (\texttt{zip}(), a_{A1}), (a_{A2}, \texttt{foo}())\}$.

As with this calculation the current joinpoint is processed, processExecutable would now continue with the next point of interest. However, in our case the list is now empty, and we return the current node and edge sets. As neither `zip()` *nor* `foo()` *contain any additional adapted joinpoints or call sites, as well as the two pieces of advice (note that the special form* `proceed` *is implicitly processed when creating the wrapper hierarchy), call graph construction then terminates in both cases. As the reader may verify the now created call graphs match the call graphs shown in Figures 6.1 (b) and (c), respectively.*

Exception-Aware Control Flow Graphs

For each method and each piece of advice we create the control flow graph. In this data structure each statement is represented by a node, and an edge between two nodes indicates potential control flow from one node to the other. This is a standard data structure for static program analysis; creating control flow graphs for advice does not add any additional issues.

However, our analysis also needs information about exceptional control flow. Therefore we augment the control flow graph with this information. While handling of `catch` and `throw` statements is straightforward, method calls (and advice application as well) are more complex as here potentially exceptions thrown in the control flow of the called method (or applied advice, respectively) can be propagated as well. Therefore we have to calculate the set of exceptions potentially propagated by each method and piece of advice.

Definition 6.4.7 (Propagated Exceptions) *Let* N_t *and* A_t *be defined as in Definition 6.3.1, e be an executable, and x an exception. Then the set of exceptions potentially propagated by e* $prop(e)$ *is defined as follows:*

$$\begin{array}{rl} prop(e) = & \{x | \texttt{throw}\ x \in e\} \\ & \cup \bigcup_{n_t \in N_t} prop(n_t) \cup \bigcup_{a_t \in A_t} prop(a_t) \\ & - \{x | x \text{ is handled in } e\} \end{array}$$

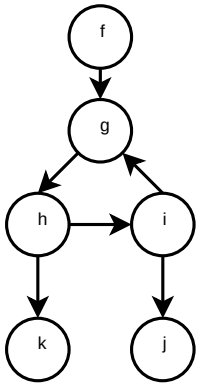
To calculate this information we process the control flow graphs for each method in the call graph in topological order and calculate the set of *propagated exceptions* for each method. For library methods we rely on their `throws`-declarations. Methods not calling any other methods or only library methods are thus the base case of the above recursion. The resulting propagation set is then inserted at each call site to an already processed method.

To deal with cycles (due to recursion) we use a standard technique. First we calculate the set of propagated exceptions for all non-cycle methods which are called by methods within the cycle. Second, we propagate exception sets around the cycle until a fixpoint is reached. These final sets are then propagated to all methods calling cycle members. Once this analysis is finished we have an exception aware control flow graph, i.e. for each method and piece of advice we know the set of exceptions thrown by them.

Example 6.4.3 *As the* `Telecom`*-example does not contain any recursive methods we use an artificial example to illustrate this approach. Consider the program shown in Figure 6.2.*

```
1  class C {
2    void f() { g(); }
3    void g() {
4      try { h(); }
5      catch (E1 e) {}
6    }
7    void h() { i(); k(); }
8    void i() { j(); g(); }
9    void j() {
10     throw new E1(); }
11   void k() {
12     throw new E2(); }
13 }
```

(a)



(b)

Figure 6.2: Calculating the set of Propagated Exceptions. Figure (a) shows the source code, figure (b) the corresponding call graph for a simple example. Note that methods {g, h, i} recursively call each other, i.e. the call graph is cyclic.

A valid topological order for the call graph shown in (b) is $\{j,k,\{g,h,i\},f\}$*, where nodes* $\{g,h,i\}$ *form a cycle and thus are treated as a single node for the topological order. We start our analysis with methods j and k, which each throw an exception,* `E1` *and* `E2`*, respectively. As both methods have no call sites and do not handle their exceptions,* $prop(j) = \{E1\}$ *and* $prop(k) = \{E2\}$.

This information is now propagated to i and h, respectively. However as both methods are part of the cycle, we have to start a fixpoint iteration. We start with $prop(i) = \{E1\}$*, due to the call to j. For* $prop(h) = \{E1, E2\}$ *due to the calls to i and k. Next, we establish* $prop(g) = \{E1, E2\} - \{E1\} = \{E2\}$*, due to the call to h and the handler for* `E1`*. Further propagating this information in the cycle does not further change the propagation sets, thus the fixpoint is reached. So we can propagate the final set for g to establish* $prop(f) = \{E2\}$.

Note that the calculated propagation sets do not represent a conservative solution (our analysis misses undeclared `RuntimeException` thrown by libraries as well as exceptions like `NullPointerException`s potentially thrown by the virtual machine). We believe that this approach has three important advantages:

1. Analysis based on this simpler heuristic approach is considerably faster compared to more precise approaches, which is an important property as we envision use of our analysis during compilation.

2. If we follow the Java convention that `RuntimeException`s are programming errors then these exceptions should not be caught but fixed and program semantics (as it should be!) consequently do not depend on advice precedence (crash for any advice order).

3. We reduce the amount of false positives as each non-trivial piece of advice can potentially throw some `RuntimeException`.

Robin Sterr implemented the construction of the control flow graphs and the exception analysis as an extension of the AOPA framework for his bachelor project [23]. The implementation of the control flow analysis presented here builds on these data structures.

6.4.3 Checking Control Dependences

We use the control flow graphs to examine advice control dependence. Note that the way how `before` and `after` advice can affect control flow in AspectJ is rather limited. The only way to do this is by throwing an exception. `Around`-advice however has to explicitly call `proceed`; otherwise the original joinpoint—and lower precedence advice—is not executed. We use the exception-aware control flow graphs to check both properties.

Analyzing `proceed`

To analyze if `around`-advice indeed calls `proceed`, we check if *exactly one* call to `proceed` is *on every path* in the *exception-aware control flow graph* through the advice using a modified depth first search, which counts the number of `proceed`-nodes visited for each path. If the depth first search reaches an exit statement, the `proceed`-count has to be exactly one in each case. If an already visited node is hit again, the counter must never exceed 1. If this is not the case we report a potential control flow dependence.

Note that due to the heuristic exception analysis this is only a heuristic as well, as some exceptional paths are missing in the control flow graph. However we believe that creating a safe analysis producing too many false positives is less valuable than an analysis missing some cases but in general reporting actual problems, especially if missed problems are closely related to programming errors (`RuntimeException`s) which should be detected by unit tests and corrected afterward, instead of handling these exceptions in the program.

Analyzing Exceptions

Analysis of exceptional behavior of advice is straightforward once the propagation sets have been calculated. For a piece of `before` or `after`-advice a we can check our criterion by checking that $prop(a) = \emptyset$.

For `around`-advice however the advice must not change the exception throwing behavior of the call to `proceed`. Thus we have to check that no exception potentially thrown by `proceed` is handled by the `around`-advice and that the advice code throws no additional exceptions. However this information is captured in the exception aware control flow graphs and thus easy to derive from them. For example we can check if there is an applicable exception handler `proceed` is control dependent on handling exceptions thrown by `proceed`. Similarly all other statements in the body of the `around`-advice throwing exceptions have to be control dependent on a applicable `try .. catch` block.

6.4.4 Checking Data Dependences

The second cornerstone of our analysis is the analysis of data flow. A prerequisite for this analysis is the calculation of the $def()$ and $use()$ sets, which is relatively easy as a call graph is available. To evaluate the interference criterion we further need the function *objects* used in Definition 6.3.1, i.e. we need to know which memory locations are actually accessed through references collected in $def()$. This is a complex analysis problem as Java—and consequently also AspectJ—allows aliasing. Therefore we have to resolve aliasing in order to approximate *objects*. Therefore pointer analysis is a suitable technique.

Two important dimensions are *flow* and *context sensitivity*. Flow sensitivity keeps track of program control flow and allows to discard some constraints, i.e. if a pointer is overwritten before the underlying object is accessed. Context sensitivity keeps track of the calling context of a method, i.e. for two different calls to the same method, points-to sets for the actual parameters and local definitions are not identified but kept separate. Which analysis technique to use in general is a trade-off of time and space versus precision.

As for AspectJ no source level pointer analysis had been available, Florian Forster prototypically implemented a proof of concept source level pointer analysis[7] for an AspectJ core language [11] based on the AOPA framework and the BDDBDDB system [26]. We built the data flow interference analysis presented here on an extension of this pointer analysis. Handling of plain Java constructs is well known, thus we focus on handling of AOP constructs in this section. The analysis we implemented for our experiments is both flow and context insensitive (but object sensitive) and is thus rather imprecise, but fast.

Clearly our analysis would benefit from a more precise pointer analysis. Especially *context sensitivity* is interesting here as for our analysis a method called by two pieces of advice often result in conflicts—even if different actual parameter objects are involved. Our simple context-insensitive analysis is not able to keep these different actual parameter objects apart. However, we consider improvement of points-to techniques for AspectJ not as a core topic of the work presented in this thesis. This is also the reason why we only give a brief overview of our method here, as it is a relative straightforward extension of known Java points-to techniques. However we in detail describe how aspect language constructs are modeled to guarantee that our calculation is conservative. For an overview of pointer analysis techniques we refer the interested reader to [13].

To justify that we did not use available byte code analysis based pointer analysis implementation, remember that—while AspectJ is also compiled to Java byte code—advice no longer exists after compilation as it is transformed to basic Java constructs. Thus available implementations unfortunately are not usable in our context. Another source level pointer analysis for AspectJ was not available, available pointer analysis implementations for Java operate on Java byte code.

Note that a precise pointer analysis is not computable, and thus *conservative approximation* is necessary. For example pointer analysis in general identifies all objects created at a single creation site (`new`-statement in Java) with this creation site. For a given pointer[8] variable in the program we thus get a set of creation sites called the *points-to set* indicating that objects created at these sites potentially are accessible by this pointer.

To calculate points-to sets, it is necessary to trace all assignment operations in the program. The starting points are creation sites, which have a fixed points-to set associated with them (themselves). For each assignment we generate a constraint that the points-to set of the l-value is a superset of the points-to set of the r-value (following Andersen's algorithm [1]). Note that field access and assignment result in special assignments. For a field reference $a.x$ it is first necessary to evaluate the point-to set of the pointer a used to access field x to get a more precise points-to set for x.

Method calls are more complex to model, as a call implicitly defines a set of assignments: from actual to formal parameters and also from return value to the respective program variable storing the result (if appropriate). While modeling traditional program constructs is well known[9], the main challenge in this context is modeling the additional syntactic constructs introduced by AspectJ.

Modeling *inter type declarations* is straightforward. Inter type members are visible only in the context of the aspect if declared `private`; otherwise they act as normal class members with one important difference: members of the declaring aspect are also accessible in introduced code. We thus modeled inter type members similarly to regular target class members but adapted lookup rules accordingly.

Handling *advice* is more complex. We modeled advice similarly to methods; however

[7] We excluded nested classes and reflection and stop analysis when libraries are reached. The pointer analysis is also not able to deal with complex pointcut expressions using boolean operators.

[8] Java only knows references, but we will use the term pointer as a synonym in this section.

[9] Note however that we had to deal with nested statements and similar constructs. This however is only a simple syntactical transformation: we introduced virtual variables to store the results of nested expressions. For example $a.b.f(g.q);$ is transformed to $\$1 = g.q; \$2 = a.b; \$3 = \$2.f(\$1);$

Joinpoint	Current Object	Target Object	Arguments
method `call`	target object*	target object**	actual param.
method `execution`	target object*	target object*	formal param.
ctor `call`	target object*	None	actual param.
ctor `execution`	target object	target object	formal param.
`get`	target object*	target object**	None
`set`	target object*	target object**	assigned value

* There is no target object in static contexts such as static method bodies or static initializers.
** There is no target object for join points associated with static methods or fields.

Table 6.1: Semantics of `this`, `target` and `args` in AspectJ depending on the joinpoint kind (taken from the AspectJ language manual).

two important properties of advice have to be considered. First, we have to provide a mapping from exposed *joinpoint context* to formal advice parameters and, second, naturally there is no explicit *advice call*. Advice is implicitly applied at adapted joinpoints. However, as we know the relevant joinpoints (due to the advice joinpoint mapping from the weaver), we know where virtual "advice calls" have to be inserted, or rather which constraints have to be generated.

To provide the context mapping, we have to analyze what part of the joinpoint context has been made available to advice via the pointcut. Analysis of the pointcut declaration allows to identify respective variables. AspectJ offers three constructs to explicitly expose context to advice: `this`, `target` and `args`. Beside these, `after`-advice can also give access to return values and thrown exception objects. Determining the variable and thus the object referred to by these constructs depends on the nature of the joinpoint, Table 6.1 gives a short overview, for more details refer to the AspectJ manual. We identify the referenced context by analyzing pointcut definitions and interpreting any encountered `this`, `target` and `args` statement in the context of joinpoints matched by the respective pointcut definition. Once these objects have been identified, we handle "advice calls" similarly to method calls by assigning these objects to formal advice parameters and also creating a respective assignment for return values of `around`-advice.

While modifications of heap objects by advice are directly captured by our analysis, `around`-advice can additionally *reassign* parameter values. As a consequence the constraints generated to model the parameter passing potentially depend on the actual advice order.

Example 6.4.4 (Relevance of Precedence Order) *The example shown in Figure 6.3 illustrates this problem. Assuming `A1` has higher precedence than `A2`, parameters are assigned as follows, when method `zip()` is executed: (i) a = l; b = m; (actuals `foo` → `around` in `A1`), (ii) u = b; v = a; (`proceed` in `around/A1` → `around/A2`) and (iii) x = v; y = v; (`proceed` in `around/A2` → formals `foo`). The opposite order yields the following assignments: (i) u = l; v = m; (actuals `foo` → `around/A2`, (ii) a = v; b = v; (`proceed` in `around/A2` → `around/A1`) and x = b; y = a; (`proceed` in `around/A1` → formals `foo`).*

Figure 6.3 (b) shows the so called points-to graphs illustrating the data flow in both cases: solid arrows denote an assignment if `A1` has higher precedence than `A2`, dotted lines the other case. An arrow a → b indicates that a can point to any object b also points to. We assume that `l` and `m` are directly resolvable to the creation sites shown as rectangles and labeled accordingly. Evaluating the resulting constraints yields the points-to sets shown in the following table.

Variable	*a*	*b*	*u*	*v*	*x*	*y*
`A1` *before* `A2`	*{l}*	*{m}*	*{m}*	*{l}*	*{l}*	*{l}*
`A1` *before* `A2`	*{m}*	*{m}*	*{l}*	*{m}*	*{m}*	*{m}*

```
1  class SomeClass {
2    void zip() {
3      ...
4      foo(l, m);
5    }
6    void foo(Object x, Object y) {
7      ...
8    }
9  }
10 aspect A1 {
11   around(Object a, Object b):
12     call(foo(..)) && args(a, b){
13     // switched
14     proceed(b, a);
15   }
16 }
17 aspect A2 {
18   around(Object u, Object v):
19     call(foo(..)) && args(u, v) {
20     // only forward v
21     proceed(v, v);
22   }
23 }
```

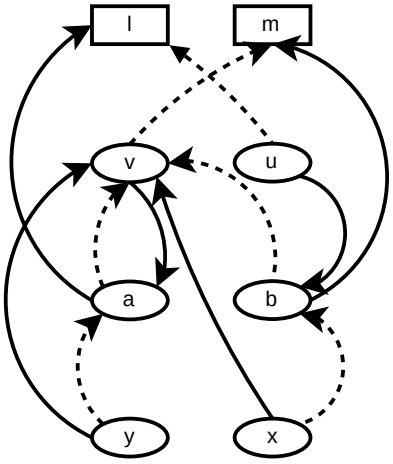


Figure 6.3: Generated constraints depend on advice order.

As points-to sets differ *depending on the execution order, we potentially miss conflicts in the interference analysis, if only one order is analyzed.*

Dynamic pointcuts raise a similar issue. AspectJ offers language constructs to restrict joinpoints matched by a pointcut. The keyword `if` for example allows to restrict joinpoint matching based on program values, `cflow` based on the shape of the call stack. Pointcuts containing these constructs cannot be evaluated statically in general. In this setup, simply assuming that advice is applied is not sufficient, as in this case some constraints might be lost if advice actually does not apply, similarly to the above case dealing with advice order.

A solution would be to create constraints for each possible order of advice and union the results. However, for n pieces of advice $n!$ different orders exists. If we also consider all subsets due to dynamic joinpoint matching this number even increases. As we do not know statically if a piece of advice referencing a dynamic pointcut actually matches at runtime we again have to model all possibilities (i.e. all subsets), resulting in total in 2^n different subsets for n pieces of (dynamically matching) advice. If we combine these two observations, we end up with $\sum_{i=1}^{n} i!\binom{n}{i}$ different possibilities. For 5 pieces of advice this results in 326 different possible execution scenarios for advice to model. This combinatorial explosion clearly demands a different solution.

To avoid it we use a simple trick. Instead of only generating one assignment from actual to formal parameters (following Andersen [1]), we also generate the opposite assignment to identify both points-to sets (as suggested by Steensgard [22]). While this approach is more imprecise it allows to conservatively approximate all possible assignment orders.

However loss of precision is again constrained as this construction is only necessary for dynamic joinpoints or undeclared precedence order if at least two pieces of advice apply at the same joinpoint and also share some exposed context. Otherwise advice application and order

is determined and constraints can be created accordingly. Thus we only use this construction if necessary, reducing introduced imprecision and overhead to a minimum.

With these models for advice and inter type declarations for both the pointer analysis and the call graph it is now possible to derive necessary constraints from the AspectJ source code. We use these constraints together with the call graph as inputs for the BDDBDDB system [26], which uses binary decision diagrams to efficiently solve them.

6.4.5 Implementation and Example

Implementing the interference criterion is straightforward once call graph, exception-aware control flow graph, and results of the pointer analysis are available. For each potentially conflicting pair of advice a_1 and a_2, we first examine $def(a_1)$. For each field store $a.x$ in the control flow of a_1, we first check if the field name x can be found in a field load or field store in the control flow of a_2. Let $b.x$ be such an entry. For each such entry we check if the intersection of the points-to sets of a and b are non-empty. If this is the case, we can report that both pieces of advice potentially interfere on the object both a and b point to. If one of the two pieces of advice is `around`-advice, we also check if `proceed`-parameters are modified or the return value is modified by the aspect. This is a straightforward implementation of the criterion (2) in Definition 6.3.3.

If we are only interested in a binary information (interference or not), we can abort the analysis for two pieces of advice once a criterion violation is found. However in general it is interesting for programmers to know *why* two pieces of advice interfere. We thus continue with the analysis to collect all objects and access patterns where the two pieces of advice potentially conflict. Thus even if our analysis reports false positives, the programmer has more detailed information to make a well-informed decision if or how precedence for two aspects has to be declared.

We implemented our analysis as a set of plug-ins for the Eclipse IDE as a part of the AOPA framework. Our system correctly identifies the data flow conflict if the `declare precedence` statement is removed in the `Telecom` example. The pointer analysis reveals that both `Timing` and `Billing` access the same `Timer` objects, where `Timing` sets `stopTime` and `Billing` reads this value to finally bill the call. As neither piece of advice throws a (checked or declared) exception, no control flow interference is found here.

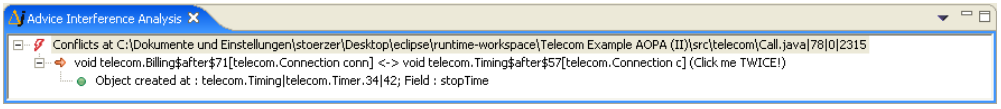


Figure 6.4: Interference Analysis Results for `Telecom`

Figure 6.4 shows a screen shot of the results presented to the user. Clicking on the first line of the view opens the editor and presents the joinpoint affected by both pieces of advice to the user. Clicking on the second line in turn opens either the `Timing` or the `Billing` advice and the final line allows to access the creation site of the timer object, thus allowing the user to quickly examine the analysis results. If aspect order is defined by adding the `declare precedence` statement, this is also detected by our system and no warnings are generated. Clearly our example can only give a first impression of the effectiveness of our approach and additional case studies are needed to better evaluate it.

Note that our current implementation is only a simple proof-of-concept prototype (mostly due to limitations in the pointer analysis). Although experience with our prototype is limited, we expect our system to scale to at least medium size programs as we use an efficient system for the most expensive part of the analysis—the pointer analysis. BDDBDDB claims

to provide this scalability. For medium size AO systems maintaining an overview of all applying aspects and their peculiarities is already hard, and thus applying our system to capture precedence related problems is valuable.

The precision of our analysis could clearly benefit from improvements in the pointer analysis underlying our algorithm, especially by using a context-sensitive pointer analysis. Our current implementation is also only applicable to a rather limited AspectJ core language. Especially the lack of support for library code and reflection, nested classes and the complex pointcut expressions has to be mentioned here. Future work will on one hand increase precision for our analysis by switching to such an analysis. Second, we are currently in the process of refactoring open source Java systems to generate some subjects to better evaluate our system and also generate some interesting runtime data.[10] Using this information we can now determine if a data dependence among two pieces of advice exists. To further increase precision and reduce spurious data dependence warnings, we could also try to resolve dynamic joinpoints statically as far as possible. As this is also an important topic for AspectJ performance optimization [3] and program analysis research in general we consider this to be orthogonal to our work.

6.5 Related Work

Our work is related to pointer analysis and aspect interference analysis. We start with a very short introduction selected work on pointer analysis, as there is a large body of work on this topic in literature.

Andersen presented a subset-based algorithm for pointer analysis for the C language in [1], which is basically also used in this work. As this algorithm is relatively expensive ($O(n^3)$), Steensgard [22] proposed a simpler algorithm which identifies points-to sets on assignments, resulting in almost linear runtime, however also a considerable loss of precision. Since then several other approaches and optimizations have been proposed. We refer to [13] for an overview of available algorithms. Extensions for object-oriented languages have to deal with dynamic binding, for example refer to [4, 7, 18].

Although the problem of aspect interference has been recognized by researchers, there are still only few approaches in literature addressing this problem. In [6] the aspect interaction problem is discussed in a position paper, although on a more abstract level and targeted to the composition filter approach [5]. While this work contains an interesting discussion of problem itself, a solution is only briefly outlined.

In [19] a reflective aspect-oriented framework is proposed which allows users to visually specify aspect-base and aspect-aspect dependences using the Alpheus tool. The tool is also used to specify aspect conflicts and resolution rules which are then resolved automatically at runtime. While the framework offers a more abstract view on aspects and provides a richer set of conflict resolution rules than AspectJ (thus leveraging some of the problems discussed in this paper), conflict detection is manual.

In [9, 20], a non-standard but base-language independent aspect-oriented framework, including support for conflict detection and resolution, is discussed. The presented conflict resolution mechanisms are more powerful that the `declare precedence` construct of AspectJ. However the presented model does not handle `around`-advice and bases conflict detection on multiple pieces of advice applying to a single joinpoint only. Our method in contrast explicitly analyzes advice for commutativity thus reducing the number of false positives.

Aspect interference and dependencies are gaining more interest by researchers recently. A workshop on this topic has been organized [8], to initiate a wide discussion on this topic. Work presented there discusses functional aspects in the context of refactoring [2], reasoning about semantic dependences [10], and also work on dependence analysis design level and

[10] For the `Telecom` example runtime is just few seconds.

requirements level dependency analysis (although this work is not directly related to the work presented here).

Feature Oriented Programming [17, 15] has been proposed as an alternative to aspect-oriented programming. While this approach has a similar power compared to aspect-oriented programming, explicit algebraic composition expressions avoid undefined aspect precedence up front, thus eliminating the necessity of detecting them later. In [16], Herrejon et. al. examine an algebraic approach to aspect composition which, similarly to feature oriented programming, avoids several interference problems of pure aspect-oriented programming by replacing *unbounded* with *bounded quantification*.

In [21] Rinard et. al. propose a combined pointer and effect analysis to classify aspects by their effects on the base system. While we use a similar underlying analysis, our work differs from theirs in several ways. First, we apply the analysis to determine effects of aspect-aspect rather than aspect-base interference. Second, their algorithm—while more precise—is also considerably more expensive than our current simple BDDBDDB-based pointer analysis, thus our approach potentially allows to analyze larger systems. As our analysis focuses on the special case of joinpoints with multiple pieces of applied advice, loss in precision seems acceptable in favor of gained performance.

Finally, in [14] Ishio et. al also address the increased complexity of AspectJ. To support program debugging they propose two techniques: First, they analyze call graphs and implemented a tool to automatically detect potential infinite recursion due to careless pointcut design. Second, they calculate dynamic slices based on a technique called DC slicing to help programmers isolate failures in code. To create the underlying data structures, they discuss similar problems as discussed here and also in part propose similar solutions, although application of their work is completely different.

The problem that conflicts are not reported at all has also been reported as a bug for AspectJ, and a compiler warning has been suggested to deal with this problem. While this makes programmers aware of potentially conflicting advice, our analysis is able to detect commutative advice thus providing more precise feedback and avoiding some spurious warnings.

6.6 Conclusion

To summarize, the contributions of our analysis presented in this chapter are threefold.

1. We provided an in depth analysis of the advice precedence problem and demonstrated its relevance.
2. We defined an interference criterion to check for relevant undefined advice precedence.
3. We used standard program analysis techniques to prototypically implement this criterion and discussed the results of our implementation for the `Telecom` example.

Our approach can support programmers working with AO systems who have to deal with the advice precedence problem by helping to avoid unexpected side effects during system construction and evolution. Thus the analysis presented here directly attacks problem P2: Aspect Interaction and to a lesser degree P4: Base Comprehensibility as outlined in Chapter 2.

Bibliography

[1] ANDERSEN, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] APEL, S., AND LIU, J. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In Chitchyan et al. [8], pp. 1–9.

[3] AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÀK, J., LHOTÀK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 117–128.

[4] BACON, D. F., AND SWEENEY, P. F. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1996), ACM Press, pp. 324–341.

[5] BERGMANS, L., AND AKSITS, M. Composing crosscutting concerns using composition filters. *CACM 44*, 10 (2001), 51–57.

[6] BERGMANS, L. M. Towards detection of semantic conflicts between crosscutting concerns. *Proceedings of workshop AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003* (2003).

[7] BERNDL, M., LHOTÀK, O., QIAN, F., HENDREN, L., AND UMANEE, N. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM Press, pp. 103–114.

[8] CHITCHYAN, R., FABRY, J., BERGMANS, L., NEDOS, A., AND RENISNK, A., Eds. *ADI'06 , Workshop on Aspects, Dependencies and Interaction, Proceedings, published as technical report COMP-001-2006 Lancaster University, Computing Department, held at ECOOP'06* (Nantes, France, July 2006).

[9] DOUENCE, R., FRADET, P., AND SÜDHOLT, M. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development* (New York, NY, USA, 2004), ACM Press, pp. 141–150.

[10] DURR, P., BERGMANS, L., AND AKSIT, M. Reasoning about Semantic Conflicts between Aspects. In Chitchyan et al. [8], pp. 10–18.

[11] FORSTER, F. Points-to analyse und darauf basierende advice interferenzanalyse für eine kernsprache für aspectj. Master's thesis, Universität Passau, Passau, Germany, September 2005.

[12] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] HIND, M., AND PIOLI, A. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2000), ACM Press, pp. 113–123.

[14] ISHIO, T., KUSOMOTO, S., AND INOUE, K. Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2004), IEEE Computer Society.

[15] LOPEZ-HERREJON, R., BATORY, D., AND LENGAUER, C. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 2006), ACM Press, pp. 68–77.

[16] LOPEZ-HERREJON, R., BATORY, D., AND LENGAUER, C. A disciplined approach to aspect composition. In *Proc. ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM 2006)* (2006), ACM Press, pp. 68–77.

[17] LOPEZ-HERREJON, R. E., BATORY, D., AND COOK, W. Evaluating support for features in advanced modularization technologies. In *ECOOP 2005 - Object-Oriented Programming: 19th European Conference, Proceedings* (July 2005), A. P. Black, Ed., Springer Berlin / Heidelberg, p. 169.

[18] MILANOVA, A., ROUNTEV, A., AND RYDER, B. G. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol. 14*, 1 (2005), 1–41.

[19] PRYOR, J. L., AND MARCOS, C. Solving conflicts in aspect-oriented applications. In *Proceedings of 4th Argentine Symposium in Software Engineering* (Buenos Aires, Argentina, September 2003).

[20] RÉMI DOUENCE, P.FRADET, M. S. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)* (October 2002).

[21] RINARD, M., SALCIANU, A., AND BUGRARA, S. A Classification System and Analysis for Aspect-Oriented Programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (New York, NY, USA, 2004), ACM Press, pp. 147–158.

[22] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[23] STERR, R. Bachelor thesis: Aspect orthogonality – control flow preservation. Master's thesis, Universität Passau, Passau, Germany, December 2004.

[24] STOERZER, M., STERR, R., AND FORSTER, F. Detecting precedence-related advice interference. In *In Proceedings of 21th International Conference on Automated Software Engineering (ASE), to appear* (September 2006), IEEE Press.

[25] TIP, F., AND PALSBERG, J. Scalable Propagation-based Call Graph Construction Algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2000), ACM Press, pp. 281–293.

[26] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation* (New York, NY, USA, 2004), ACM Press, pp. 131–144.

7 Subjects for Case Studies

7.1 Available Aspect-Oriented Systems

We first give an overview of available AspectJ systems. Unfortunately, the list is short, and not all of the systems listed below are useful subjects to study, as some of them are either very small, only available in a single version, and/or are not associated with a (JUnit) test suite.

7.1.1 AspectJ Examples

The AspectJ distribution itself comes with a few small example programs, `Bean`, `Introduction`, `Observer`, `TJP` (reflective joinpoint access) and `Tracing`. The most interesting of them however are the `Telecom` example and the `SpaceWar` program.

The `Telecom` example has already been described in Section 6.2, however we will briefly recapitulate this description here. The program simulates a telecommunication system. The base system provides functionality to set up and drop connections between different customers. Two aspects, `Timing` and `Billing` extend the system by recording the call duration and billing the calling customer, respectively. While the `Telecom` example is not associated with a JUnit test suite, three classes providing a main method are part of the system, to execute the system in a controlled way. These classes can (and have been) transformed to simple JUnit test cases.

`SpaceWar` implements a clone of the famous "Asteroids" computer game. The program uses aspects for *debugging* (i.e. a tracing aspect), *thread control* (`SpaceWar` incorporates the `Coordination` example, which is also part of AspectJ), *access control* and to implement the *observer pattern*. `SpaceWar` uses both inter type declarations and advice in its aspects. In contrast to the `Telecom` example, `SpaceWar` also uses `around`-advice. It is thus a more interesting subject to study. In total `SpaceWar` has 3052 lines of commented AspectJ code (including the Coordination example code). Unfortunately there is no test suite for `SpaceWar`, and—as `SpaceWar` is an interactive application—providing a simple test suite is also not straight forward. These properties reduce the usefulness of `SpaceWar` as a subject for case studies relying on dynamic analysis techniques. Both the `Telecom` example and `SpaceWar` are only available in a single version.[1]

[1] Apart from code modifications to adapt to changed AspectJ syntax.

Program	LoC System (.java+.aj)	LoC Aspects (.aj)
Bean	394	100
DCM	4225	339
LawOfDemeter	4225	147
NullCheck	5233	77
ProdLine	2601	1195
Tetris	5215	408

Table 7.1: The *abc* test suite—Size in Lines of Commented AspectJ Code.

7.1.2 The *abc* Test Suite

Comparable to the AspectJ examples, the programs of the *abc* test suite are not available in different versions and thus only of limited value for the evaluation of change impact analysis. The *abc* test suite consists of a few programs without any associated JUnit test suite. Thus it is most useful as a subject for static analysis techniques. However, some batch test drivers exist which can be used to create JUnit tests to execute the system in a controlled way (although these tests per definition always succeed as long as the program does not crash). Note however that the resulting tests are qualitatively different from a fine grained JUnit unit test suite as they in general show the characteristics of system regression tests. While the programs are a bit larger than the AspectJ examples, they are still small. This is especially true for the size of the applied aspects, with exception of the `DCM`, `ProdLine`, and `Tetris` examples. Table 7.1 gives an overview of the size of the programs (as lines of commented AspectJ code).[2]

Base Application: `CertRevSim`

Several of the example aspects discussed below (`LawOfDemeter`, `NullCheck`, and `DCM`) are applied to the same application provided by a Java package `certrevsim`. The classes provided in this package are part of the *Certificate Revocation Performance Simulation Project*. According to its web-site[3], the purpose of this project is to simulate some of the certificate revocation schemes described in the author's thesis, especially to derive performance data.

While the code provided with the *abc* test suite does not provide a (JUnit) test suite, the web site[4] lists a set of reasonable *inputs and results* which can be used to write a JUnit test suite with 15 distinct tests. Some of the analysis techniques discussed in this thesis trace test execution to derive their results. During experiments with the aspect impact analysis we aborted execution of a single test[5] with applied `LawOfDemeter` concern after two days runtime and a produced trace file of > 60 GB. While tracing in general involves overhead, this massive overhead was nevertheless astonishing. We thus analyzed the mere overhead resulting from aspects, and got the results shown in Table 7.2. As can be seen, both `LawOfDemeter`

	`plain system`	`LawOfDemeter`	`NullCheck`	`DCM`
Execution Time	0.33 s	198.07 s	1.23 s	18.85 s
Overhead Factor	1	600.2	3.7	57.1

Table 7.2: *abc* benchmark results: Execution times for `java CertRevSim/Simulator 30 10000 25 10 1 1 1` in the given configurations (user time measured with Unix `time` command, averages over 5 runs).

[2] Due to the size of just 117 LoC we omitted the `Figure` example.
[3] `http://www.pvv.ntnu.no/~andrearn/certrev/sim.html`
[4] `http://www.pvv.ntnu.no/~andrearn/certrev/results.html`
[5] `java certrevsim/Simulator 30 10000 25 10 1 1 1`

(execution time multiplied by factor 600) and DCM have considerable impact on runtime, and thus—although maybe not in a 1:1 manner—also on trace sizes. As the plain trace reaches sizes of around 3.7 GB (uncompressed), this would mean an estimated trace size of several 100 GB for the LawOfDemeter example. Thus for dynamic analysis techniques not all tests were usable due to the high aspect overhead.

Unfortunately only a *single version* of this base application was available to us, so we could not explore different system versions by changing the base program version.

The Bean Example

The Bean example comes with a single aspect BoundPoint, which adds property change listener support to a plain Java Point class, thus allowing usage of the class in the Java Bean framework. Furthermore, the aspect adds serialization support to class Point.

The Bean example comes with a main(..) method which can be used to execute the system in a controlled way. However, to make the non-aspectized version compilable, we also had to remove two lines in the Demo.main(..) method, which resulted in compiler errors as they explicitly reference methods introduced by the aspect.

The DCM Example

The DCM example computes dynamic coupling metrics as described in [8]. It defines three aspects, AllocFree, ClassRelationship, and Metrics and additionally three pointcuts in a separate aspect (Pointcuts) which each implement a certain (dynamic) metric. For example aspect AllocFree counts constructor invocation (i.e. object creation) and their finalization (i.e. final object deletion). [6]

All aspects in this example are optional, all aspects can be added/removed without provoking compiler errors, as long as their auxiliary classes are also added.

The base system the aspects are applied to is the CertRevSim system. Thus all properties stated for this system apply.

The LawOfDemeter Example

The LawOfDemeter example program implements a policy enforcement aspect. The provided aspects check if a program adheres to the *Law of Demeter* [12, 13]. The implementation consists of aspects Pertarget, Percflow, Check and Any. The first three aspects depend on the latter, so Any has to be applied if any other aspect should be applied.

The base system the aspects are applied to is the CertRevSim system. Thus all properties stated for this system apply.

The NullCheck Example

Similarly to the LawOfDemeter example the NullCheck example also demonstrates a policy enforcement aspect. Aspect EnforceCodingStandards is used to monitor if a method returns a null reference and in that case prints an error message.

Again, the the aspects are applied to is the CertRevSim system. Thus all properties stated for this system apply.

The ProdLine Example

The ProdLine example uses aspects to independently define features for a system, which are then combined on demand, so allowing to define a set of similar applications, i.e. a product

[6] Note that this is a potential problem if a finalize() method already exists in a class to be monitored. In that case the defined method is used, which most likely does not report object deletion as the interface method does.

Aspect	Purpose	Category
Benchmark	a profiling aspect	development
ProgTime	test driver for profiling	development
CC	calculate connected components	feature
Cycle	find cycles	feature
DFS	depth first traversal	feature
MSTKruskal	Kruskal's algorithm	feature
MSTPrim	Prim's algorithm	feature
NoPrinting	empty log method implementation	(feature)
Undirected	defined an undirected graph	(base) feature
Weighted	adds edge weights	feature

Table 7.3: Aspects defined by the `ProdLine` example. All aspects are optional, but dependences among these aspects exist.

line. This example specifically defines a graph application and adds several graph algorithms to the system by using respective aspects. Algorithms for graph traversal (`DFS`), strongly connected component calculation (according to the JavaDoc; aspects `MSTKruskal` and `MSTPrim`, although these algorithms most likely calculate minimal spanning trees), etc. Beside the feature aspects, `ProdLine` also defines a profiling aspect (aspects `Benchmark` and `ProgTime`) These aspects are all structurally very similar. Table 7.3 gives an overview of all aspects and their purpose.

All algorithms defined by aspects (in total there are 11 aspects) are optional, i.e. the system without any aspect applied compiles. However, dependences among the different aspects exists, i.e. some aspects use algorithms implemented by others in their implementation (for example aspect `DFS` needs aspects `Undirected` and `NoPrinting` to compile).

For a certain aspectized program version, the `main()`-method of `ProgTime` can be used as a test driver. This is however the only `main()`-method defined in the program.

The `Tetris` Example

Aspect	Purpose	Category
TestAspect	check images only loaded once	development
DesignCheck	check design rules	development
GameInfo	panel to display information	feature
Menu	extends GUI with a menu	feature
Counter	count deleted lines	feature
Levels	uses Counter to advance level	feature
NewBlocks	adds two new types of blocks	feature
NextBlock	adds the next block preview	feature

Table 7.4: Aspects defined by the `Tetris` example. All aspects are optional, but dependences among these aspects exist.

We finally examine the `Tetris` example. As the informed reader might have guessed this is an implementation of the famous computer game with the same name. For this example a set of optional aspects is defined, which is shown in Table 7.4.

The aspects provided for this program can be classified as either development aspects (aspects `TestAspect` and `DesignCheck`) or additional features added to the system. Similarly to the `ProdLine` example, all aspects are optional, although (in part only semantic!) dependences among different aspects exist (for example `Levels` needs `Counter` to work, but also

compiles without this aspect applied).

`Tetris` as an interactive computer game is hard to test in general. However, the system provides a `Driver` class which allows to replay a certain interaction. This driver function can be used as a test driver. We recorded player actions using the support in the game to derive four test cases, which can be distinguished by the different play length, which affects level advancement, high score, etc.

abc—Summary

To summarize, the main problem of the *abc* test suite for evaluation of the methods and techniques presented in this thesis is the lack of multiple versions and a existing test suite. We can generate distinct versions by only applying subsets of the provided aspects, but this is qualitatively different from comparing versions with actual code edits.

Fortunately we can however use most projects to evaluate not only our static, but also our dynamic analysis techniques, as we can execute most aspectized systems either by using provided test drivers (`Tetris`) or by using the batch interface of `CertRevSim`.

7.1.3 Auction System

Awais Rashid from Lancaster University provided us with an auction system implemented in AspectJ we could use for our case studies. The system has been implemented by Damon Oram as a student project during his master studies.

Aspect	Purpose	Category
`AuctionUpdateNotificationAspect`	user feedback	observer
`CreditLog`	security log	security, logging
`RefreshAuctionStateAspect`	auction status updates	observer
`SerializationAspect`	serialization support	persistence
`TestAspects`	tracing aspect	tracing
`AuctionSystemExceptionHandler`	exception handling	error handling
`DataManagementAspect`	persistence handling	persistence
`MaintainAuctionIDAspect`	auction id handling	persistence
`ValidateUserAspect`	check user objects	security

Table 7.5: Aspects defined by the Auction System. All aspects except aspect `AuctionSystemExceptionHandler` can be removed without provocing compiler errors.

The system comes with 9 aspects which we summarized in Table 7.5. Although the system is rather small (2354 lines of commented Java code and 670 lines of AspectJ code, in total 3024 lines), it is very intersting as aspects are used for many different features in the system. However, all aspects can be classified in the few well-known standard categories, as can be seen in Table 7.5.

Apart from aspect `AuctionSystemExceptionHandler` all aspects can be removed from the system without provoking compiler errors. However, in this case, the system does not produce any output any more, i.e. the resulting funcitonally is *not* equivalent. Unfortunately only a single version of the system is available, i.e. the system is of only limited benefit for multi version program analysis, but an interesting case study to determine the effects of aspect impact.

While the system does not provide a JUnit test suite, it comes with a text file describing a series of tests we used to create a small JUnit test suite consisting of six different tests.

7.1.4 AspectJ Development Tools

Since November 2004, the AspectJ Development Tools (*ajdt*) integrated in Eclipse use an observer aspect in the *org.eclipse.ajdt.ui* package. While this is "only" an observer aspects, several versions are available, which allows to study changes between versions.

The *ajdt* plug-ins are one of the larger available systems, however there are three restrictions: (i) *ajdt* is an Eclipse plug-in projects, and thus cumbersome to analyze dynamically as it involves the Eclipse runtime, (ii) the contained aspect is not very interesting, and (iii) building older versions is cumbersome due to dependences to older Eclipse plug-in.

Additionally we are not aware of a test suite for *ajdt*. Consequently *ajdt* is also not a useful subject for dynamic analysis techniques. Especially the build problems with older version of *ajdt* due to the dependences to old Eclipse plug-in versions and the limited use of aspects led us to abandon *ajdt* as a case study. We however mention it here for completeness reasons.

7.1.5 AJHotDraw

AJHotDraw [18] is a aspect-oriented version of the JHotDraw project.[7] Cited from the homepage:

> JHotDraw is a Java GUI framework for technical and structured Graphics. It has been developed as a "design exercise" but is already quite powerful. Its design relies heavily on some well-known design patterns. JHotDraw's original authors have been Erich Gamma and Thomas Eggenschwiler.

The AJHotDraw project used this system and refactored it to use some aspects. Currently three versions of AJHotDraw are available on SourceForge[8], in particular:

AJHotDraw 0.1 Version 0.1 is the original JHotDraw 6.0 source. However the test suite adds some invariants checking aspects in the tests.

AJHotDraw 0.2 Released February 6, 2005. Includes a refactored *Persistence Concern* for selected figures.

AJHotDraw 0.3 April 15, 2005. Includes an aspect-oriented refactoring of a contract enforcement concern for commands, and an exemplary aspect-oriented refactoring for an instance of the Observer pattern.

The developers of AJHotDraw in parallel also maintain the test project *TestJHotDraw* to guarantee some degree of semantic equivalence. While best to our knowledge this is the largest system publicly available at the moment, it only contains few aspects, and the last version is more than a year old. However it is the most interesting publicly available subject to study as a plain Java (plus aspects in the test suite) and two different aspect-oriented versions as well as a test suite are available.

Consider Table 7.6 for an overview of all aspects introduces in the different versions. The first column "VS" gives the program version the aspects are introduced in. The next two columns name the different aspects and give a short comment on their purpose. The fourth column classifies the aspects according to the concern these aspects belong to. The final column states if the aspect is optional, i.e. if the aspect can be removed without resulting in compiler errors.

We start our discussion with version 0.1 of AJHotDraw. While the base system is still the original Java base system, the test suite is augmented with a set of aspects for invariant checks and the equivalence services needed by the tests.

[7] http://www.jhotdraw.org/
[8] http://ajhotdraw.sourceforge.net/

VS	Aspect	Purpose	Feature	Rem.?
0.1	FigureEquivalence	equivalence services for tests	tests	no
0.1	Invariant	abstract invariants aspect	invariants	yes
0.1	FigureInvariants	concrete invariants for `AJHotDraw`	invariants	yes
0.1	InvariantTest	(testing aspect for invariants)	invariants	yes
0.2	PersistentAttributeFigure	adds methods `read` and `write`	persistence	yes
0.2	PersistentCompositeFigure	adds methods `read` and `write`	persistence	yes
0.2	PersistentDrawing	adds methods `read` and `write`	persistence	no
0.2	PersistentFigure	adds methods `read` and `write`	persistence	no
0.2	PersistentImageFigure	adds methods `read` and `write`	persistence	yes
0.2	PersistentTextFigure	adds methods `read` and `write`	persistence	yes
0.3	CmdCheckViewRef	check view reference not `null`	design rule	yes
0.3	FigureSelectionObserverRole	(figure selection) observer role	observer	no
0.3	FigureSelectionSubjectRole	(figure selection) subject role	observer	no
0.3	SelectionChangedNotification	change of active figure	observer	yes

Table 7.6: Aspect in the different `AJHotDraw` versions.

There are three aspects related to invariants checking. One of these however is a simple test aspects which does not adapt any joinpoint in the `AJHotDraw` system. From the two remaining aspects, one is an abstract aspect providing some basic invariants checking functionality, while the third aspect instantiates the invariants checks for `AJHotDraw`. Invariants checking is optional.

Aspect `FigureEquivalence` introduces methods `boolean equivalent(Storable s)` to relevant figure classes. The test suite uses these methods to check if a stored figure is equivalent to the figure loaded from disk. Consequently this aspect is not optional as the test suite will not compile without it. However the introduced methods are not references by the base system itself, i.e. the base system compiles, only the tests depend on this aspect.

`AJHotDraw` version 0.2 introduces aspects to encapsulate persistence. Examining the respective aspects in more detail reveals interesting details. For each figure class providing persistence functionality, a specific persistence aspect exists which introduces `write()` and `read()`-methods to the respective target class. In other words, persistence is implemented with inter type declarations only. These introduced methods are then called by standard base system classes. However, as empty default implementations are introduced to the abstract super class `AbstractFigure`, at least four persistence aspects can be removed, although `PersistentDrawing` and `PersistentFigure` are necessary for the system to compile. Note that persistence has only been refactored by example, but not for the complete figure hierarchy. Thus several `read()` and `write()`-methods remained in their original classes.

We will now discuss aspects added in version 0.3. Therefore consider Table 7.6. The new aspects provided by this program version can be classified in two categories: a design rule enforcement aspect and the observer aspect. Note that the aspects of version 0.2 are still in the system (semantically unchanged). Unfortunately only the design enforcement feature can be removed completely from the system without provoking compiler errors.

For the observer feature the situation is similar. Two aspects define the observer and subject rules, respectively, and also use inter type declarations to add relevant methods and fields to respective target classes. These aspects also contain several `declare parents` statements. Aspect `FigureSelectionSubjectRole` also defines two pieces of advice to initialize and reset the list of observers. The observer update is then triggered in part by aspect `SelectionChangeNotification`, in part however also by explicit calls to method `figureSelectionChanged()`. Thus only the update aspect `SelectionChangeNotification` can be removed without provoking compiler errors.

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	419	8%	1098/13173	9%	152/1708	1.553
org.jhotdraw	1	62%	5/8	100%	1/1	0
org.jhotdraw.applet	4	0%	0/253	0%	0/23	1.617
org.jhotdraw.application	20	4%	21/520	0%	0/52	1.443
org.jhotdraw.aspects.equivalence	5	91%	68/75	97%	33/34	0
org.jhotdraw.aspects.invariants	12	35%	23/65	31%	4/13	1
org.jhotdraw.cmdcontracts	1	75%	3/4	100%	1/1	0
org.jhotdraw.commands	8	44%	91/205	43%	6/14	0
org.jhotdraw.contrib	127	0%	5/2467	0%	0/310	1.645
org.jhotdraw.contrib.dnd	10	0%	0/461	0%	0/77	2.167
org.jhotdraw.contrib.html	46	0%	0/801	0%	0/94	1.477
org.jhotdraw.contrib.zoom	14	0%	0/512	0%	0/52	1.54
org.jhotdraw.figures	163	10%	186/1860	11%	25/237	1.604
org.jhotdraw.framework	26	63%	74/117	100%	12/12	1.063
org.jhotdraw.lib	1	N/A	N/A	N/A	N/A	0
org.jhotdraw.observselection	5	17%	7/42	0%	0/3	1
org.jhotdraw.persistence	22	96%	197/206	100%	15/15	0
org.jhotdraw.samples.javadraw	27	0%	0/448	0%	0/40	1.488
org.jhotdraw.samples.minimap	2	0%	0/19	N/A	N/A	1
org.jhotdraw.samples.net	9	0%	0/93	0%	0/7	1.412
org.jhotdraw.samples.nothing	2	0%	0/46	N/A	N/A	1
org.jhotdraw.samples.pert	12	0%	0/228	0%	0/21	1.442
org.jhotdraw.standard	201	8%	254/3030	8%	36/441	1.547
org.jhotdraw.util	52	10%	158/1610	7%	19/258	1.781
org.jhotdraw.util.collections.jdk11	5	0%	0/92	0%	0/3	1.015
org.jhotdraw.util.collections.jdk12	1	67%	6/9	N/A	N/A	1
testundo	1	0%	0/2	N/A	N/A	0

Figure 7.1: Coverage Report for version 0.3 of `AJHotDraw`, measured using *Cobertura*.

Note that the test suite provided by `AJHotDraw` is rather small—it only comprises 16 tests. Not surprisingly the coverage is not very high. Figure 7.1 shows the summary coverage report generated with *Cobertura*.[9] Note that while the overall coverage is very low (only 8% statement coverage and 9% branch coverage), the test suite is designed to explicitly test the aspectized functionality. For the relevant packages `org.jhotdraw.aspects.equivalence` (91% and 97%), `org.jhotdraw.aspects.invariants` (35% and 31%), and `org.jhotdraw.aspects.persistence` (96% and 100%) coverage is considerably higher.

To summarize, the aspects in `AJHotDraw` massively rely on inter type declarations. Introduces methods are also referenced by base classes, thus these aspects are in part tightly coupled with the base system and cannot be removed without invasive system changes. As this clearly is in conflict with the obliviousness property of aspect-orientation, this might be another argument to criticize obliviousness as a characteristics of aspect-orientation.

7.2 Refactoring the Open Source Java Database `HSQLDB`

To address the lack of aspect-oriented programs to study we decided to create an aspect-oriented version of `HSQLDB`. `HSQLDB`[10] is an implementation of a relational database system in Java. The system is for example used as a part of the OpenOffice office suite. Cited from the homepage:

> `HSQLDB` is the leading SQL relational database engine written in Java. It has a

[9] `http://cobertura.sourceforge.net/`
[10] `http://www.hsqldb.org/`

> JDBC driver and supports a rich subset of ANSI-92 SQL [...] plus SQL 99 and 2003 enhancements. It offers a small (less than 100k in one version for applets), fast database engine which offers both in-memory and disk-based tables and supports embedded and server modes. Additionally, it includes tools such as a minimal web server, in-memory query and management tools (can be run as applets) and a number of demonstration examples.

HSQLDB also has a JUnit test suite which we used to guarantee at least a certain degree of semantical equivalence for the refactored version. For HSQLDB the coverage of the available test suite (both batch and JUnit tests) is acceptable, with a statement coverage of 48 % and a branch coverage of 51 % (average over all packages, measured with *Cobertura*).

Package	# Classes	Line Coverage	Branch Coverage	Complexity
(default)	1	0% 0/2	N/A N/A	1
All Packages	291	46% 20297/44344	51% 3930/7639	3,481
org.hsqldb	98	74% 14112/19022	81% 2853/3541	4,276
org.hsqldb.index	1	N/A N/A	N/A N/A	1
org.hsqldb.jdbc	16	34% 775/2311	41% 148/360	2,04
org.hsqldb.lib	63	46% 1108/2433	50% 264/528	2,257
org.hsqldb.lib.java	1	65% 17/26	100% 2/2	1,3
org.hsqldb.persist	20	47% 1035/2200	56% 210/377	2,875
org.hsqldb.resources	1	73% 37/51	89% 8/9	3,143
org.hsqldb.rowio	12	50% 421/849	41% 48/117	2,004
org.hsqldb.sample	5	0% 0/285	0% 0/21	3,63
org.hsqldb.scriptio	8	62% 307/494	72% 44/61	1,99
org.hsqldb.store	8	69% 564/819	76% 174/228	4,62
org.hsqldb.test	56	31% 1907/6249	29% 178/611	3,297
org.hsqldb.types	2	39% 14/36	17% 1/6	2,077
org.hsqldb.util	89	0% 0/9567	0% 0/1778	4,84

Classes in this Package	Line Coverage	Branch Coverage	Complexity
hsqlServlet	0% 0/2	N/A N/A	1

Figure 7.2: Coverage for HSQLDB, measured with Cobertura.

A first step for such a refactoring project is clearly to identify relevant crosscutting implementations in the available code base. While several automatic aspect mining techniques exist (for example [1, 2, 16, 19]), all these techniques suffer from two important problems: (i) the result set often contains a considerable amount of false positives, and (ii) the seed quality, i.e. the percentage of the crosscutting implementation actually discovered by the tool can be low, and thus nevertheless requires manual work to identify the code of a crosscutting concern.

Due to these restriction—and also to try something new—we used another approach in this project. We started with an accepted catalog of well-known crosscutting concerns—for this project we used Laddad's book [11]—and then tried to find classes, methods, or fields related to the respective concern. For example we tried to find a crosscutting implementation of a caching/value pooling concern by looking for the keyword "`pool`". Once a matching program element—for example a class—was found we used FEAT [15], a Java cross referencing and concern mining tool to find all code related to this class potentially relevant for refactoring. The JQuery tool [9] offers similar support. While this approach was very efficient and target-oriented, it has one important drawback: in general it will not find crosscutting concerns which are not listed in the used catalog. However, for a refactoring project, this is nevertheless a good strategy to start. We reported our experience with this approach of aspect mining in comparison to a related project using available aspect mining approaches in [17].

Uli Eibauer [4] refactored version 1.8.0.2 of HSQLDB as part of his master thesis. He identified and refactored several standard aspects guided by Laddad's book, and in the process

also manually identified some application specific crosscutting implementations in the code. We recorded and tagged 17 (not including the initial Java version) modified version before and after each major refactoring step, including versions with failing tests. To get a better understanding of the above versions we will briefly discuss the concerns refactored in this system in the following, starting with well-known standard example logging.

7.2.1 Refactoring Logging

We explicitly distinguish between *logging* and *tracing*. Displaying information for operations performed is called logging. Tracing however is a special case of logging, where messages are displayed whenever entering or leaving a method, thereby potentially also displaying parameter values. The main purpose of tracing is to get an overview of the actual control flow. Logging in contrast offers considerably more detailed information. Laddad [11] actually calls tracing logging in his book, we will however discuss actual fine grained logging in this section. Note that logging is frequently named as a standard example of a crosscutting concern. However in general people most likely refer to tracing, as refactoring detailed, location specific logging in general is non-trivial.

Analysis of `HSQLDB` actually showed three 'logging' concerns: the database logging mechanism, servlet logging, and the actually debug-logging we refer to in this section. Log statements in the code in general call method `printSystemOut` of class `org.hsqldb.Trace`. However, the code also contains some plain `System.out.println` statements which (erroneously?) do not use the tracing class.

Analyzing the log code showed that log statements were used in various different contexts, resulting in a large set of specific pointcuts and also specific pieces of advice referencing them. This is especially problematic if individual character strings are printed to the log—in these cases either a particular piece of advice has to be formulated to handle the respective joinpoint (including the hard-coded string information) or this specific information has to be skipped, i.e. a quantified implementation is not possible with AspectJ.

Another problem is that logged context is often not accessible for AspectJ, as log statements access local variables or—as seen above—hard-coded literals. To access such values in some cases a preceding object-oriented refactoring can help. For example we could create new methods to make previously local variables accessible as method parameters or even promote local variables to fields. Note however that—while this technically might solve the problem—this can be a very bad solution from a software engineering point of view. The restructuring of code with the only purpose to access otherwise unaccessible values using aspects has been called the *arranged pattern problem* in [6].

To summarize, refactoring general logging was problematic for `HSQLDB`. The problems we encountered actually prohibited a semantically equivalent aspect-oriented logging implementation. In particular, the software quality is not promoted by refactoring logging. Crosscutting concerns like logging have been called *heterogeneous aspects* [3] as they in general have joinpoint specific behavior, although the underlying concern is crosscutting. Aspects which apply the same behavior to multiple joinpoints in contrast are called *homogeneous aspect*. It is currently cumbersome to write heterogeneous aspects using available aspect languages, while homogeneous aspects can often be formulated easily.

7.2.2 Refactoring Tracing and Profiling

Tracing and Profiling are closely related to Logging as they can be seen as special cases of the more general logging discussed above. Tracing only logs method entry and exit events (including parameter values); profiling logs time stamps for these events to measure the time spent in a method body.

The original implementation mixed up logging and tracing, both concerns used the same mechanism. For our aspect-oriented refactoring we did not separate tracing and logging either. In contrast to the above mentioned specific log statements, implementation of tracing was however easily possible with a very general piece of `before`-advice (and `after`-advice, respectively).

We also found a profiling mechanism in HSQLDB, which used a `StopWatch` object to record and aggregate method runtime. Refactoring the profiling concern was—similarly to tracing—very easy, apart from two exceptions. The original implementation also offered to take interim times. The respective calls are however located in the middle of methods, and were thus not always accessible via an aspect. This also involved for example measuring loop execution time. We used similar techniques as in the case of logging to access some relevant joinpoints, if acceptable from a software engineering point of view. A second problem was that `HSQLDB` also used nested profiling, i.e. two different profilers to measure runtime on a more fine grained level than methods. Refactoring nested profiling could be solved by using specific pointcuts for nested profilers, which only selected few joinpoints.

To summarize, we again saw that general tracing and logging behavior was easy to refactor, however as soon as joinpoint-specific behavior had to be implemented or we needed a granularity below the method level we either had the problem that relevant joinpoints and/or context were not accessible or that we had to write joinpoint specific advice.

7.2.3 Policy Enforcement and Pre-/Postcondition Checks

Policy Enforcement refers to the automatic enforcement of design guidelines or calling conventions. AspectJ offers language constructs `declare warning` and `declare error` to make such guidelines part of a program. Violation of these guidelines then results in a compiler warning or even a compiler error. Traditional approaches to design guidelines are mostly based on providing adequate documentation, which are then checked by for example code reviews. These rules in a scarcely defined in the code itself let alone automatically checked by the compiler.

Due to the above, the `HSQLDB` code base did not contain any classes referring to programming guidelines, i.e. we could not find any scattered implementations of a policy enforcement concern. By chance we however found a comment describing that stored procedures in the procedure library should not call overloaded or inherited public static methods. However, as AspectJ currently does not contain language constructs to express *inherited* or *overloaded* we were not able to formulate a respective policy enforcement statement. In other words, AspectJ is not expressive enough to state complex design rules which cannot be formulated by using asserts.

However, although we could not formulate and thus enforce the above described global design policy by using aspects, we noted that the `HSQLDB` code base contained several locations where `check*`-methods were called (similarly to the authorization checks described below). These methods check conditions which have to hold in order to execute a certain operation, i.e. they can be interpreted as a check of complex preconditions. If such a condition is violated, most of the time an exception is thrown.

We were able to extract the calls to these check methods and move them to aspects, although we again ran into problems with unaccessible joinpoints. The code to work around these problems resulted in a cumbersome and hard to understand implementation, so that we actually prefer the original crosscutting implementation.

7.2.4 Refactoring Pooling and Caching

Both *pooling* and *caching* provide a mechanism to reuse objects which are expensive to create and delete. These two concerns are thus often classified as optimization concerns. In Laddad's

book pooling and caching are distinguished by the access pattern to objects stored in the cache. In contrast to caching, pooling explicitly asserts exclusive access to the respective object.

Note that the caching concern described in this section had to be distinguished from the database caching core concern. The latter tries to optimize access to user data retrieved frequently from the database, but is not concerned with program internal data like the caching we refactored as an aspect.

In the `HSQLDB` code base, several modules explicitly called a cache class to reuse immutable objects like instances of `Integer`. Aspectizing this cache is straightforward, by intercepting constructor calls to cachable objects by `around`-advice and instead returning a cached object if available by using the existing cache implementation.

Caching is a classical homogeneous crosscutting concern. It is applied to several joinpoints, and at each joinpoint exactly the same behavior is executed. Refactoring such concerns is in general straightforward using AspectJ. An interesting side-effect of our refactoring is that we discovered several places where the cache can be used but has not been used before. We also put those locations under caching control. Note that this only required a minor pointcut modification instead of invasive system changes compared to a traditional approach. For caching the aspect-oriented implementation is clearly superior compared to the traditional approach.

7.2.5 Refactoring the Worker Object Creation Pattern

A *Worker Object* or *Function Object* is an instance of a class which only defines a single method (often called `run()`). The purpose of such objects is to allow to pass a certain functionality as a parameter object (see for example the `Comparator` interface in the Java API) or to execute a task in a different thread of control. The *worker object creation pattern* is used to remove the explicit construction of such worker objects from the code. Instead a piece of `around`-advice is created which transparently wraps calls to a method to be executed in a worker object and then passes this object to, for example, another executing thread.

In the `HSQLDB` code base worker objects are frequently used to execute time consuming operations in a separate thread. As refactoring all these constructions is a cumbersome task (the original worker object creation, object passing and execution has to be removed from the code, often resulting in some side effects), we chose to only refactor *swing thread safety* as an example.

GUIs written in Java Swing require that only the event dispatching thread modifies visible components to work properly. If this rule is violated, race conditions like lost updates, missing repaints and even deadlocks can occur if no proper manual synchronization is provided. This means that any other thread wishing to modify the GUI has to create a worker object encapsulating the modification operation and pass this object to the event dispatcher thread. Analysis of the source code of `HSQLDB` uncovered several violations and misuses of the above described technique. We unified and fixed the implementation by removing all explicit worker object creation code from the system. Instead we wrote pointcuts and advice which intercept calls to GUI component mutators and wrapped them in worker objects using `around`-advice.

Compared to the plain Java implementation the aspect-oriented version has the advantage that it enforces Swing's single thread rule. However there is also a price to pay, as now—even if no spurious pointcut matches occur—each single call to GUI mutators is intercepted and run as a separate worker object, which might be far too fine grained. For example consider a method changing appearance of several widgets. A worker object is now created for each single widget update, instead of one worker object for all updates.

7.2.6 Exception Introduction Pattern: Refactoring Exception Handling

Exception handling has been proposed as a fruitful application of aspects, especially if a common error handling policy can be found throughout the system. A very popular candidate here is the famous *log and forget* error "handling" policy.

Analysis of the HSQLDB code base showed that also in this case, log and forget has been used frequently—in total we found 157 handlers where the exception was either only logged or even thrown away without logging. We could also group many of the remaining exception handlers into seven other categories.

To refactor exception handling as an aspect, we removed the common original exception handlers and instead intercepted exception handler execution using the `handler` pointcut designator. While this implementation is again straightforward, we again have to emphasize that this technique was only applicable for very general exception handlers—or, in other words, if they could be implemented as homogeneous aspects.

As soon as a certain handler performs more complex joinpoints specific exception handling, we experienced the same problems as outlined for general logging. Another important drawback are `finally`-blocks, as AspectJ currently does not offer a primitive to also intercept execution of statements contained in finally blocks. We also found the resulting code hard to read, if the actual error handler did something else than log and forget as the specific handling strategy is now hidden in the aspect. In general we have to state that aspect-oriented exception handling has to be applied with considerable caution in order not to reduce readability of a given system. These results are in line with the result previously published by others [14].

7.2.7 Refactoring Authorization

Laddad's book lists *authorization* and *authentication* as standard examples where aspects can be beneficial. Although Laddad analyzes an aspect-oriented implementation for systems using the Java Authentication and Authorization Service (JAAS), his analysis is applicable in a broader context.

While JAAS is not used in HSQLDB, the implementation of authorization is nevertheless crosscutting in its code base. Usually authorization is tested against the current `Session`-object which in turn passes the request to the respective `User`-object which finally delegates the request to a `Grantee`-object associated with each user.

We refactored the authorization concern by removing these direct invocation on the `Session`-object and instead intercepted the relevant calls to authorized operations using customized `around`-advice. The advice first evaluates if the current user is authorized to accomplish a certain operation by accessing the respective `Grantee`-object. If the operation is permitted, execution proceeds normally, otherwise access is denied.

While we did not encounter major problems while refactoring authorization in this case study, we noted that calls relevant for authorization were encapsulated in only two classes in the HSQLDB code base, i.e. the authorization concern was already encapsulated and did not cut across major parts of the system. The benefit of the aspect-oriented implementation is thus limited.

7.2.8 Refactoring Trigger Firing

Due to the increased familiarity with the code of HSQLDB we noticed that several operation in the code at the end activated database trigger execution. As these execution statements were scattered in the implementation of class `Table`, we also extracted them in an inner aspect `AspectTriggerFiring` of that class.

The refactoring was mostly unproblematic, the inner aspect is easy to understand and straightforward. The only problem was the trigger firing context (e.g.

`Trigger.INSERT_BEFORE_ROW`), which was hard-coded in the respective trigger call. We solved this problem by creating several pieces of advice, which was acceptable due to the limited number of relevant constants. Thus we can say that the refactoring of trigger activation was a success.

7.2.9 Summary & Conclusions

In this section we briefly described which crosscutting concerns were found during the analysis of `HSQLDB`, and if we were able to actually refactor these crosscutting implementations into aspects. To find the crosscutting implementations of known crosscutting concerns, we used a semantics guided approach. Using a catalog of well-known aspects we located related classes and then used the concern mining tool FEAT to find relevant referencing code. This approach was very successful for our case study, as it clearly did not produce any false positives.

Note however that we were not always successful in actually refactoring crosscutting implementations. This is due to (i) language limitations and (ii) heterogeneous aspects. Using AspectJ it was not always possible to access necessary context and joinpoints needed for a specific refactoring. While traditional refactoring can be used to make some of the joinpoints including their context accessible, from a software engineering point of view the resulting necessary code modification often had to be dismissed.

The second problem is a more semantical problem. While there is an underlying crosscutting concern—for example logging—the actions to perform at different joinpoints are not homogeneous. Instead each joinpoint has to be augmented with a specific piece of advice. In these cases, the aspect-oriented implementation has no advantages compared to the plain Java implementation, and the code is often bloated and less readable.

For those cases where we refactored a homogeneous crosscutting concern—like for example the swing worker thread creation or caching—refactoring in general was straightforward and the resulting implementation also superior compared to the plain Java implementation.

We also tried to find implementation of design patterns in the `HSQLDB` code base, as [7] suggests aspect-oriented implementations of the design patterns introduced by [5], and claimed that several of these implementations were superior compared to the object-oriented variants. Unfortunately—even with the help of the project administrator—we only found four implementations of irrelevant design patterns in the code of `HSQLDB`. We thus can not report any experience about refactoring of design pattern implementations.

In this thesis we only give a brief summary of the `HSQLDB` refactoring project. For details on the manual aspect mining approach, the actual refactorings we did (including code examples), and also a evaluation of the code quality after the aspect-oriented refactoring we refer the reader to [17] and Ulrich Eibauers master thesis [4]. The different aspect-oriented version of `HSQLDB` are available at `https://sourceforge.net/projects/ajhsqldb`.

Although the case study of refactoring `HSQLDB` in itself was very interesting, the main purpose for us was nevertheless to create a software system we could use as a case study for the analysis techniques described in the previous sections. This is however not unproblematic. As `HSQLDB` is a system which has been refactored by ourselves—although we did not intend it and also did our best to avoid it by assigning different people to work on tools and refactoring—it is possible that the results of this self-created case study are biased. While this is a threat to validity, we think that the available code base of `HSQLDB` is nevertheless a very interesting subject to study.

7.3 Conclusions

As can be seen, the number of available subjects to study is rather limited. For the work presented in this thesis, this lack of code was a major problem when evaluating our approach.

From the discussion above it can be seen that neither the AspectJ examples, nor `AJHotDraw` are good subjects to study as they only comprise few, in part trivial aspects. The *abc* test suite provides some interesting examples, but only comprises programs with few thousand lines. A development history for multi-version analysis is also missing. Fortunately at least `AJHotDraw` is associated with a test suite, and for the *abc*-examples test cases can be constructed relatively easy. The auction system can be easily augmented with a (JUnit) test suite, but is only available in a single verion.

There are also few systems not explicitly discussed here, resulting from published case studies (see Chapter 2). However, these systems either use old AspectJ versions or—with few exceptions—are not publicly available.

We also tried to set up a public repository for AspectJ programs, to collect further subjects to study. We did so by asking for code on the AspectJ users mailing list. We actually got two (2) answers: (i) the *abc* test suite and, (ii) "Thanks, very good idea, this is necessary!". In total this was rather disappointing and (mostly) futile effort.

Due to this lack of code to evaluate our methods on, we started the `HSQLDB` refactoring project. We tried to perform this refactoring project without any special provisions for our analysis tools. Nevertheless we agree that it can be discussed if case studies on own code are valid. However this is the best we can offer. The `HSQLDB` system is of reasonable size, a development history is available (although only a *refactoring history*), and the system is also associated with a test suite.

The lack of publicly available AspectJ code is actually surprising, if we consider the number of publications related to aspect-oriented programming. The first (stable) version of AspectJ was already published in 2001—five years ago. Meanwhile, for the paper discussing the languages [10] *CiteSeer* lists 219 citations.[11] While we do not expect availability of lots of large systems with a project history of several years, it seems that the availability of at least some non-trivial AspectJ projects could

- help to demonstrate and teach aspect-oriented concepts,
- allow experiments with aspects in practice,
- develop a design methodology for aspects,
- identify problems of this approach, and develop methods to counter them,
- and last but not least also allow to evaluate tools, like the tools resulting from the work presented in this thesis.

In our opinion, this is—beside the problems outlined and addressed in this thesis—the major challenge for aspect-orientation today. With the refactoring project introduced above we addressed this problem, although—as a refactoring approach—gained experience is limited and using these systems to evaluate our own techniques can be criticized. However, given the lack of other publicly available systems, this seemed to be the best option to evaluate the techniques introduced here.

[11] `http://citeseer.ist.psu.edu/kiczales01overview.html`

Bibliography

[1] BREU, S., AND KRINKE, J. Aspect Mining using Event Traces. In *19th International Conference on Automated Software Engineering, 2004. Proceedings.* (Linz, Austria, September 2004), pp. 310–315.

[2] BREU, S., ZIMMERMANN, T., AND LINDIG, C. Mining Eclipse for Cross-Cutting Concerns. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories* (New York, NY, USA, 2006), ACM Press, pp. 94–97.

[3] COLYER, A., AND CLEMENT, A. Large-scale AOSD for Middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development* (New York, NY, USA, 2004), ACM Press, pp. 56–65.

[4] EIBAUER, U. Studying the Effects of Aspect-Oriented Refactoring on Software Quality using HSQLDB as Example. Diplomarbeit (Master Thesis), Universität Passau, FMI, Passau, Germany, Innstraße 32, 94032 Passau, August 2006.

[5] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] GYBELS, K., AND BRICHAU, J. Arranging Language Features for more robust Pattern-based Crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 60–69.

[7] HANNEMANN, J., AND KICZALES, G. Design Pattern Implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM Press, pp. 161–173.

[8] HASSOUN, Y., JOHNSON, R., AND COUNSELL, S. A Dynamic Runtime Coupling Metric for Meta-level Architectures. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* (March 2004), IEEE, pp. 339–346.

[9] JANZEN, D., AND VOLDER, K. D. Navigating and Querying Code Without Getting Lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 178–187.

[10] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An Overview of AspectJ. *Lecture Notes in Computer Science (LNCS) 2072* (2001), 327–355.

[11] LADDAD, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[12] LIEBERHERR, K., HOLLAND, I., AND RIEL, A. Object-Oriented Programming: an Objective Sense of Style. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1988), ACM Press, pp. 323–334.

[13] LIEBERHERR, K., LORENZ, D. H., AND WU, P. A case for statically executable advice: checking the law of demeter with AspectJ. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM Press, pp. 40–49.

[14] LIPPERT, M., AND LOPES, C. V. A Study on Exception Detecton and Handling using Aspect-Oriented Programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA, 2000), ACM Press, pp. 418–427.

[15] ROBILLARD, M. P., AND MURPHY, G. C. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM Press, pp. 406–416.

[16] SHEPHERD, D., PALM, J., POLLOCK, L., AND CHU-CARROLL, M. Timna: a Framework for Automatically Combining Aspect Mining Analyses. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering* (New York, NY, USA, 2005), ACM Press, pp. 184–193.

[17] STOERZER, M., EIBAUER, U., AND SCHOEFFMANN, S. Aspect Mining for Aspect Refactoring: An Experience Report. In *Proceedings of First International Workshop 'Towards Evaluation of Aspect Mining' TEAM'06*. Silvia Breu, Leon Moonen, Magiel Bruntink and Jens Krinke, 2006, pp. 17–20. Held in conjunction with European Conference on Aspect-Oriented Programming ECOOP'06.

[18] VAN DEURSEN, A., MARTIN, M., AND MOONEN, L. AJHotDraw: A Showcase for Refactoring to Aspects. In *In Proceedings of AOSD'05 Workshop on Linking Aspect Technology and Evolution* (2005).

[19] ZHANG, C., AND JACOBSEN, H.-A. PRISM is Research in Aspect Mining. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 20–21.

8 Conclusions and Final Remarks

This final chapter summarizes the work discussed in previous chapters and assesses aspect-oriented ideas and in particular AspectJ in the context of the problems which became apparent while working on this thesis.

8.1 Summary

In this thesis Chapter 1 started with a discussion of the term "Separation of Concerns" before introducing the basic ideas of aspect-orientation in general and of the most popular aspect-oriented language, AspectJ, in particular.

Chapter 2 examined published case studies of aspect-oriented programming (all of them done with (in part outdated versions of) AspectJ) and extracted several problems the authors of these case studies reported. Based on these reports we analyzed the effects of aspect-oriented programming on code quality criteria and formulated four important problems of aspect-orientation (as implemented with AspectJ) today. The remaining of this chapter then reviewed approaches of other researchers addressing these problems on the language level.

In this thesis we proposed tool support to deal with the problems of aspect-orientation we identified in Chapter 2. Chapter 3 first discussed implications of static crosscutting, derived a non-interference criterion and also showed how this criterion can be implemented. Chapter 4 then presents the change impact analysis technique of *Chianti* in the context of Java. We show how this technique can be used to detect failure inducing changes as well as to early release a subset of committable changes. We then finally discuss how we used this technique for impact analysis of aspects and how this technique can also be extended to provide complete *Chianti*-style change impact analysis for aspect-oriented programs. Lifting *Chianti*-style change impact analysis also allows the application of the related debugging support and the early change release technique for aspect-oriented software.

Chapters 5 and 6 finally present static analysis techniques. Chapter 5 introduces pointcut delta analysis, a technique to deal with the fragile pointcut problem. This technique (re)uses atomic changes to explain pointcut deltas. Chapter 6 finally deals with aspect-interference, or rather the special case of precedence related advice conflicts. This chapter proposes an analysis of both control and data flow within advice to decide conservatively whether two pieces of advice commute.

Chapter 7 finally gives an overview of the available aspect-oriented systems we used to evaluate the methods we proposed here. In this chapter we also discuss suitability of the different systems for static and dynamic analysis as well as multi version analysis.

We summarize the contributions of this thesis as follows:

1. This thesis critically reviewed available case studies and collected important problems the authors of these case studies reported in their papers. We also did a fundamental study of features of AspectJ and outlined how use of aspects can have subtle unintended side effects. The identified problems thus give a good overview of deficiencies of AspectJ and in part of aspect-orientation in general.

2. We reviewed several approaches proposed by other researchers to address the problems identified before. In our view, most problems still remain unsolved. Some problems like aspect interference have even been mostly neglected in literature until recently.

3. For each major problem we identified, this thesis proposes a tool guided approach to detect potential unwanted effects and thus help programmers using AspectJ. To implement these tools, we proposed, applied, and in part also adapted available methods of static and dynamic program analysis. In particular we addressed the following topics:

   Static Crosscutting: This thesis showed that inter type declarations can change system semantics and provides a non-interference criterion to guarantee semantic preservation in the presence of inter type declarations.

   Change Impact Analysis: This thesis discussed how the change impact analysis of *Chianti* can be used to support programmers in finding bugs and also in releasing a subset of their code edits to a repository without breaking tests for both Java and AspectJ.

   System Evolution: Not only do we describe the fragile pointcut problem, but we also state that a general solution to it might not be possible at all, as the problem is a direct consequence of the quantification property of aspect-orientation. The pointcut delta analysis we propose however allows programmers to make informed decisions when unexpected changes in the set of adapted joinpoints occur.

   Aspect Interference: The problem of aspect interference has been mostly neglected in research, although it is an important problem considerably threatening the scalability of aspect-oriented programming. Although we do not provide a complete solution, our interference analysis at least allows to automatically detect semantical changes due to undefined aspect precedence.

   Refactoring `HSQLDB`: We finally addressed the lack of publicly available AspectJ programs by refactoring the Java Open Source system `HSQLDB`. The results of this project are valuable in three ways: (i) we successfully used a semantics-guided approach to aspect mining, (ii) we gathered interesting insight in the suitability of AspectJ to implement crosscutting concerns, and (iii) we finally produced a system we could use as a case study to evaluate our tools with.

   Most of these tools (with the exception of the complete multi-version change impact analysis and some deficiencies in the implementation of the aspect interference analysis) are implemented and evaluated. This thesis also contains data from case studies with available AspectJ code to assess the usefulness of these tools by example.

8.2 Aspects and `ComeFrom`

Before summarizing the critics on AOP in a more serious way, in this section we will recapitulate an interesting comparison of aspects with the famous `GoTo` statement [8]. `GoTo` is a

statement which easily results in unreadable code, as GoTo does not provide structured control flow. Today, this view on GoTo is widely accepted.

Back in the late 60ties this was however not the case. In 1968 Edsger W. Dijkstra wrote a famous article "go to statement considered harmful" [4] in which he analyzed the GoTo statement with regards to its effects on program understanding. He argued that, in oder to support comprehensibility, each language construct in a programming language should maintain a *unique coordinate system*, allowing programmers to map variables to values at each point of a program.

In his article, he analyzes statements, loops and procedure calls, and argues that all these programming language constructs maintain such a coordinate system, as the programmer—maybe with some simple program intrinsic information like a loop counter or a call stack—is in general able to map variable names to values by reading the code. Dijkstra continues by examining the GoTo statement. He argues that, as soon as GoTo is added to a language, this coordinate system is lost, as it is no longer possible to deduce from a certain source code location where control flow actually comes from, let alone the values variables have at that point.

Based on this analysis, Dijkstra claims that GoTo should be banished. Time showed he was right. If we look at modern programming languages, many of them do not contain a GoTo statement. And even if such a statement is present, it is rarely used as it is in general considered to be bad programming style.

This article of Dijkstra got great attention during the discussion around GoTo elimination back then. During this discussion, several other approaches have been suggested aiming at the elimination of GoTo. One of them was the suggestion of a ComeFrom statement, as an inverse GoTo [3]. Consider the following example:

```
1  int i=1;
2  i = i*2;
3  print i;
4
5  ComeFrom 2;
6  i = i * i;
7  print i;
```

This program would start with initializing i with 1, the multiplying it with 2, and then the ComeFrom-statement would transfer flow of control to line 6, where i is multiplied with itself, before the program finally prints 4 and exists.

While ComeFrom was an April's Fool joke and not meant seriously, there is an interesting parallel of ComeFrom and advice. We will formulate the above example a bit different, to make this obvious.

```
1  class C {
2    int i=1;
3    void foo() {
4      i = i*2;
5      print(i);
6    }
7  }
8  aspect A {
9    void around(int i): call(* print(int))  args(int) {
10     i = i * i;
11     print(i);
12   }
13 }
```

The above code is of course legal AspectJ code. And we will now compare the `around`-advice to the `ComeFrom`-statement discussed above. Similarly to `ComeFrom`, advice transfers control flow to a different location without showing any hint of this changed control flow in the advised code (see loss of modular reasoning). Of course `ComeFrom` is a statement even worse than `GoTo`. If `GoTo` already destroys the coordinate system Dijkstra described, the `ComeFrom` completely turns this coordinate system upside down. Nobody would really suggest to add a `ComeFrom`-statement to serious programming language[1], however advice has tremendous success. Interesting.

To be fair, there is a difference of advice and `ComeFrom`—advice in general returns to the location where the control flow has been hijacked. This difference is important. Nevertheless, the similarity is there and should be considered by researchers and practitioners.

Note that this section is a summary of a position paper [8] written for a panel discussion, and was intended to be provocative. It is interesting to note that this position paper generated most feedback of all publications of the author of this thesis.

8.3 An Assessment of Aspect-Orientation

In this thesis, we discussed several problems of aspect-orientation as we see it today. While we admit that the main focus of this analysis is on AspectJ and there might be other aspect-oriented languages not suffering from one or the other particular problem, we claim that in general aspect-oriented languages suffer from the problems addressed in this thesis, in particular

Aspect-Base Coupling: Current pointcut languages in general *couple* aspects to base programs. While research suggested to define more abstract pointcut languages to avoid this coupling, the discussion in Section 5.1.2 shows that this desired goal might be impossible to reach. If this is true, the aspect-base coupling is an inherent problem of aspect-orientation.

Loss of Modular Reasoning for Base Modules: While some researchers state that the problem (i) has already been present before, (ii) does not exist as aspects only add behavior (which is not true) or (iii) is irrelevant as the "evil aspect"[2] most likely will not be part of the system and consequently neglect the problem, the discussion of related work in Section 2.3.1 shows that this problem is valid and has been recognized by other researchers as well.

Although "complete" modular reasoning might not be possible for object-oriented languages as well due to dynamic dispatch, today it is accepted that overriding methods should not have "surprising effects". Such accepted guidelines for aspect usage are still missing today, and thus programmers in part fear aspect-oriented programs as the code they read potentially has nothing to do with what is actually executed, a true nightmare when looking for errors.

Indeed the AspectJ Development Tools *ajdt* are a direct reaction to the loss of modular reasoning as they make aspect influence (on a syntactic level) visible in (base) modules to alert programmers that module semantics are potentially externally modified. Thus tool-supported approaches have a good tradition for AspectJ. Approaches reported in literature to deal with this problem on the language level up to now are either incomplete, considerably restrict the power of aspects, are in contrast to the obliviousness property of aspect-orientation or even multiple of the above.

[1] Serious because of INTERCAL: `http://www.muppetlabs.com/~breadbox/intercal/home.html` or `http://www.catb.org/~esr/intercal/`. Another provocative question: is INTERCAL due to `ComeFrom` an aspect-oriented language?

[2] As author of this thesis I do not fear the "evil aspect"—I rather fear the "brain dead aspect".

Aspect-Interference: A major problem in our view is also aspect interference. It is alarming that only few research papers up to now even address the problem, although it is obviously relevant. Only recently an ECOOP workshop addressed the topic.[3] While our tool-based approach might lighten the problem, this is a general deficiency of most aspect-oriented languages comparable to AspectJ, which is not resolved up to now.

Evolution Problems: The inherent aspect-base coupling and the loss of modular reasoning finally have massive impact on the evolvability of aspect-oriented systems. Note that we completely agree that evolvability of crosscutting concerns—i.e. of the functionality defined by aspects—is considerably improved by aspect-orientation. However it is also important to note that aspects—or better their specific applications—are fragile, as we discussed in Chapter 5, and—even worse—that due to this fragility system semantics can be silently altered.

In this thesis we applied techniques of program analysis, more specifically of change impact analysis, to detect potential problems stemming from these language deficiencies. While the resulting AOPA tool suite can detect and thus lighten several of the problems discussed above, the more interesting question in this context is:

> Do we really need/want a language which can only be used safely with a massive tool suite?

While we completely agree that software tools can considerably improve productivity and thus should be used by programmers, we think that a dependency on *lint*-like tools should nevertheless be avoided. Note that the above statement does not at all attack the benefit of tools like *lint* or our own work. The methods we introduced are—even if aspect-orientation may turn out to be a dead end—definitely relevant for software written in currently available aspect-oriented languages. Although only very few systems are publicly available at the moment, we nevertheless believe that an increasing body of AspectJ programs exists.

While the discussion in this thesis focuses on language deficiencies in part specific for AspectJ, other researchers addressed other fundamental semantical problems of aspect-oriented languages and aspect-orientation in general. For example [2, 5] discusses semantical problems of aspects which address themselves. [7] discusses several of the deficiencies we also addressed in this thesis and states that these problems cannot be solved on the language level without ending up with a non-aspect language.

Beside the critique stated above, another fundamental problem has to be addressed by the aspect community. While there is a wealth of different languages, frameworks, and aspect-oriented methods, there are only very few systems using these languages, frameworks or methods publicly available today. And if such systems are available, they often tend to be toy examples. For the programming language community it is customary to write the compiler in the new language as proof of concept. But for aspect-oriented systems even this standard example is missing. So what we currently see is a lot of new ideas, which are published at several major conferences but are unfortunately often not sufficiently evaluated. This lack of available code on the one hand hinders interested people to learn aspect-oriented programming, on the other hand it raises doubts if the proposed ideas fulfill their promises and actually scale.

While new ideas are important and should not be suppressed, in our opinion it is however at least as important to answer another question:

> Does it work?

We think, after 9 years (referring to the 1997 paper of Kiczales), it is time to answer this question by providing larger, preferably long term case studies. Up to now the largest aspect-oriented program we have seen (in terms of lines of aspect code[4]) is the `ProdLine` program

[3] ADI, Aspects, Dependencies and Interactions, held in conjunction with ECOOP'06.

[4] The size of the modified base application seems less relevant in this context.

with less than 1200 lines of aspect code. And this program best of our knowledge is not actively developed.

Last but not least we also address a fundamental problem in the way how programming languages should be built. When creating Java, Sun tried to completely define the semantics of the language. Even ridiculous constructs in Java still have a defined semantics. For AspectJ however, as an extension of Java, this is no longer true. We give two examples:

Advice Precedence: As discussed in Chapter 6, *advice precedence can be undefined*, resulting in undefined system semantics as advice in general is not commutative.

Pointcut Evaluation: AspectJ's pointcut language knows an `if`-operator, which can call any Java method returning a `boolean` value. Obviously such methods can have side effects. For AspectJ pointcuts with multiple `if` pointcut designators however it is neither defined *in which order* nor *how often* nor *when* these statements are executed.[5]

We have to emphasize that we do not talk about semantics of ridiculous constructs like `i+++i++` or `f(i++, ++i)` in this context (these are defined in Java!), but about expressions which can show up in every day programs. We thus think that AspectJ—at least in this context—is a step backward. While this is not a fundamental critics of aspect-orientation in general, it is a critique of the most popular language AspectJ. Other languages however may suffer from similar loopholes in language semantics.

To summarize, aspect-orientation as we see it today and AspectJ in particular suffer from several important problems which seem to be—at least in part—inherent problems of this approach to implement software systems. Thus in our opinion, before aspect-orientation can be recommended as the successor of object-orientation, the above problems have to be solved and the adequacy of the provided solutions has to be sufficiently evaluated. In its current state, aspect-orientation might be a useful tool for development aspects like tracing, profiling or design rule enforcement (which are valid applications!) and also for small toy programs, but if it really fulfills its promises still has to be shown. Note that we do not attack that aspect-orientation tries to solve a valid problem—we completely agree on this—but we state that the current solution is not yet satisfactory.

8.4 Conclusions

What we have seen in this thesis is that aspect-oriented programming in its current form and especially AspectJ does not have the potential to replace any other programming paradigm. We discussed several serious problems—consider for example aspect interference, fragile pointcuts, the lack of evaluation and also the lack of a defined semantics for several constructs—which have to be solved before aspect-orientation can be recommended without hesitation.

Due to the wealth of problems, aspect-oriented programming seems to be a perfect candidate to construct programmer support tools. However, as stated above, it seems that solutions on the language level should better avoid outlined problems up front instead of fixing them afterward. While our analysis techniques are an appropriate way to deal with these problems for existing systems, for new systems we rather recommend to use a language which does not suffer from them.

In this thesis we addressed deficiencies of AspectJ by using program analysis techniques to build supporting tools. This effort is in the spirit of code checking tools like *lint* for the C

[5] I.e. not defined but by the compiler. Cited from the AspectJ semantics appendix: "Note that the order of evaluation for pointcut expression components at a join point is undefined. Writing if pointcuts that have side-effects is considered bad style and may also lead to potentially confusing or even changing behavior with regard to when or if the test code will run."

language or *CheckStyle* for Java, although the analysis techniques presented here are considerably more powerful and explicitly address system evolution.

In summary these techniques show that tool support in general and change impact analysis based tools in particular can be used to (semi)automatically detect potential problems in the source code. In other words, appropriate tool support can considerably improve programmer productivity and also increase code quality. As we used techniques of program analysis, this thesis also once again demonstrates that program analysis can provide important support for programmers considering failure detection and program understanding.

Together all the techniques introduced in this thesis form the AOPA tool suite. We think that such tools can considerably support programmers in avoiding and finding problematic code before a problem occurs "in the wild". A natural question is now if these tools actually work in practice. To really answer this question, a user study would be necessary.

However, here our evaluation faced an important problem, as we did not find a group of programmers using AspectJ for a non-trivial project. It turned out that it is even very hard to find existing AspectJ code to test our tools with. Chapter 7 gives an overview of the subjects we finally studied. Unfortunately the list is (i) short and (ii) only contains two non-trivial examples: `AJHotDraw` and `HSQLDB`. The latter is even a system which has been refactored in our group, specifically to have some code to analyze (although we did not take any influence on the development with regard to our tools). So we admit that our evaluation can also be improved. This however requires the availability of more multi-version open source AspectJ applications. The lack of available non-trivial systems in itself however is also an interesting finding.

Some techniques we presented in this thesis are specific for aspect-orientation (like the pointcut delta analysis presented in Chapter 5) or even AspectJ (advice interference, Chapter 6). But several of the proposed techniques are also applicable in a wider context. For example the techniques presented in Chapter 3 can be used for impact analysis in the Java context as well as to estimate the impact of adding new method and fields or changing the class hierarchy (this corresponds to the lookup changes of *Chianti*, although field lookup changes are not covered there). Chapter 4 explicitly discusses application of change impact analysis techniques in the context of Java and then shows how they can be applied to AspectJ programs.

8.5 Looking Ahead

While this thesis presents important results, there is certainly room for improvements of the AOPA tool suite. For example a first improvement is to unify the analysis techniques presented in Section 4.5 and Chapter 5 to actually implement a complete *Chianti*-style change impact analysis for AspectJ. We outlined how this can be done theoretically, but currently we still have two separate tools.

Second, the advice interference analysis currently only works for a subset of AspectJ and only uses a rather crude pointer analysis. We also demonstrated its benefit for the `Telecom` example only. Switching to a more precise pointer analysis variant and extended case studies including a study of the achieved precision is necessary.

This thesis would benefit from more and especially more interesting case studies (i.e. large programs with several non-trivial aspects) in general. This is especially true for a user study. However this is considerably hindered by the lack of available code and programmers.

Another interesting research direction could be to explore how the outlined problems can be avoided by modifying the language or even using different approaches. In particular, it is important to examine if the core concept of aspect-orientation—quantification—is in tension with modular reasoning and system evolution in general. Findings of some researchers (see Chapter 2.3) and also our own finding suggest that this might be the case. If this actually turns out to be true, this might considerably question if the claims made by aspect-orientation can

be fulfilled.

Feature Oriented Programming [1] as well as Multidimensional Separation of Concerns [6, 9] for example both attack the same problem as aspect-orientation, but in principle do not suffer from several problems we described in this thesis.

Although we end this thesis here with these final remarks, we are sure that aspect-orientation will still be a research topic for several years. Hopefully this thesis is also a contribution to avoid or solve several of the problems we see today, to allow language designers to provide better languages and programming paradigms in the future.

The End.

Bibliography

[1] BATORY, D. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 702–703.

[2] BODDEN, E., FORSTER, F., AND STEIMANN, F. Avoiding Infinite Recursion with Stratified Aspects. In *Net.ObjectDays 2006* (2006).

[3] CLARK, R. L. A linguistic contribution of goto-less programming. *DATAMATION* (December 1973).

[4] DIJKSTRA, E. W. Letters to the editor: go to statement considered harmful. *Commun. ACM 11*, 3 (1968), 147–148.

[5] FORSTER, F., AND STEIMANN, F. Aop and the Antinomy of the Liar. In *Foundations of Aspect-Oriented Languages Workshop at AOSD 2006* (2006), C. Clifton, R. Lämmel, and G. T. Leavans, Eds., pp. 47–56.

[6] HARRISON, W., AND OSSHER., H. Subject-Oriented Programming (a Critique of pure Objects). In *OOPSLA* (1993), pp. 411–428.

[7] STEINMANN, F. The Paradoxical Success of Aspect-Oriented Programming. In *OOPSLA 2006, Proceedings: Essays* (Portland, Oregon, USA, October 2006), ACM, ACM Press.

[8] STOERZER, M., CONSTANTINIDES, C., AND SKOTINIDES, T. AOP Considered Harmful. Position Statement for Panel Discussion, 2004. Panel Discussion at Workshop EIWAS 2004, Berlin, Germany.

[9] TARR, P., OSSHER, H., HARRISON, W., AND STANLEY M. SUTTON, J. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 107–119.