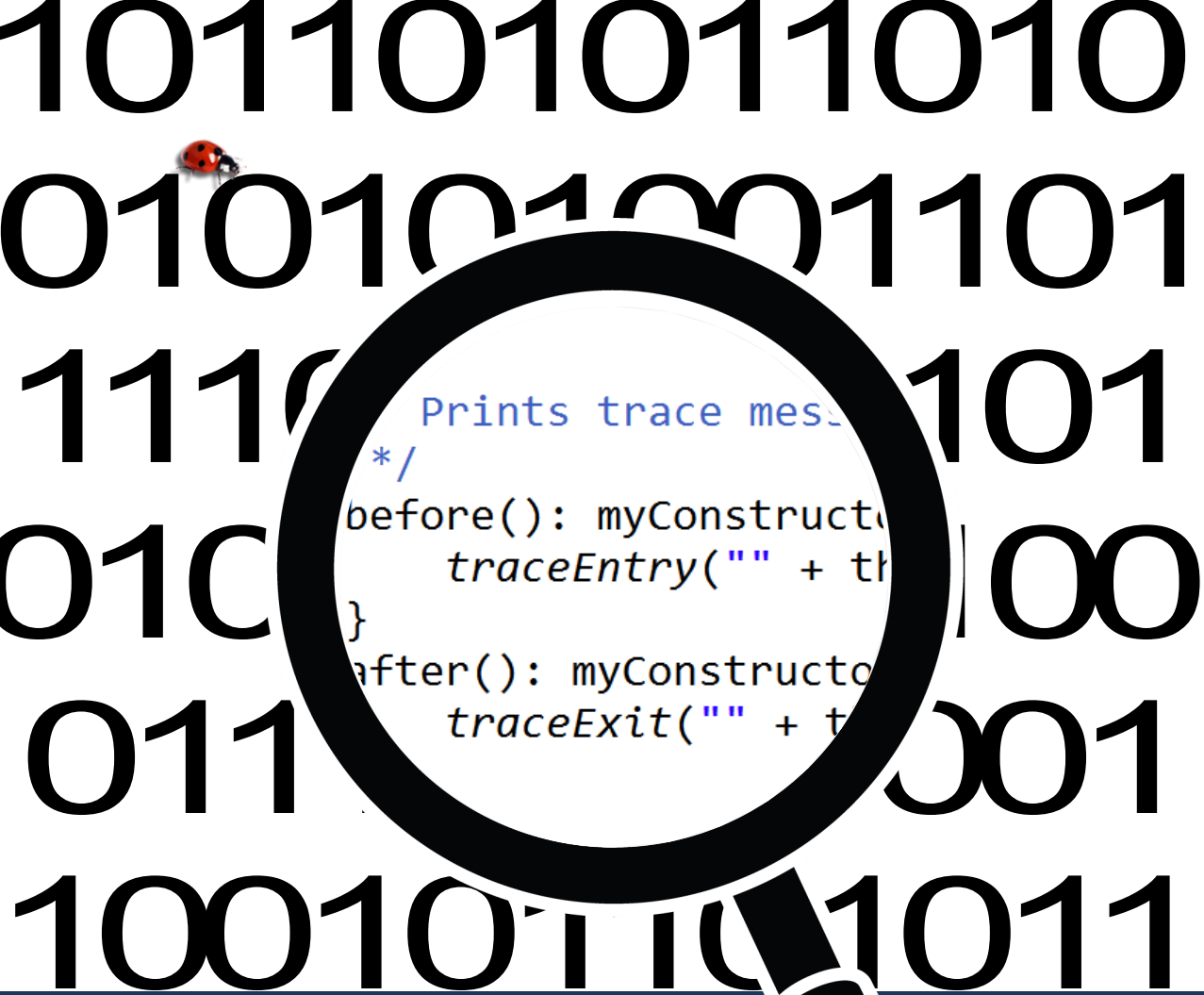


Defusing the Debugging Scandal - Dedicated Debugging Technologies for Advanced Dispatching Languages

Haihan Yin

Defusing the Debugging Scandal - Dedicated Debugging Technologies for Advanced Dispatching Languages

Haihan Yin

Ph.D dissertation committee:

Chairman and secretary:

Prof. Dr. Ir. A.J. Mouthaan, University of Twente, The Netherlands

Promoter:

Prof. Dr. Ir. Mehmet Akşit, University of Twente, The Netherlands

Assistant promoter:

Dr. Ing. Christoph Bockisch, University of Twente, The Netherlands

Members:

Prof. Shigeru Chiba, University of Tokyo, Japan
Prof. Jianjun Zhao, Shanghai Jiao Tong University, China
Prof. Dr. Eric Bodden, Technische Universität Darmstadt, Germany
Prof. Dr. Jaco van de Pol, University of Twente, The Netherlands
Dr. Ir. Maurice van Keulen, University of Twente, The Netherlands

CTIT Ph.D. Thesis Series No. 10-170
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands

The work described in this thesis was performed at the Software Engineering group, Center for Telematics and Information Te Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, The Netherlands.

ISBN 978-90-365-3569-4
ISSN 1381-3617 (CTIT Ph.D. thesis Series No. 10-170).
DOI 10.3990./1.9789036535694

Cover designed by Haihan Yin
Printed by Ipskamp Drukkers B.V., Enschede, The Netherlands

DEFUSING THE DEBUGGING SCANDAL - DEDICATED DEBUGGING TECHNOLOGIES FOR ADVANCED DISPATCHING LANGUAGES

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof. Dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday the 4th of December 2013 at 12.45

by

Haihan Yin

born on the 21st of June 1986
in Changsha, China

This dissertation is approved by

Prof. Dr. Ir. Mehmet Akşit (promoter)
Dr. Ing. Christoph Bockisch (assistant promoter)

To my parent
Jianwei and Chunhui

致我的父母
建伟和春晖

Abstract

To increase program modularity, new programming paradigms, such as aspect-oriented programming, context-oriented programming, and predicated dispatching, have been researched in recent years. The new-paradigm languages allow changing behavior according to various kinds of contexts at the call sites. A recent statistics shows that at least 66% of skilled engineers world-wide, courses and third party vendors are using languages that support AOP features. Many well-known companies are using these languages in product developments.

In our research, we classified these new languages as advanced-dispatching languages. Advanced-dispatching languages are usually implemented as an extension of main-stream languages. After compilation, their programs are transformed to the compiled form of the main-stream language. Due to this compilation mechanism, the information, especially features that are specific to the advanced-dispatching languages, existing in the source code cannot be fully restored during debugging. The information loss increases the effort of comprehending advanced-dispatching programs and fixing advanced-dispatching-specific defects. In this thesis, we performed four works to improve the comprehensibility of debugging information from three debugging techniques — interactive debugging, trace-based debugging, and slicing.

We first performed research on the most common debugging technique— interactive debugging. Due to the availability of existing fault models, we proposed a generalized fault model based on four aspect-oriented-specific fault models. The generalized fault model consists of seven fault types and we found a likely pattern for fixing a defect corresponding to each type. Then, we analysed the pattern description and extracted eleven atomic advanced-dispatching-specific debugging tasks. To support the tasks, we modified an existing compiler to keep aspect-oriented information after the compilation, built a dedicated advanced dispatching debugging model, and implemented graphical user interfaces.

In addition to the form of debugging information, we analysed the problem where and when the program should be suspended. In interactive debugging, breakpoints are essential to determine the suspending places. For fixing a specific defect, breakpoints are logically related. However, existing debuggers do not support to build logic between breakpoints. Therefore, programmers have to manually perform some repeated tasks. In this thesis, we analysed five common debugging scenarios that require multiple breakpoints. Targeting the investigated scenarios, we designed and implemented a breakpoint language that uses point-

cuts to select suspension times in the program. In our language, an expected suspension can be expressively programmed and reached with less or even no irrelevant suspensions.

Though the interactive debugging is the most commonly used debugging technique, it becomes handicapped in cases that the observed failure is far away from the defect. Locating the defect is typically a backtracking process through the execution history. However, interactive debugging is performed along with the program execution and cannot restore a past state. Trace-based debugger records events at runtime and supports inspection offline. Existing researches on trace-based debugging for advanced-dispatching languages are limited. We proposed a dedicated trace-based debugger with an user interface that allow programmers to navigate and query the recorded trace.

Automation is what software aims for. Slicing is a debugging technique which automatically selects program statements that can influence the failure. It is performed on dependency graphs that contain the inter-relationships between program statements, such as control dependencies, call dependencies. We discussed how three aspect-oriented-specific constructs, which are join point shadows, program compositions, and non-argument context values, can influence the execution of aspect-oriented programs. Considering the three constructs, we developed dependency graphs that are dedicated to aspect-oriented programs, as well as a slicing algorithm that is performed on the developed dependency graphs. We showed that our approach can be applied efficiently for small projects and is able to select all relevant source code.

The three debugging techniques analysed in this thesis fully cover phases before, during and after the program execution. For each technique, we have provided a debugging model that explicitly models advanced-dispatching concepts. The generated debugging information is in terms of advanced-dispatching abstractions. Thus, the comprehensibility of debugging advanced-dispatching programs is increased.

Acknowledgements

Life is like a running program with occasional failures. Some people choose to ignore failures and live an easy life. In the end, those failures may deviate their lives from what they expected further and further. Some people choose to discover the cause, eliminate such failures, and live happily ever after. I started my PhD journey in 2010 October. Since then, failures have never ceased to accompany with me. People around me gave me guidances and supports to overcome these failure. This thesis belongs to everyone who ever helped me.

First, I would like to appreciate my promoter Mehmet Akşit to offer me the opportunity to work in the software engineering group as a PhD student. Your insightful guidance showed me promising directions to work on. You have always been a great source of inspiration for me.

Christoph Bockisch is my daily supervisor during my entire PhD phase and he is also the supervisor of my master project. No word can describe how I am grateful about what he contributed to my work. There is a saying in China, "Even if someone is your teacher for only a day, you should regard him like your father for the rest of your life." Interestingly, people call a supervisor as "PhD father" in German. You are always patient to listen to my thoughts even they are simple and naive. You always encourage me even though I sometimes submitted really unsatisfied work. You are the one who can always make time for me even for unexpected visits. You always explain things step by step and teach me things hand by hand. In the third round submission to AOSD'12, we had discussion almost every day and you were extremely strict with the quality of the paper. I wrote a crucial section with only two pages eleven times. As a result, our paper not only got accepted but also won the "best paper" award. It was a great encouragement to me and I became much more confident since then. What I learnt from you is not only about research and knowledge, but also the German way of thinking and working. I digested your way of supervision and successfully applied it to three master students, including two Chinese students. It is my great fortune to choose you as my supervisor. I wish you, your wife, cute Julia, lovely Sarah, and your families all the best. Prost!

I also would like to thank the members of my PhD committee: prof. Shigeru Chiba, prof. Jianjun Zhao, prof.dr. Eric Bodden, dr.ir. Maurice van Keulen, prof.dr. Jaco van de Pol. I am very honored that you accepted our invitation to be part of my committee and thank you for reading my thesis and providing valuable feedback to improve it.

I enjoyed my life in Software Engineering group. Our secretary Jeanette Rebel-de Boer supported me a lot in handling administrative tasks. If I have any non-academic problem, I can always get a suggestion or even a solution from her. Suse Engbers, who is the secretary of the shortly established ISSE group, helped me to successfully apply for a dormitory before I almost became "homeless". I would like to mention my office mates, with whom I had fun over the years: Arda Goknil, Arjan de Roo, Kardelen Hatun, Somayeh Malakuti, Stenven te Brinke, Yongsheng Shen, Zhou Lu, Bugra Mehmet Yildiz. I especially want to thank Steven te Brinke. You reviewed my paper twice with detailed comments and your valuable opinions helped me to win the second price of PhD carousel in CTIT symposium 2012.

In September 2013, I had a one-month visit in Shanghai Jiao Tong University. Prof. Jianjun Zhao arranged my work and accommodation. We had several inspiring talks about the cooperation in the future. The cooperation is and will be an important bridge between the two groups. I also had a great experience working with Fei Lv and Jingzhou Liu. Their diligence and modest attribute are quite impressive. I especially thank Chang Cui for her enthusiasm. Shanghai doesn't become yet another city because of you.

PhD study is sometimes boring, frustrated, and tough. Fortunately, there are always friends around me and they make my life much more cheerful. Qiu Xian and Feng Yuan, thank you for sharing so much information during lunch talks. Yanting Chen, Wei Cheng, and Yu He, the life when we live together was so much fun. Song Ma and Wei Chen, thank you for always inviting me to dinner parties. Chen Ling, thank you for introducing horse riding to me. There are so many friends that I shared unforgettable moment with, Zhiyu Ru, Dongshuang Hou, Pengkun Chen, Junwen Luo, etc. I hope our friendship lasts no matter where we are.

Last, but not the least, I want to thank my family. My uncle Zhisheng Huang always guides me at the spiritual level and shows me a fresh view to the world. My aunt Yuanhua Chen cooks the best food I can have in the Netherlands. My dear cousin Lanhong Huang, those open-heart talks with you decorated my life.

I dedicate the greatest thank to my parent: Jianwei Yin and Chunhui Feng. It is your love that keeps me pacing forward steadily. No matter I am happy or sad, successful or failed, you are always there and support me. I love you!
我将我最大的感谢送给我的父母：殷建伟和冯春晖。是你们的爱，让我一步一步坚定前行。不管我是高兴或忧伤，是成功或失败，你们总在那，在那支持我。我爱你们！

Haihan Yin
December 2013

Contents

List of Figures

List of Tables

1

Introduction

To increase program modularity, languages such as AspectJ[1], Compose*[2] and JPred[3] are proposed. These languages allow choosing behavior alternatives without modifying the call site of that behavior. In this thesis, we call them *advanced-dispatching (AD) languages*.

Aspect-oriented programming (AOP) is the most wide-spread type of AD. It is a programming paradigm that can modularize concerns that are cross-cutting in systems with only traditional paradigms. Since it was first proposed by Grogor Kiczales at Xerox PARC in 1996, it never ceases to influence the following development of programming languages as well as the involved software. In the ranking released on TIOBE [4] in October 2013, 9 out of the 10 most popular programming languages have been extended with AOP[5]. It indicates that at least 66% of skilled engineers world-wide, courses and third party vendors are using languages that support AOP features. Well-known frameworks, such as JBoss[6] and Spring [7], apply AOP in their core components. Influential companies, such as ASML [64] and Motorola [19], integrate AOP techniques in product development to seek better quality and more efficiency. Other types of AD, such as predicate dispatching, context-oriented programming, and feature-oriented programming, are mainly used in the field of language research.

In the life cycle of software, almost 25% of maintenance is carried out for repairing faults [55] and locating defects, which is usually called *debugging*, is a crucial task. While use and research of AD languages are booming, the de-

[1]See `http://eclipse.org/aspectj/`.
[2]See `http://composestar.sourceforge.net/`.
[3]See `http://www.cs.ucla.edu/~todd/research/jpred.html`.
[4]See `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.
[5]See `http://en.wikipedia.org/wiki/Aspect-oriented_programming`.
[6]See `http://www.jboss.org/jbossaop`.
[7]See `http://docs.spring.io/spring/docs/2.5.x/reference/aop.html`.

velopment of debugging support for AD languages is stagnant. In 1997, Henry Lieberman proclaimed the "Debugging Scandal" [54]: the computer science community had largely ignored the debugging problem and debugging methods were still in the state as they were 30 years ago.

Today, we are once again facing the next generation of the "Debugging Scandal". Like the first implementations of many programming languages, such as C++ and Java, the approach of implementing AD languages is to transform the source code to code of an established language, the so-called base language. While this facilitates the use of tools like debuggers that already exist for the platform of the base language, developers are stuck with debugging their code at the level of abstractions of the base language: They see a complex, synthetic composition of low-level abstractions [83]. Early in 1972, debugging research already argued that "All information presented to the user should be expressed in terms of his source program" [73]. However, before we started the work described in this thesis, little to none dedicated debugging support for AD languages have been developed.

This chapter is organized as follows. It starts with the background on general concepts used in this thesis. This is followed with the problem statements of this thesis. Last, we briefly discuss four solutions to the proposed problem statements as well as contributions of each solution.

1.1 Background on Debugging

As the name suggests, debugging is the process of removing bugs. In this thesis, we choose to reuse the terminology defined in a book written by A. Zeller [90]. He describes debugging as follows:

> *The issue of debugging is to identify the infection chain, to find its defect, and to remove the defect such that the failure no longer occurs.*

A *defect*, which is also called the *root cause*, is a *mistake* made by programmers in the code that can cause infections. *Infections* may occur after the defect is executed. An infection is a deviation of the actual execution from the programmers' expectation. It propagates in the following execution and leads to other infections. A *failure* is an observable unexpected symptom, such as a wrong behavior or a wrong program state. The cause-effect chain from defect to failure is called an *infection chain*. Some works also call it a *cause-effect chasm* [25].

Figure 1.1 shows a generic debugging process model (DPM), which includes activities involved in debugging. The source program written by programmers is transformed to the debuggee program. During debugging, programmers selectively search part of the debugging information generated from the debuggee

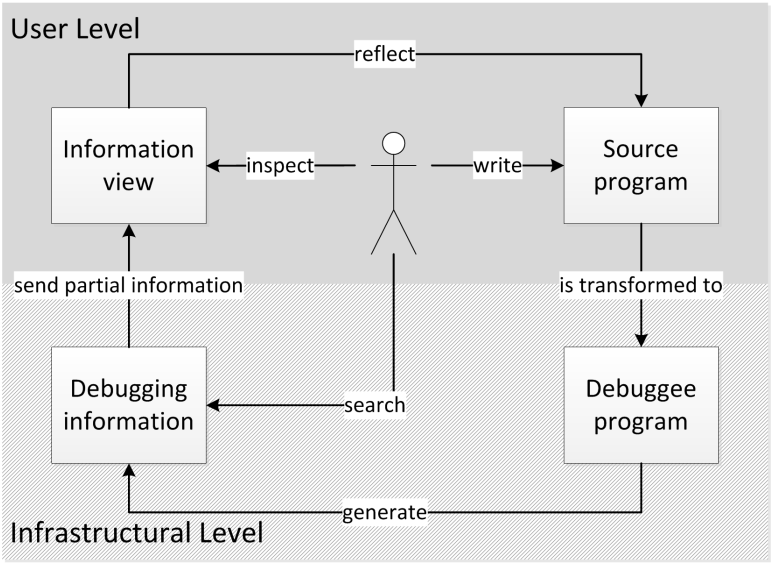


Figure 1.1: A generic debugging process model

program. The interesting debugging information is sent to an information view, which reflects the corresponding source code.

Various debugging techniques have been developed to assist programmers to find the defect. Among existing techniques, interactive debugging, trace-based debugging, and static slicing are explored in our work. We use an example Java program in Listing 1.1 to introduce each of these debugging techniques in the following subsections.

```
1  public class Point {
2    private int x;
3    private int y;
4    void setX(int _x) {
5      x = _x;                                    ▲
6    }
7    void setY(int _y) {
8      y = _y;
9    }
10   public static void main(String[] args) {
11     Point p = new Point();                     ▲
12     p.setX(−2);                                ▲
13     p.setY(1);
14   }
15 }
```

Listing 1.1: A sample Java program

Interactive Debugging

Interactive debugging is one of the most common debugging techniques and it is the initial debugging support, such as jdb[1], gdb[2], and pdb[3], for almost all the popular languages. Interactive debugging suspends program execution and provide inspections of the program state at that suspension. Suppose the program is suspended at line 13, statement p.setX(−2) has been executed but p.setY.(1) has not yet. Therefore, the state of the point p is "x=-2, y=0". The interactive debugging is conducted along with the execution of the debuggee program. It only provides information at the current suspension and past or future information is not accessible.

Breakpoints Programmers can inspect the program state only when the execution is suspended. To suspend the program, an essential step is setting breakpoints, which define where and when suspensions should occur. There are different ways of setting a breakpoint. In *jdb*, programmers need to type the corresponding command, like "stop at Point:8". In a modern IDE like Eclipse, programmers only need to double-click on the vertical bar of the editor.

When a breakpoint is *set*, it is *activated* by default. A breakpoint works only when the program runs in the debugging mode. When the execution reaches a breakpoint, the breakpoint is *hit* and the execution suspends. When an activated breakpoint is *deactivated*, it does not work as if it has never been set until it is activated again. Finally, a breakpoint is discarded after it is *deleted*.

The most common type of breakpoint is the *line breakpoint*, which suspends the program when the execution reaches the first instruction of that line. The *watchpoint* is set on a field and it suspends the program when the corresponding field is accessed, modified, or both. The *conditional breakpoint* has additional conditions, such as an expression and desired hit counts, evaluated when the breakpoint is reached. If the evaluation result is true, the breakpoint is hit and the program suspends.

Stepping Actions When the program is suspended, programmers can perform stepping actions to control the execution and choose following suspensions without setting breakpoints. These actions increase the efficiency of exploring the adjacent execution that follows a suspension. Common stepping actions are listed below.

- **Step into** executes the next instruction. The execution suspends on the first line of a method if the instruction invokes the method.

[1]See `http://docs.oracle.com/javase/1.3/docs/tooldocs/solaris/jdb.html`.

[2]See `http://www.gnu.org/software/gdb/`.

[3]See `http://docs.python.org/2/library/pdb.html`.

- **Step over** executes the current line. The execution suspends on the next executable line.
- **Step return** resumes execution until the return statement in the current method is executed. The execution suspends on the next executable line.
- **Resume.** The execution continues to run until it hits another breakpoint.
- **Terminate.** The execution is terminated immediately.

Runtime Inspections There are mainly three ways of inspecting program states. First, the values of variables are presented as printed information or in a dedicated variable view. Second, expressions can be evaluated with variables at the current state. Third, if an expression is *watched*, it is evaluated automatically at every suspension.

The *stack trace* shows how the execution comes to the current point from the program entrance, like the main() method in Listing 1.1. A stack trace consists of a list of *stack frames* and each frame represents the execution of a method or a constructor. Stack frames are organized in the *first-in-last-out* order in a trace. Therefore, the current suspended place is represented by the top frame.

Locating is another important debugging task, because it bridges the information presented by the debugger and the source code written by the programmer. Eventually, programmers need to locate the defect in the source code according to the information presented by debuggers.

Trace-Based Debugging

Trace-based debugging [22, 40, 65, 71] records events that occur during the execution, and then lets programmers debug the program on the recorded information. This technique originates from *print debugging* where programmers insert statements in the source code to print program states. After the execution, programmers can analyse the printed information and deduce where the infection started. The print debugging requires significant manual effort in adding and deleting printing statements in the source code. Trace-based debugging records information through a dedicated debugger instead of modifying the source code.

Two key challenges of trace-based debugging [53] are (1) determining what events should be recorded and (2) providing information in an efficient manner so that programmers can conveniently use it. Listing 1.2 shows part of an execution record of the code in Listing 1.1. Each line consists of the description and the source location of the event. A fundamental advantage of trace-based debugging is that it allows programmers to navigate the recorded information in any manner, such as stepping one instruction back, stepping to where an object is created, and even jumping to any record.

```
1 ......
2 create an object of Point {Point.class:line 11}
3 assign the object to variable [p] {Point.class:line 11}
4 call Point.setX() on [p] with parameters [−2] {Point.class:line 12}
5   execute Point.setX() {Point.class:line 4}
6   assign [_x] to Point.x {Point.class:line 5}
7   return Point.setX() {Point.class:line 6}
8 call Point.setY() on [p] with parameters [1] {Point.class:line 13}
9 ......
```

Listing 1.2: A pesudo code of a recorded trace

Static Slicing

Slicing automatically selects statements which are relevant with respect to a criterion. The criterion refers to the value of a variable or the execution of a statement. Slicing is performed on *dependency graphs* that consist of nodes representing elements in the source code and arcs representing the inter-relationships between those elements.

Slicing can be divided to *static slicing* and *dynamic slicing*. Static slicing [27, 43, 50, 81] selects statements that may satisfy the criterion. Dynamic slicing [1, 49] selects statements that actually satisfy the criterion with a given input. In this thesis, we are only interested in static slicing.

Figure 1.2 shows a dependency graph of the code in Listing 1.1. Circles represent member declarations and program statements, and we call them *declaration nodes* and *statement nodes* respectively. The rectangle represent class Point. Circles are labelled by the line numbers of referred source elements. Ellipses represent parameters, and we call them *parameter nodes*. Labels ai_in or ai_out are for actual parameters and fi_in or fi_out for formal parameters. The legend illustrates different types of arcs as well as their usages.

Two Traverse Slicing Any runtime state in a typical program execution contains a call stack, which tells how the execution comes to the current point. The stack contains a sequence of frames and each frame represents a call site. A call site is in the control flow of methods represented by preceding frames and takes returned values from methods represented by preceded frames.

According to this, Horwitz et al. [43] proposed a slicing algorithm for interprocedural programs in which one procedure can call another. The algorithm consists of two steps: ascending traverse and descending traverse. In the first step, the algorithm traverses backwards along all arcs except parameter-out arcs, and marks nodes reached in the DG. In the second step, the algorithm traverses

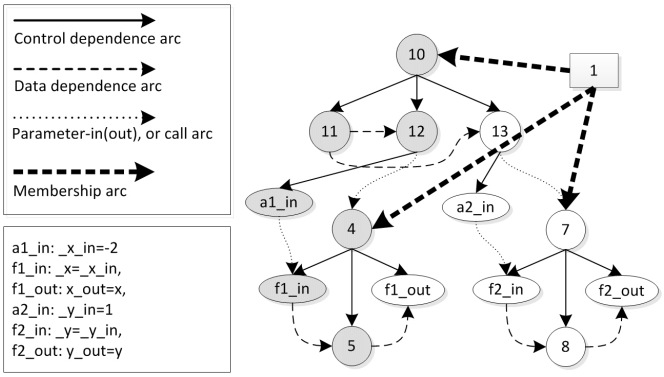


Figure 1.2: A dependency graph of the program in Listing 1.1 and a slice on slicing criterion (*5*, x)

backwards from all nodes having been marked during the first step along all arcs except call and parameter-in arcs, and marks reached nodes in the DG. The slice is the union of the marked nodes during the two steps. The separation of two traverses can exclude other call sites of a procedure called by the investigated statement.

To find all statements that can influence either the value of x or the execution of the statement on line 5, we can use the two-traverse slicing. The first traverse starts from node 5 and backwards marks nodes $f1\_in$, 4, $a1\_in$, 12, 11, and 10. Since there is no parameter-out arcs connecting the marked nodes, the second traverse does not mark any node. Figure 1.2 shows the slicing result as shaded nodes. The executable statements in the slicing result are marked with black triangles in Listing 1.1.

A Comparison between the Three Debugging Techniques

Each techniques has its strengths and weaknesses, which are independent from the programming languages they serve. The first two techniques provide runtime information but static slicing analyses source code. We first compare interactive debugging and trace-based debugging, and then we state how static slicing complements the first two.

Interactive debugging provides comprehensive information at a suspension. Some interactive debuggers are so "interactive" that they allow programmers to edit code or even modify runtime state during suspension. The following execution will be executed according to the changed program instead of the original program. This is not possible in trace-based debugging, because it only records execution and provide analysis offline on the recorded information.

Locating the defect from the observe behavior is a backward analysis. This

is contradictory to the working mechanism of interactive debugging. Interactive debugging is performed along with the program execution. If a suspension passes the defect, it may require to start over a new debugging session. Trace-based debugging provides more flexible ways of inspecting information. It can provide an overview of the execution, an inspection at any point of the recorded information, and backwards inspection. In the case with a significant cause-effect chasm, using fully-recording trace-based debugging is easier to locate the defect. However, a price needs to be paid to use such a debugger—tremendous runtime overhead. To reduce the overhead, a trace-based debugger typically allows programmer to configure which information should be recorded. If an expected information is not recorded, the trace-based debugging needs to be reconfigured and start over.

Static slicing is performed on dependency graphs, which are generated from the source code. It cannot directly help programmers to find the defect. Instead, it provides a list of suspicious code, which has influence on the code where the failure is observed. In a large-size project, static slicing can significantly reduce the scope of code that needs to be further analysed with runtime information. A slice can direct programmers to suspend the program in interactive debugging or configure a trace-based debugger. Besides, dependency graphs can be used in detecting code anomalies, such as dead code or infinite loops, which are difficult to be detected by using runtime analysis techniques.

1.2 Problem Statement

AspectJ is the most popular aspect-oriented programming (AOP) language and it is implemented as an extension of Java. Listing 1.3 shows an example AspectJ program, which consists of a class Main and an aspect Monitor. Aspect Monitor contains an advice (lines 8-10) that is executed *before* method Point.setX(**int**) is called with a negative argument. AspectJ programs are compiled to bytecode and executed on JVM. In the following subsections, we describe the difficulties of debugging AspectJ programs based on the compiled code and the source code respectively. Though we use only AspectJ as the example, debugging programmers written in other AD languages also suffers from the same problems.

```
1 public class Main {
2   public static void main(String [] args) {
3     Point p = new Point();
4     p.setX(2);
5   }
6 }
```

```
7  public aspect Monitor {
8    before(int i) : call(public void Point.setX(int)) && args(i) && if(i<0) {
9      printWarning();
10   }
11 }
```

Listing 1.3: A sample AspectJ program

1.2.1 Bytecode-Based Debugging

Listing 1.4 shows the bytecode of method Main.main() (lines 2-5, Listing 1.3). The link between the source code and the bytecode on lines 10-14 in Main.main() is not obvious. Suppose the program is suspended on line 4 in the source code, which corresponds to line 6 in the byte code, programmers expect the next suspension is in the body of setX() after performing a "step-into" command. However, according to the bytecode, the next suspension jumps into method the synthetic ajcifba() (line 10).

```
1  public static void main(java.lang.String[] args);
2    0 new base.Point [17]
3    3 dup
4    4 invokespecial base.Point() [19]
5    7 astore_1 [p]
6    8 aload_1 [p]
7    9 iconst_2
8    10 istore_2
9    11 iload_2
10   12 invokestatic azpect.Monitor.ajc$if$ba(int) : boolean [42]
11   15 ifeq 25
12   18 invokestatic azpect.Monitor.aspectOf() : azpect.Monitor [35]
13   21 iload_2
14   22 invokevirtual azpect.Monitor.ajc$before$azpect_Monitor$1$3f7b0(int) : void [38]
15   25 iload_2
16   26 invokevirtual base.Point.setX(int) : void [20]
17   29 return
```

Listing 1.4: Byte code of method Main.main() in Listing 1.3)

During the compilation, the AspectJ infrastructure transforms AspectJ-specific language constructs to Java constructs so that AspectJ programs can run on a standard JVM. The transformations are such that the information, especially AspectJ-specific features, existing in the source code cannot be fully restored in debugging tools, such as interactive debuggers and trace-based debuggers, built on

the byte code. The information loss increases the effort of comprehending AspectJ programs and detecting defects related to AspectJ-specific language constructs. The programmer has to manually map the low-level abstractions presented by the debugging tool to the source language. This requires intimate knowledge of the compilation strategy that transforms the source code to a synthetic host language program.

1.2.2 Source-Based Debugging

As an extension of Java, AspectJ introduces new language features, such as aspects, join points, and advices. The new features further introduce new types of interactions with the traditional parts, such as advising, program composition, and introduction. Static debugging techniques typically use an abstraction model of the source program. The abstraction model can be seen as the compiled code on which the static debugging can be performed.

Unlike bytecode-based debugging, the source-based debugging can be wrong or even cannot be carried out if no proper extension has been applied. For example, slicing is performed on dependency graphs, which reify the internal relationships between elements in source code. The process of building dependency graphs for Java programs is unable to handle AspectJ-specific constructs. Thus, slicing on AspectJ programs cannot be performed. However, few of existing works have thoroughly explored AspectJ-specific features in slicing.

1.3 Overview and Context

What we claim in this thesis is that with a new generation of programming languages and with increasing tool support for early development phases, the "Debugging Scandal" rekindles. New tools and techniques are required that allow observing and interacting with program executions in terms of abstractions natural to the developer. These abstractions must correspond to the language that the developer used to define program elements.

The work presented in this thesis is initially carried out as an extension of project ALIA4J[1]. ALIA4J contains an AD-language model LIAM and an execution environment NOIRIn where AD-specific concepts are modelled and run as first class objects.

To allow programmers to access runtime states of NOIRIn in real time, we first explored the interactive debugging, which can suspend the execution of the debuggee program. Our idea of developing an interactive debugger for AD languages was firstly published in the third International Workshop on Academic

[1]See `http://www.alia4j.org/alia4j/`.

Software Development Tools and Techniques (WASDeTT'2010) [86]. The work was then greatly developed and published in the 11th International Conference on Aspect-Oriented Software Development (AOSD'2012) [87]. As the best paper in AOSD'2012, the work is once again extended and published in Transactions on AOSD volume 10 [89].

As an essential and crucial step in interactive debugging, setting breakpoints in traditional ways is not efficient. We developed a novel approach to describe breakpoints by using an AspectJ-like language and published this work in the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013) [88].

An inherent drawback of the interactive debugging is that only the current snapshot of the execution is accessible but the execution history cannot be provided. This causes troubles when programmers need to inspect a past state. Therefore, we explored the trace-based debugging for AD languages and discussed our preliminary study on this domain in a workshop paper (1st Workshop on Comprehension of Complex Systems, CoCoS'2013) [80]. Later on, we fully designed and implemented a trace-based debugger and present it in this thesis.

The interactive debugging and the trace-based debugging are bytecode-based debugging approaches. For the source-based debugging, we looked at slicing particularly and discuss our progress so far in this thesis.

1.3.1 Context: ALIA4J

Dispatching is the process of resolving abstractions and binding concrete functionality to their usage. The place where a dispatching is performed is called *dispatch site*. The bound functionality is determined by the runtime context at the dispatch site. A typical example is the receiver-type polymorphism of object-oriented programming. *Advanced dispatching* (AD), which includes aspect-oriented programming (AOP) and predicate dispatching, requires contexts which are beyond the receiver type, such as the control-flow in AOP languages and the argument types in predicate dispatching.

The implementation of any programming language, advanced-dispatching or not, typically consists of two parts, a *front-end* and a *back-end*, which are decoupled by means of an intermediate language. The front-end processes source code and compiles it into the intermediate language. The back-end either executes this *intermediate representation* (IR) directly or further compiles it into a machine-executable form.

An AD language, such as AspectJ and JPred, is usually implemented as an extension of a main-stream language, which is also called the *base language*. The front-ends of AD languages transform the AD programs into the IR of the base language, which cannot directly express some AD concepts.

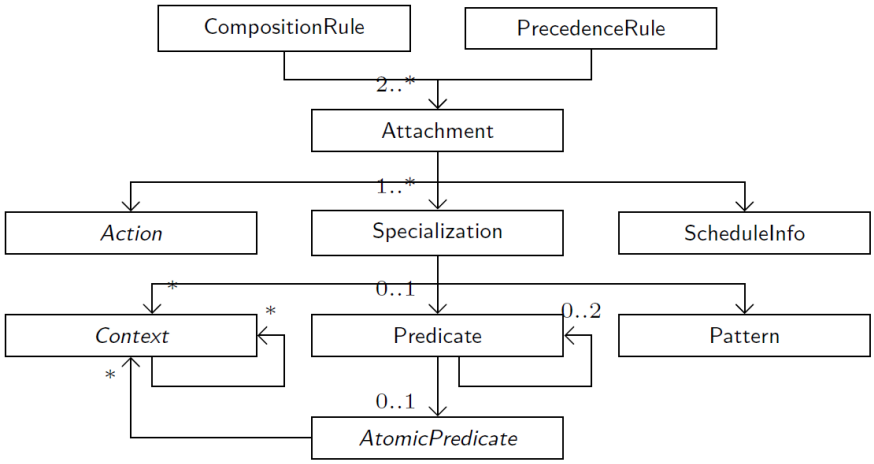


Figure 1.3: The LIAM meta-model of advanced dispatching

ALIA4J is an architecture for implementing AD languages. It raises the abstraction level of the IR for AD languages by proposing a language independent meta-model (LIAM). LIAM models preserve the AD abstractions after the front-end transformation. Besides, ALIA4J contains a framework (FIAL), which defines common workflows to execute LIAM models. It has three instantiations with different execution strategies. This thesis only uses the NOIRIn execution environment, which is implemented as an interpreter and therefore easy to be extended. The goal of ALIA4J is to ease the burden of programming-language implementation resting upon researchers of new abstraction mechanisms; also several domain-specific programming languages can be mapped to a dispatching problem.

A Meta-Model of Advanced-Dispatching

The meta-model, LIAM, defines *categories* of language concepts concerned with (implicit) invocation and how these concepts relate; e.g., a dispatch may be ruled by *atomic predicates* which depend on values in the dynamic *context* of the dispatch. LIAM has to be *refined* with the concrete language concepts like the **cflow** or **target** pointcut designators of AspectJ. Figure 1.3 shows the structure of LIAM.

Pattern. Each dispatch site, e.g., field access or method call site, has a signature. These signatures can be quantified over by means of LIAM's Patterns. As one can model the majority of advanced-dispatching languages using just five kinds of dispatch sites (calling a method, constructor, or static initializer, and reading or writing a field), LIAM offers five predefined subclasses of Pattern.

Context. Dynamic dispatch is context-dependent. In LIAM, the Context meta-entity models this dependence on runtime values: Its refinements embody different kinds of values that are available during dispatch, e.g., a single argument (ArgumentContext) or the receiver object (CalleeContext). Context entities can also model values that are derived from other values, e.g., the reification of an AspectJ join point (ThisJoinPointContext) accessible through the **thisJoinPoint** keyword, which aggregates various values like the caller, callee, and arguments, all of which are modeled by contexts of their own.

AtomicPredicate. The Atomic-Predicate meta-entity models a single test parameterized with Contexts specifying the values to test. For example, a dynamic type check (InstanceofPredicate) parameterized with an argument of a call (ArgumentContext) models multiple dispatching.

Predicate. A Predicate is composed by AtomicPredicate entities in the tree form, whose inner nodes are either conjunctions or disjunctions and whose leafs are either an AtomicPredicate or its negation.

Action. The Action specifies functionality that is executed once the predicate has been evaluated at a dispatch site.

Specialization The Specialization declares a list of runtime values (Context), which must be exposed to Actions, and defines static (Pattern) and dynamic properties (Predicate) of state on which the dispatch depends.

Schedule Information The Schedule Information models the time relative to a dispatch site when the action should be executed, i.e., before, after or around.

Attachment An Attachment associates an Action, a set of Specializations, and a Schedule Information. When both the Pattern and the Predicate of a Specialization match, the exposed Context is passed to the Action and the Action is executed at the time defined by the Schedule Information.

Precedence Rule and Composition Rule The Precedence Rule models partial ordering of actions and Composition Rule models the applicability of actions at a shared join point.

Interpretation-Based Dispatching Strategy - NOIRIn

NOIRIn is a fully portable JVM extension implemented using the java.lang.instrument API. Using the ASM bytecode engineering library, it replaces every dispatch site in the program with an invocation of a call-back method. When this call-back

is invoked, NOIRIn retrieves the execution model associated with the dispatch site in question and interprets it by traversing the execution model's object structure and evaluating every LIAM entity it encounters. A complete dispatching consists of the following steps in NOIRIn.

Deployment and Undeployment At deployment of an *Attachment*, NOIRIn identifies which of the already-loaded dispatch sites are affected and modifies them accordingly; at undeployment, the corresponding *Action*, *Predicate*, and *Contexts* are removed from the dispatch site.

Dispatch Preparation When a dispatch site is reached during the execution, NOIRIn collects the dispatching contexts, such as the line number of the dispatch site, the type of the callee and caller, the argument values, etc. Based on the dispatching contexts, NOIRIn finds deployed and applicable *Attachments* and creates a *dispatch function*, which is structured as a binary decision diagram (BDD). In the BDD, an inner node represents an *AtomicPredicate* that needs to be evaluated, and a leaf node represents a list of *Actions* that needs to be executed.

Dispatch Evaluation NOIRIn evaluates the dispatch function from the root to a leaf according to the call context. To evaluate a LIAM entity, first, all context entities it depends on must be evaluated. Next, the evaluation method is invoked passing the result values from the previous step as arguments. During the evaluation of the dispatch function, the traversal of the BDD depends on the boolean value returned by the evaluation of the *AtomicPredicates*. When the evaluation reaches a leaf, the bound functionality at this dispatch site is determined.

Dispatch Execution *Actions* to be performed at a dispatch site are executed in the order specified by the precedence rules and their schedule information.

1.4 Works and Contributions

In the "debugging scandal" that we claimed, the information presented by existing debuggers cannot reflect the source program correctly. Therefore, the *reflect*-relation in Figure 1.1 is broken. In this thesis, we propose four works to rebuild the *reflect*-relation. They are briefly described in the following subsections. In the conclusion chapter (Chapter 6), we will illustrate how each of these works instantiates the DPM.

1.4.1 An Interactive Debugger for Advanced-Dispatching Languages

We started by systematically generating requirements of building an AD-aware interactive debugger. Generally, the requirement analysis consists of four steps. First, collect existing fault models. Programmers may make mistakes when they write source code, e.g. misuse of constructs, failure to express logic. A fault model describes possible origins for mistakes, which cause failures at runtime. Second, classify the possible mistakes in the collected fault models according to which construct a mistake is from. Third, identify a pattern of fixing bugs corresponding to each fault category. A pattern consists of a set of actions performed by programmers sequentially. Fourth, extract atomic actions from the identified patterns. Each atomic action needs to be supported by a corresponding functionality in the AD debugger accordingly. This process can be generalized and applied to any other newly emerging language.

In our case, we found four fault models for AOP, which is one category of AD. Then, we identified seven fault categories: incorrect pointcut composition, incorrect pattern, incorrect designator, incorrect context, incorrect composition control, incorrect flow change, and violated requirements. Finally, we identified eleven AO-specific atomic actions that programmers need to perform in debugging AO programs, such as evaluating pointcut sub-expressions and inspecting program compositions.

To increase the consistency between the source code and the compiled code, we modified the *abc* compiler [6] so that it can prevent weaving at compile time and store the required AD information in a file that is absent in the traditional compilation. The file contains the inter-relationship between AD constructs, as well as locations of AD constructs. It organizes information as XML elements, and it has a similar structure to that of LIAM.

To perform dispatches that are specified in the source but are not woven in the compilation, we extended the execution environment NOIRIn to interpret the file and build runtime LIAM instances according to the recorded information. Runtime information in NOIRIn is organized in a dedicated debugging model. On top of the debugging model, we extended an existing Java debugger with the identified functionalities.

This work is described in Chapter 2 and its main contributions include:

- An identification of eleven debugging tasks that an ideal debugger for AD languages should support.

- An approach that preserves information of AD-specific constructs as well as their locations after compilation.

- A dedicated debugging model that models AD-specific concepts as first-class values.
- A user interface that provides functionalities, which includes visualizing dispatches, to support all eleven required tasks.
- A debugger structure that allows language designers to customize the textual representations on the user interface.

1.4.2 A Pointcut Language for Setting Advanced Breakpoints

An essential step of using interactive debuggers is setting breakpoints. Breakpoints define the places where a program is suspended and programmers can inspect the runtime states at that suspension. To accomplish a specific task, such as finding the defect of a bug, the locations of breakpoints tend to be logically related, e.g., sharing syntactical characteristics, referring to the same variable, etc. However, existing debuggers view breakpoint as individual units. Programmers need non-trivial manual effort in some debugging scenarios. We identified the following five scenarios which are frequently encountered. Though four out of five scenarios are based on programs that do not contain AD features, all scenarios are applicable to AD programs, because our investigated AD languages are designed as extensions of a base language. More importantly, this is the first exploration of using a pointcut language to define breakpoints.

The scenarios are as follows:

- Selecting multiple locations with common syntactical characteristics like places creating objects of a specific class. This requires programmers to manually find all locations with join point shadows satisfying the syntactical requirements within the scope of the system.
- Monitoring operations that are called on a specific field, such as modifying a collection. This requires programmers to manually find all locations with join point shadows satisfying the syntactical requirements within the scope of the field's class.
- Finding which dereference operation on a line causes a *NullPointerException*. This requires programmers to repeatedly suspend the execution at each dereference operation on the specific line until the cause is found.
- Recording values of variables in the past execution and using them at the current suspension. The recorded information can be used for validation,

evaluation, enabling or disabling other breakpoints, etc. This requires programmers to record and use information manually.

- Exploring join points advised by multiple interesting advices. Programmers need to manually find all the join point shadows shared by the interesting advices and verify the suspensions at those join point shadows at runtime.

We sought solutions in existing pointcut languages, such as AspectJ and Trachmatch [3]. None of them can completely express the logic of all five scenarios. Besides, they do not provide a mechanism of separating source programs and programs for defining breakpoints. This lack of separation may introduce unnecessary maintenance effort in the future.

Targeting all scenarios, we proposed a breakpoint language which models breakpoints as first-class values. Breakpoints are named and they are defined by AspectJ-like pointcuts which use comprehensible source-level abstractions. We devised five completely new pointcut designators and improved two of AspectJ's pointcut designators. In our language, primitive and composite breakpoints are treated uniformly and the composition level can be infinite. It is the first language to support selecting join points with a specific advice composition.

To validate our work, we performed a code analysis on nine Java projects and five AspectJ projects. We defined seven metrics, such as to what extent a constructor or a method is overloaded, for collecting corresponding statistics and calculating the probable manual effort involved in each scenario. Our results show that a significant portion of cases involve more than one source location. In these cases, our solution can save programmers unnecessary effort.

This work is described in Chapter 3 and its main contributions include:

- An identification of five frequently encountered debugging scenarios that require to use multiple breakpoints, which are related to each other.
- A code analysis on nine Java projects and five AspectJ projects to measure the possibility of setting multiple breakpoints in each identified scenario.
- A novel pointcut-like language for setting breakpoints. It is the first approach that can select a join point with a specific advice composition.

1.4.3 An Trace-Based Debugger for Advanced-Dispatching Languages

A defect is always executed before the failure is observed. To find the defect from the failure, debugging is actually a backward process. A typical interactive debugger only supports forward debugging, which means that programmers cannot

freely inspect past states in a single debugging session. This conflict increases the effort of finding defects by using interactive debuggers.

A better solution is trace-based debugging. Trace-based debugging records information at runtime and provides inspections of the recorded information off-line. Programmers can navigate the recorded information forward, backward, or even jump to an arbitrary point. However, with existing trace-based debuggers, the information is generated from the woven code.

We first designed and implemented a trace model, an identifier model, and a storage model. The trace model defines which information needs to be collected at runtime. The identifier model defines how different entities are identified in a uniform way. The storage model defines how the collected information is structured on the storage media. It organizes the information from different calling depths in a nested structure.

The prototype is implemented on NOIRIn, because of its AD awareness. We built an graphical user interface to navigate and inspect the recorded information. To render abundant information within a limited space, we use a tree-map visualization to view information at one calling depth. Programmers can zoom in or zoom out the visualization to switch between different depths. In addition to step-wise navigation, we allow programmer to perform queries on the recorded information. A distinguishing feature of our work is that we support AD-specific queries. Programmers can search information, such as executions skipped by an around advice, pointcuts with failed evaluation, precedence rules evaluated at a specific join point, etc. We selected frequently used queries and implement them as a library.

This work is described in Chapter 4 and its main contributions include:

- A trace model that contains AD-specific events as well as their relationships with other events in execution.
- A library that implements frequently used AD-specific queries and stepping actions in XQuery.
- The first approach that supports query in trace-based debugging for AD languages.

1.4.4 A Slicing Algorithm for Aspect-Oriented Programs

In addition to debugging techniques that require manual inspections, we also investigated an automated technique—slicing. Slicing can automatically filter relevant statements according to a given criterion. A criterion is the execution of a statement or the value of a variable. Slicing is performed on dependency

graphs (DG) that consist of nodes representing elements in the source code and arcs representing the inter-relationships between those elements.

AOP languages introduce new source constructs that are not modelled by the traditional DGs. Some of them, such as advice precedence, determine how the program is executed. Therefore, supporting these constructs in the DG is crucial for slicing AO programs. We identified three fundamental problems that must be considered in slicing AO programs statically. Without considering these three problems, either the DG cannot be constructed or slicing cannot be performed. First, join point shadows (JPSs) are places where advices are applied. Naturally, we need elements representing JPSs to bridge DGs of advices and advised programs. Second, multiple advices can be applied to the same JPS. A different execution order of these advices can cause a different work flow and eventually may generate a different output. Therefore, it is necessary to contain information of schedule and precedence of advices in the DG. Third, arguments are already included in the traditional DG. However, advices also access non-argument-context values. These values may be modified by advices and passed to the advised program. Thus, slicing on the non-argument-context values should be supported.

Considering the three problems, we extended the traditional DG with AO-specific features. The most notable features are: (1) extending the declaration nodes of members to be JPSs; (2) connecting declaration nodes to advices through coordination arcs labelled with join point type and advice schedule; (3) adding precedence arcs between related advices. Accordingly, we developed a slicing algorithm and discussed how each extended feature is used. Some features are added for comprehension purposes and not necessary required in the algorithm.

We evaluate this work in two ways: the efficiency of building a DG and the effectiveness of the slicing algorithm. The result of the efficiency evaluation shows that the compilation overhead is acceptable for small projects. The result of the effectiveness evaluation shows that our slicing algorithm can include all relevant nodes.

This work is described in Chapter 5 and its main contributions include:

- An identification of three key problems that need to be solved to correctly slice AOP programs.
- An extension of the declaration nodes of members to be join point shadows.

2

A Fine-grained, Customizable Debugger for Advanced-Dispatching Languages

2.1 Introduction

Aspect-oriented programming (AOP) allows programmers to modularize concerns which would be crosscutting in object-oriented programs into separate *aspect*s. An aspect can define functionality *and* when it must be executed, i.e., other modules do not have to explicitly call this functionality. Due to this implicitness, it is not always obvious where and in which ways aspects apply during the program execution. A recent study carried out by Ferrari et al. [29] focuses on the fault-proneness in evolving aspect-oriented (AO) programs. They investigated the AO versions of three medium-sized applications. It shows that 42 out of 104 reported AOP-related faults were due to the lack of awareness of interactions between aspects and other modules.

For locating faults in AO programs, a programmer can inspect the source code and browse static relationships. This is supported by tools like the AspectJ Development Tools (AJDT)[1] and Asbro [69]. To detect a fault in this way, programmers are required to inspect multiple files and mentally construct the dynamic program composition, which is a tedious and time-consuming task. Furthermore, connections between aspects and other modules are often based on runtime states which cannot be presented by static tools. Debuggers are, thus, needed for inspecting to the runtime state to help programmers understanding the program behavior and eventually finding a fault.

AOP languages are nowadays compiled to the intermediate representation

[1] See `http://www.eclipse.org/ajdt/`.

(IR) of an established non-AO language; this usually entails transforming code already provided in that IR [6], a compilation strategy often called *weaving*. A typical example is AspectJ which is compiled to Java bytecode.

Because of that approach, it is possible to use an existing debugger for the underlying non-AO language, like the Java debugger in the case of AspectJ. But a consequence of that weaving approach is that the AO source code is compiled to an IR whose abstractions reflect the module concepts of the so-called base language, but not those of the AOP language. Therefore, what is inspected in the described approach is actually the woven and transformed code instead of the source code.

Other emerging languages with advanced-dispatching (AD) concepts, such as predicate dispatching or many domain-specific languages, share this implementation technique and its limitations. Nevertheless, the identified problems are most significant in AOP languages with their implicit invocation. This is why we focus our study of the state-of-the-art on the wide field of AOP languages, while our solution is applicable more generally to AD languages.

Multiple authors discuss AOP debuggers to provide information closer to the source code, such as the composite source code in Wicca [24] and the aspect-aware breakpoint model in AODA [21]. Nevertheless, all of these debuggers use only the woven IR of the underlying language. AOP-specific abstractions, such as aspect-precedence declarations, and their locations in the source code are partially or even entirely lost after compilation.

While, e.g., the AspectJ language provides runtime-visible annotations that can represent all AO source constructs, these annotations are not suitable to alleviate the above mentioned limitations. Also in the presence of these annotations, bytecode is woven and it is not always possible to retrieve the annotations that have influenced certain instructions during debugging.

In this chapter, we introduce our concept and implementation of a dedicated debugger for AO programs which is able to support locating all types of dynamic AO-related faults identified in previous research, such as that of Ferrari, mentioned above. Our debugger is aware of AO concepts and presents runtime states in terms of source-level abstractions, e.g., pointcuts and advices. It allows programmers to perform various tasks specific to debugging AO constructs. Examples of such tasks are inspecting an aspect-aware call stack, locating AO constructs in source code, excluding AO definitions at runtime, etc. Our debugger is integrated into Eclipse and provides visualizations illustrating, e.g., pointcut evaluation and advice composition.

Our implementation is independent of a concrete source language and provides a generic, default visualization for all AO constructs. While being generic, it still matches the structure of the debugged program; most importantly, all source-level definitions and their dependencies are explicit in our model. To make the

experience of using our debugger even more integrated with the source language used, we offer an extension point for customizing the textual representation in the debugger.

Section 2.2 describes how we generate requirements from existing AOP fault models. Section 2.3 introduces a dedicated advanced-dispatching meta model and how we improve the compilation process to preserve advanced-dispatching information. Sections 2.4 and 2.5 present the underlying debugging model and the user interface of our debugger. Section 2.6 shows how to extend our debugger to customize the visualization in favor of a specific language. Section 2.7 and 2.8 list related works and conclude this chapter respectively.

2.2 Problem Analysis and Requirements

Recently fault models for AOP languages have been investigated with the goal to systematically generate tests that execute all potentially faulting program elements. We can use the results of these studies to derive the capabilities required of a debugger to locate all faults in a program related to (dynamic) features of aspect-orientation. In the following subsections, we summarize the work on AO fault models, discuss tasks required to localize the faults, evaluate the capabilities of existing debuggers, and formulate requirements for a debugger with full support for AOP.

2.2.1 AO Fault Models

We have investigated four fault models—which cover pointcut-advice and inter-type declarations—proposed in the literature and summarize them in Table 2.1. As inter-type declarations change the static structure of a program, identifying faults in them requires different kinds of tools than identifying faults in dynamic features. We focus our study on the dynamic features because the static code inspection tools offered by modern IDEs such the AJDT are already usually sufficient for localizing these faults. For example, a wrongly declared inheritance (**declare parents**) in an aspect can be detected from the editor or the type hierarchy view on Eclipse.

In Table 2.1, the first column shows the fault model by Alexander et al. [2] which contains examples of AOP-specific faults, such as incorrect pointcut strength. Ceccato et al. [56] extend this model with three types concerning exceptional control flow and inter-type declarations (ITD). Ferrari et al. [30] proposed a fault model, presented in the second column, reflecting where a fault originates, i.e., in pointcuts, advices, ITDs or the base program. Column three shows the fault model of Baekken [7] which follows a similar approach; he focuses on AspectJ [47]

programs and systematically considers its syntactic elements as potential fault origins. In the last column, we define a category name summarizing the fault kinds described in literature and presented in the same row.

Alexander et al. (extended by Ceccato et al.)	Ferrari et al.	Baekken	**Category**
	Advice bound to incorrect pointcut	Incorrect or missing composition operator; Inappropriate or missing pointcut reference	**Incorrect pointcut composition**
Incorrect strength in pointcut patterns	Incorrect matching based on exception throwing patterns; Base program does not offer required join points	Incorrect method/ constructor/ field/ type/ modifier/ identifier/ parameter/ annotation pattern	**Incorrect pattern**
	Incorrect use of primitive pointcut designators	Mix up pointcuts method call and execution, object construction and initialization, cflow and cflowbelow, this and target	**Incorrect designator**
	Incorrect matching based on dynamic values and events	Incorrect arguments to pointcuts this/ target/ args/ if/ within/ withincode/ cflow/ cflowbelow	**Incorrect Context**

Alexander et al. (extended by Ceccato et al.)	Ferrari et al.	Baekken	**Category**
Incorrect aspect precedence	Incorrect advice type specification	Incorrect advice type	**Incorrect composition control**
Incorrect changes in control dependencies; Incorrect changes in exceptional control flow (extended)	Incorrect control or data flow due to execution of the original join point; Infinite loops resulting from interactions among advices	Incorrect or missing position of proceed; Incorrect arguments to proceed	**Incorrect flow change**
Failure to establish expected postconditions; Failure to preserve state invariants	Incorrect advice logic, violating invariants and failing to establish expected postconditions		**Violated requirement**

Table 2.1: A systematic and comprehensive fault model for aspect-oriented programs.

2.2.2 Detecting Faults

When a programmer encounters an error during the execution of an AspectJ program, this can be caused by a fault in one of the categories presented in the previous sub-section. But the observed error does not yet tell the programmer what the actual fault is. To figure this out, a debugger may be used. In the following, we discuss tasks to be provided by an ideal debugger for identifying a fault in each of the fault categories. We tag these tasks in the format "**T#**".

If a pointcut-advice definition is faulty, the programmer needs to (**T1**) set a breakpoint at the join point[1], rerun the program, analyze program states, and eventually (**T2**) locate faulty constructs.

[1]In this chapter, we use the term *join point* to refer to a code location (often also called join-point shadow) and to its execution.

Detecting Pointcut-Related Faults

If the programmer finds out that an advice is unexpectedly executed or not executed, she knows that the pointcut evaluated to the wrong value at one join point. To understand the exact cause why the pointcut matches or fails to match, the programmer needs to further (**T3**) evaluate sub-expressions of this pointcut and to check the structure of the pointcut. As the right-most column in Table 2.1 shows, possible causes are *incorrect pointcut composition*, *incorrect pattern*, *incorrect designator*, or *incorrect context*.

Incorrect pointcut composition First, the programmer can consider the correctness of the pointcut structure which may include references to named pointcuts and composition operators. To inspect the actual pointcut expression that is evaluated, pointcut references must be (**T4**) substituted with their definition. To check the composition operators &&, ||, and !, the programmer needs to (**T3**) determine the evaluation result of sub-expressions, perform further evaluations on them and check whether the structure violates the intention.

Incorrect pattern From the above inspection, it may turn out that a pointcut designator like **call** or **get**, which defines a pattern matching a signature, is wrong. Patterns are composed of sub-patterns; thus, the programmer needs to (**T5**) evaluate each sub-pattern to find the actual fault. As an example, consider the AspectJ pattern $*$ Customer.payFor($*$); it matches any method named payFor in the Customer class that takes one argument with any type and returns any type. When debugging the evaluation of that pattern at a join point with the signature **void** Customer.payFor(**int**, **boolean**), a programmer should be able to determine that the parameters sub-pattern causes the pattern to fail.

Incorrect designator The programmer may also determine the fault in a pointcut designator specifying a dynamic condition instead of a pattern, like *target* constraining the type of a runtime value, or *cflow* specifying the currently executing methods. Then the programmer needs to (**T6**) check the runtime values on which the evaluation of that pointcut designator depends; or she must (**T7**) inspect the current control flow, i.e., the join points which are currently executing on the stack.

Incorrect context When a pointcut designator depends on a runtime value and the evaluation result is unexpected, the programmer needs to (**T6**) inspect the context value to which the designator refers and (**T3**) evaluate the restriction on this value specified by the pointcut designator. As an example, consider the pointcut sub-expression **target**(Customer); the callee object is required to be an

instance of the type Customer. The programmer must be able to inspect the value and type of the callee object to determine if the pointcut is specified wrongly or the program uses the wrong object.

Detecting Advice-Related Faults

An error can also occur when an advice is neither missing nor redundant at a join point but the advice does not behave as expected. Possible faults leading to such an error are *incorrect program composition*, *incorrect flow change* and *violated requirements*.

Incorrect program composition There are four types of composition control in AspectJ influencing the execution order of advices at shared join points: advice-type specification, precedence declaration, lexical order, and aspect inheritance. Advice-type specification, e.g., the keywords **before** or **after**, define the order between advices relative to the join point. Precedence declaration defines the partial order between different aspects. The precedence of advices defined in the same aspect is determined by their lexical order. The aspect inheritance implies that advices in the inheriting aspect precede those in the inherited aspect.

To detect incorrect program composition, a programmer needs to (**T8**) inspect how programs are composed at a join point, be able to (**T9**) reason about the composition controls affecting that composition, and (**T2**) locate the definition of the composition controls.

Incorrect flow change The execution of an advice at a join point may alter the control flow or the data flow at that join point. Take the **around** advice as an example: It can skip the join point execution or modify runtime values from the dynamic context of the join point by invoking *proceed*.

To determine which advice is responsible for the wrong control or data flow, the programmer needs to (**T7**) inspect the stack of executing join points including (**T8**) the composition of advices applicable at each join point. To observe data flow, she needs to (**T6**) inspect the runtime values.

Violated requirements Advices may also violate requirements, like post conditions or state invariants, of the modules they apply to. To localize such faults, the programmer may need to (**T6**) inspect runtime values. Another technique often used for localizing faults is to run the program with one or more modules disabled; if the error disappears, the fault most likely lies in the disabled module. To be able to apply this technique, the programmer must be allowed to (**T10**) disable single pointcut-advice pairs, ideally at runtime.

Tag	Task Name
T1	Setting AO breakpoints
T2	Locating AO constructs
T3	Evaluating pointcut sub-expressions
T4	Flattening pointcut references
T5	Evaluating pattern sub-expressions
T6	Inspecting runtime values
T7	Inspecting AO-conforming stack traces
T8	Inspecting program compositions
T9	Inspecting precedence dependencies
T10	(De-)activating AO definitions
T11	Inspecting the history of (de-)activation

Table 2.2: Tasks that an ideal AOP debugger should perform

Dynamic (de-)activation of aspects or advices has the risk of leaving the aspect in a wrong state, e.g., when join points at which the aspect performs an initialization have already passed. This can happen when (de-)activating pointcut-advice manually or programmatically in the source code[1]. (De-)activation can also be performed statically, e.g., in AspectJ, all declared pointcut-advice pairs are deployed before the program is executed. Different (de-)activations may be interleaved and it is confusing to observe the current (de-)activation state without knowing the history. Therefore, programmers must be able to (**T11**) inspect the history of (de-)activation. In this way, when a wrong behavior of an advice is observed during debugging, programmers can (at least in some cases) recognize if this is due to a fault in the program or due to wrong usage of the debugger.

Table 2.2 summarizes the required debugging tasks identified in the previous discussions.

2.2.3 Requirements for an AOP Debugger

Based on the above observations and discussions, we formulate requirements for a dynamic debugger dedicated to AO programs. In the following four sections, we describe how we achieve each of these.

- An intermediate representation must be provided that preserves all AO constructs found in the source code as well as their source locations. Since

[1]For example, the languages JAsCo or CaesarJ support programmatic, dynamic deployment; thus, not all advices are deployed at all times.

many AO languages greatly overlap in their execution semantics, an IR suitable for several languages is desirable.

- A fine-grained debugging interface must be provided to allow observation of and interaction with the execution at the granularity of AO abstractions. The past interactions must be transparent to the users.
- The debugging infrastructure should be integrated with an integrated development environment (IDE) to provide a dedicated user interface on which all tasks listed in Table 2.2 can be performed.
- The information presented to the developer in the user interface should have a representation specific to the concretely used AO language.

2.3 Debugging Information

We chose to base the implementation of the debugger on a generic implementation architecture ALIA4J, which is introduced in Section 1.3.1. This makes our debugger applicable to a wider range of programming languages than AOP. One of the main components of this *ALIA4J* architecture[1] [10] is a meta-model of AD declarations, called *LIAM*[2]. When implementing, e.g., AspectJ in ALIA4J, an advanced-dispatching declaration corresponds to a pointcut-advice definition. A model instantiating the LIAM meta-model is an intermediate representation (IR) of the AD program elements.

For our debugger, we have extended LIAM to store detailed source-location information with every element in the IR. Since ALIA4J keeps the IR as first-class objects at runtime, it can be accessed by our debugger to observe the program execution in an AD-specific way. This fact as well as the declarative and fine-grained nature of LIAM facilitate the support for all identified debugging tasks. We cannot claim that the identified tasks are also fully sufficient when debugging programs written in AD languages which are not AO, since a systematic study of respective fault models is currently missing. Nevertheless, our approach supports at least debugging such language concepts that overlap with AOP.

2.3.1 Compilation Process

In a traditional compilation process, the declarations of AD—such as pointcuts and advices—written in the source code is discarded after transformations like

[1] The Advanced-dispatching Language Implementation Architecture for Java. See `http://www.alia4j.org`.

[2] The Language-Independent Advanced-dispatching Meta-model. See `http://www.alia4j.org/alia4j-liam/`.

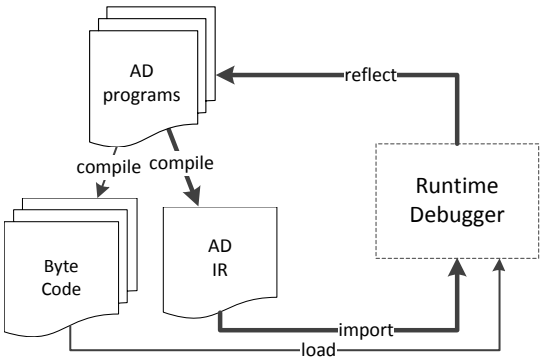


Figure 2.1: Debugging information life cycle

weaving. In result, one source file may be compiled to several compiled files and one compiled file may originate from several source files. The traditional debugger assumes that there is a one-to-one mapping between source files and compiled files. Therefore, it sometimes shows incorrect information in AD programs.

Figure 2.1 shows the compilation strategy used in our approach. Compared to the traditional compilation, there are two differences. First, each source file is compiled to a separate IR file. Thus, the one-to-one relationship is kept. Second, AD declarations written in the source code are stored in a separate AD IR file.

Following the bold directed lines, AD declarations are collected from the source code and then compiled into the AD IR file. At runtime, the AD IR file is interpreted and the program is executed taking the aspect definitions into account. The AD IR can be in any form, e.g. text, or binary. We chose to use XML in our implementation.

This approach requires a specific compiler to generate the IR. In the context of this chapter, we just elaborate on our implementation of an AspectJ compiler based on the *abc* compiler [6]. As an example of the compilation, consider the AspectJ code in listing 2.1. After compilation, it is transformed into an *Attachment* XML element presented in listing 2.2.

There is a many-to-many relationship between source language constructs and LIAM entities. For example, in listing 2.1, the pointcut designator **target**(b) is transformed to two LIAM entities, because it plays two roles: It specifies a dynamic condition under which the pointcut matches a join point (represented by the *AtomicPredicate* in lines 4–12, listing 2.2), as well as a value that is exposed to associated advices (represented by the *Context* in lines 13–15). The pointcut designator, and thus also the atomic predicate, additionally depends on the declaration of the formal advice parameter Base b: The callee object must be an instance of type Base. Thus, the atomic predicate is influenced by two places in the source code and the locations of both places are stored in our IR, as shown

on lines 6 and 10 in listing 2.2.

```
1 aspect Azpect {
2   before(Base b) : call(* Base.foo()) && target(b) { ... }
3 }
```

Listing 2.1: An aspect example in AspectJ

```
1  <attachment language="AspectJ">
2    <specialization>
3      <pattern> ... </pattern>
4      <atomicPredicate type="InstanceofPredicate">
5        <requiredTypeName
6          file="Azpect.aj" line="2" column="9" endLine="2" endColumn="13">
7          test.Base
8        </requiredTypeName>
9        <context type="CalleeContext"
10         file="Azpect.aj" line="2" column="25" endLine="2" endColumn="50">
11       </context>
12     </atomicPredicate>
13     <context type="CalleeContext"
14         file="Azpect.aj" line="2" column="25" endLine="2" endColumn="50">
15     </context>
16   </specialization>
17   <action> ... </action>
18   <scheduleInfo> ... </scheduleInfo>
19 </attachment>
```

Listing 2.2: XML-based AO intermediate representation

Besides <*attachment*> elements for pointcut-advice pairs, the AD IR also contains two more types of elements. The <*precedence*> element corresponds to the statement **declare precedence** and records its location in the source code. The <*inheritance*> element corresponds to aspect inheritance and it takes the line, where the **extends** clauses is declared, as the source location.

With our intermediate representation (IR) presented above, we support the task *locating constructs* (**T2**) presented in Section 2.2.2. Besides locations, we also store the source language in the IR (line 1); in case of multi-language projects, this information can be used to choose appropriate visualizations in the user interface (see Section 2.6 for details). All elements nested in the same *attachment* share the same language attribute. The usage of the language attribute is described in Section 2.6.

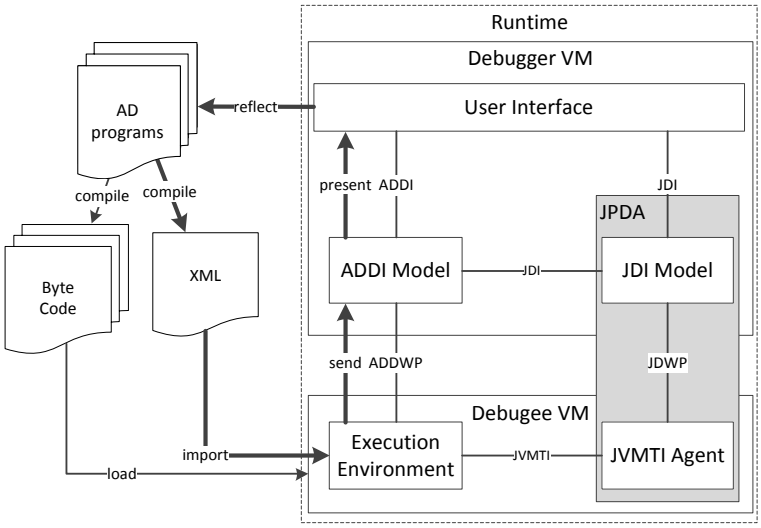


Figure 2.2: The architecture of our interactive AD debugger

2.4 Infrastructure

Extending Figure 2.1, the overall structure of our debugger is presented in Figure 2.2. It consists of a debuggee side and a debugger side; both sides communicate via the *Java Platform Debugger Architecture* (JPDA)[1] and the Advanced-Dispatching language Debugging Wire Protocol (ADDWP). The debuggee-side virtual machine runs the debuggee program and sends debugging data and events via the two channels. Our user interface (debugger side) presents this information and provides controls to the programmer to interact with the debuggee. These controls are implemented by using the Java Debug Interface (JDI) and the Advanced-Dispatching Debug Interface (ADDI). As our debug interface is based on ALIA4J's meta-model of advanced dispatching, we reuse that terminology in our infrastructure, even though our case study is based on AspectJ.

The ADDWP is implemented as two agents running on the debugger and debuggee sides, respectively. It has a similar structure and working mechanism as the JDWP but sends and receives AD-specific information. The following subsections describe the execution environment and the ADDI in detail. The UI is explained in the next section.

2.4.1 Debuggee Side

In the ALIA4J approach, an execution environment is an extension to a Java Virtual Machine (JVM). The extension allows deploying and undeploying LIAM

[1] See `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

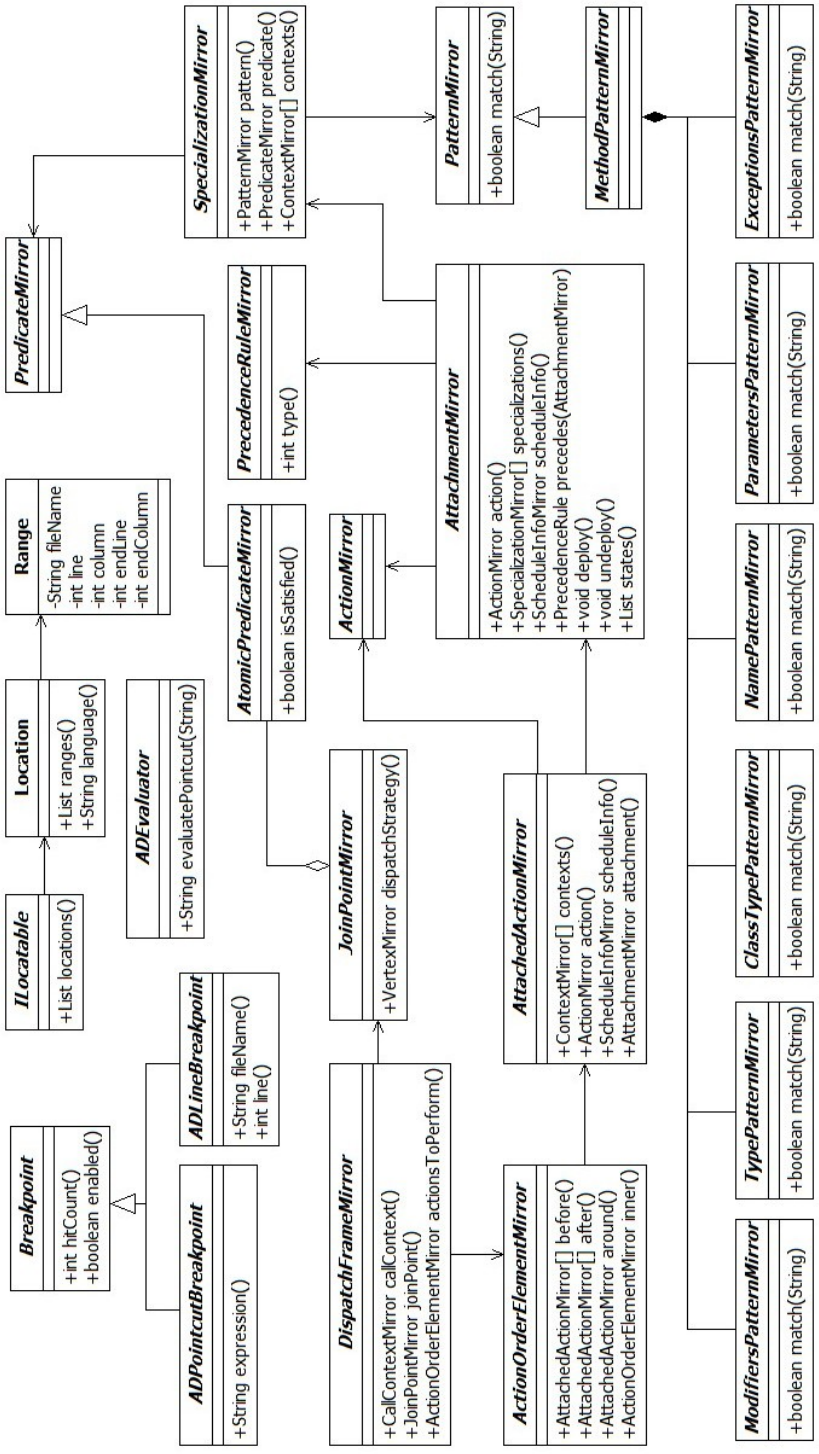


Figure 2.3: A simplified UML class diagram of the Advanced-Dispatching Debug Interface

dispatch declarations and derives an *execution strategy* per call site that considers all dispatch declarations present in the program.

The execution strategy consists of the so-called dispatch function (for details see Sewe et al. [76]) that characterizes which actions should be executed as the result of the dispatch in a given program state. This function is represented as a binary decision diagram (BDD) [15], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with the actions to be executed. For each possible result of dispatch, the BDD has one leaf node, representing an alternative result of the dispatch, i.e., which actions to execute and in which order.

Our current implementation of the debugger is based on the ALIA4J NOIRIn execution environment [10], which is implemented as a Java 6 agent intercepting the execution of the base program to perform the dispatch. NOIRIn can integrate with any standard Java 6 JVM, therefore our approach does not require using a custom virtual machine.

2.4.2 Advanced-Dispatching Debug Interface

The Advanced-Dispatching Debug Interface (ADDI) is the debugger-side interface of the debugging infrastructure. It provides various functionalities to perform the tasks identified in Section 2.2.2, and implements them in collaboration with the debuggee virtual machine. A simplified UML class diagram of ADDI is presented in Figure 2.3.

The Java Debug Interface (JDI) provides mirrors for every runtime entity in a Java program, like objects, classes, or threads. The ADDI extends the JDI by additionally providing mirrors for the LIAM entities, which exist in the debuggee virtual machine and represent the pointcut-advice definitions. Since LIAM entities are plain Java objects, the ADDI mirrors are implemented by aggregating the JDI mirrors of those objects.

ADDI's breakpoints do not wrap the breakpoint event provided by the JDI. When a breakpoint is set, the debugger-side sends the breakpoint information to the execution environment at the debuggee side. The execution environment registers a breakpoint event according to the received information. When a registered breakpoint event occurs, the execution environment sends the JDI command for suspending the virtual machine. Below, we discuss the top-level mirrors of the ADDI:

Breakpoint reifies breakpoints that can be set at join points (**T1**). There are currently two different ways for specifying join points, by specifying a valid pointcut expression (ADPointcutBreakpoint) and by specifying a line location (ADLineBreakpoint). ADPointcutBreakpoint matches all join points satisfying

its expression and ADLineBreakpoint matches all join points on the specified line.

ILocatable is an interface for locating entities. Multiple equivalent AD IR elements applicable at the same join point are represented by a single entity by NOIRIn for performance reasons. Therefore, the *locations()* method returns a list of all source locations a runtime entity may originate from. A Location consist of one or more *ranges*, i.e., positions in a file (see Section 2.3.1). In contrast to the AD IR, which stores the source language information of many entities jointly in an Attachment, to simplify the access, in the ADDI this information is provided through the locations of an entity. ActionMirror, AtomicPredicateMirror, PatternMirror, AttachmentMirror, DispatchFrameMirror, and PrecedenceRuleMirror implement this interface. Thus, corresponding constructs can be located in the source code (**T2**).

ADEvaluator can perform evaluation on given pointcut expressions or subexpressions (**T3**). It takes strings as input, and sends them to the backend. The back-end compiler compiles received strings into LIAM entities, evaluates their value according to the current program state and returns the result to the debugger side. If the expression is syntactically incorrect, an error message is returned.

DispatchFrameMirror reifies a stack frame containing the execution strategy at a join point (**T7**). It provides inspection of the call context (**T6**) and of the program composition (**T8**) at the current join point.

AtomicPredicateMirror reifies primitive pointcut sub-expressions (**T3**).

ActionOrderElementMirror reifies the program composition (**T8**). It consists of four parts, namely *before*, *after*, *around*, and *inner*. The before, after, and around parts point to advices (respectively the action representing the join point operation) which are sequentially executed at a join point. The inner part refers to the actions to be executed when the *around* advice performs the *proceed* operation.

AttachmentMirror first provides access to the three parts of an attachment declaration (corresponding to a pointcut-advice): action, specialization (corresponding to the pointcut) and schedule information. Second, it can be activated or deactivated at runtime (**T10**). This mirror also stores the history of (un-)deployments in the *states* list (**T11**). A history record contains information whether the (un-)deployment was performed manually through the debugger, or programmatically in the source code, or statically. In the

case of programmatic (de-)deployment, additionally, the source location is stored.

PrecedenceRuleMirror reifies the ordering relations between attachments (**T9**). The *type* of a precedence rule represents how it was specified: in AspectJ, precedence can be defined through the **declare precedence** statement (*declared*), through the **before**, **after** or **around** keywords (*implied*), through the lexical order of advice definitions (*lexical*), and through aspect inheritance (*inherited*).

SpecializationMirror reifies static and dynamic sub-expressions of pointcuts which are decomposed into a pattern, a predicate, and contexts.[1] Referenced named pointcuts are resolved and inlined in the specialization (**T4**).

PatternMirror can be used to perform evaluations to patterns used in pointcuts. As illustrated by the example of method patterns in Figure 2.3, patterns consist of smaller sub-patterns which are separate entities in ADDI and can be evaluated respectively (**T5**).

2.5 User Interface

The front-end of our debugger is integrated into the Eclipse IDE, although any IDE with a comparable infrastructure would also be applicable. Our AD debugger extends the Eclipse Java debugger with additional user interfaces. These are Eclipse views specific to visualizing and interacting with ALIA4J's representation of pointcut-advices in order to support the tasks discussed in Section 2.2. The developed debugger provides four new views, namely the *Join Point* view, the *Attachments* view, the *Pattern Evaluation* view, and the *Advanced Breakpoints* view.

Throughout this section, we illustrate the functionalities of our debugger by means of an example AspectJ program. Listing 2.3 shows the base program whose actions are advised by the aspect in listing 2.4. There are four advices (on line 5, 8, 12, and 15, listing 2.4) declared in Azpect. Suppose the program is currently suspended at line 16 of listing 2.4. We introduce each view in this scenario in the following sub-sections.

[1] See Bockisch et al. [9] for a detailed discussion of how to transform any AspectJ pointcut to our data structure.

```
1  package test;
2  public class Base {
3    private int someField;
4    public static void main(String [] args) {
5      Base b = new Base();
6      b.normalMethod();
7    }
8    public void normalMethod() {
9      advicedMethod();
10   }
11   public void advicedMethod() {
12     someField = 1;
13   }
14 }
```

Listing 2.3: An example base program

```
1  package aspects;
2  import test.Base;
3  public aspect Azpect {
4    pointcut base() : call(* Base.advicedMethod());
5    before() : base() && target(Base) {
6      System.out.println("before-target");
7    }
8    Object around() : base() {
9      proceed();
10     return null;
11   }
12   before() : base() && !target(Base) {
13     System.out.println("before-!target");
14   }
15   after() : set(* Base.someField) {
16     System.out.println("after-set");
17   }
18 }
```

Listing 2.4: An example aspect

2.5.1 Join Point View

The *Join Point* view is the central view of the debugger showing runtime information about the join point at which the debuggee is currently suspended. A snapshot of the Join Point view is given in Figure 2.4.

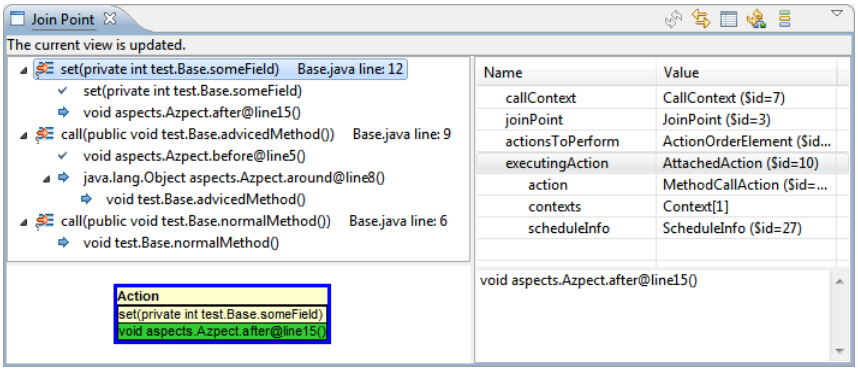


Figure 2.4: A snapshot of the Join Point view

Structure of the Join Point View The view has several parts to allow the programmer interacting with the debuggee. The top left panel displays the stack of join points that are currently executing when the debuggee is suspended. Each explicit invocation—whether selected by a pointcut or not—is represented as one row in the stack trace. For each join point, the signature and the source location of the corresponding join-point shadow are presented (**T7**). By unfolding a join point, corresponding applied actions can be inspected. Thus, the join point stack covers all information also presented in the standard stack trace, plus additional information about advice application.

Actions are organized as the structure of the program composition at each join point (**T8**). Take the second frame representing a call to the advicedMethod for example, it sequentially executes a before advice and an around advice with a nested execution to the advicedMethod. We divide actions into three types which are executed, executing, and to be executed and use tick, arrow, and exclamation mark icons to tag them respectively.

Locating an entity consists of two steps. First, double-clicking an item, like a label representing an action in the stack, activates a *location window* which is shown in Figure 2.5. The window contains a list of items which represent different locations the corresponding entity has. Each location is described by its file name and ranges, like "(5,25)–(5,36)" where the four numbers represent the start row, the start column, the end row, and the end column respectively. Second, the editor highlights the source code ranges when the user double-clicks one of the listed items (**T2**). If there is only one possible location, the location window is not opened, but the corresponding source range is immediately highlighted.

The bottom-left panel gives a graphical representation of the execution strategy for the join point selected in the top-left panel (**T8**). Each label represents an action that has been executed, is executing, or will be executed at this join point. Figure 2.4 displays one composition with two sequential actions which are a field

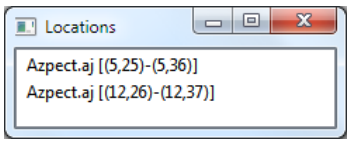


Figure 2.5: A snapshot of the location window

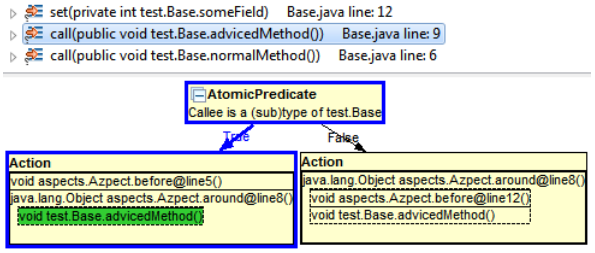


Figure 2.6: A graphical representation of dispatch

assignment and an advice execution. In AspectJ, advices do not have names. Therefore, we chose to use the name of the aspect and the line number where an advice is defined to uniquely identify the advice, like Azpect.after@line15(). The label with green (highlighted) background indicates that the action it represents is currently executing.

The top-right panel of the Join Point view uses a tree viewer to show all context values needed to evaluate the join point's execution strategy and exposed to the actions (**T6**). The bottom-right panel gives a string description of the item currently selected in the tree view.

Graphical Representation of Dispatch The graphical representation of a join point visualizes the execution strategy applied by the ALIA4J execution environment and allows navigating to the corresponding definitions in the source code. For illustration, consider that the second frame is selected in the example. Figure 2.6 shows the join point visualization for this case.

This graphical representation consists of an *AtomicPredicate* testing whether the callee object at this call site is an instance of test.Base and two *Action* nodes with different program compositions according to the evaluation result of the *AtomicPredicate* (**T3**). The blue (bold) path indicates the evaluation result of the atomic predicates and the composition of actions to be performed at the current join point. The highlighted *Action* node first performs Azpect.before@line5() and then Azpect.around@line8(); when the latter proceeds, Base.advicedMethod() is executed. The dashed box surrounding Base.advicedMethod() visualizes the fact that the execution of *proceed* cannot be decided until it is actually performed. Double-clicking on a label representing an atomic predicate or an action reveals

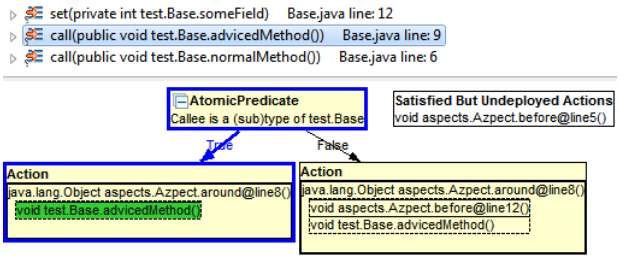


Figure 2.7: A graphical representation of dispatch with a node showing "Satisfied But Undeployed Actions"

the source location or, if multiple locations are possible, invokes the location window (**T2**).

If more complex pointcuts apply to this join point, i.e., more atomic predicates are evaluated, the size and complexity of the BDD may grow significantly. To reduce the presented information the "-" icon in labels representing atomic predicates can be clicked to collapse subtrees. Furthermore, a more compact tabular representation of the execution strategy is available as detailed below.

We provide additional information to show the potential influence of currently undeployed attachments in the graphical representation of dispatch. Suppose, the same join point occurs as explained above and the attachment with the action to call Azpect.before@line5() is defined in the program, but was not deployed. The graphical representation of dispatch in this scenario is shown in Figure 2.7. Compared to Figure 2.6, there is an additional node with the title "Satisfied but Undeployed Actions" which lists all actions that would have been applied at the current join point if the corresponding attachments were deployed (**T11**); more details about **T11** is given in Section 2.5.2. Double-clicking an action can perform locating (**T2**).

Textual Representation of Dispatch By clicking the "Table" button on the toolbar, the bottom-left panel is switched to a table, as shown in Figure 2.8. This table contains several pieces of information to support **T3** and **T8**: First it lists all actions that are potentially applicable at this join point, i.e., the standard join point action (Base.advicedMethod()) and all advices whose pointcut statically matches the join point.

Second, for all actions whose pointcut dynamically matches the join point, the execution sequence and nesting levels (for *around* actions) are shown. For example, "2.1" for Base.advicedMethod() means that this action is executed as the first action when the second action from the level above (advice Azpect.around@line8() numbered with 2) performs *proceed*. Similar to the graph representation, the currently executing action is highlighted with green background. For those actions

Order		Evaluation
◢ 1	void aspects.Azpect.before@line5()	
	Callee is a (sub)type of test.Base	true
2	java.lang.Object aspects.Azpect.around@line8()	
2.1	void test.Base.advicedMethod()	
◢ X	void aspects.Azpect.before@line12()	
	NOT (Callee is a (sub)type of test.Base)	false

Figure 2.8: A textual representation of dispatch

whose pattern statically matched, but where the dispatch function determined that they are not applicable at this call, the table shows an 'X' in the order column.

Third, the table shows the results of all atomic predicates of pointcuts that are evaluated at this join point. Compared to the graphical representation, the table does not show the process of evaluation and other possible program compositions.

Visualization of Precedence Dependencies To reason about the composition of advices at a join point (**T9**), the precedence relationships between the advices are visualized. To illustrate how the visualization of precedence dependencies works, we use four additional aspects which are shown in listing 2.5. Three aspects, PrecedingAzpect, AbstractPrecededAzpect, and PrecededAzpect, declare a **before** advice. Among them, PrecededAzpect extends AbstractPrecededAzpect. The aspect IrrelevantAzpect defines the precedence between PrecedingAzpect and PrecededAzpect.

```
1  package aspects;
2  import test.Base;
3  aspect PrecedingAzpect {
4    before() : call(* Base.advicedMethod()) { ... } }
5  abstract aspect AbstractPrecededAzpect {
6    before() : call(* Base.advicedMethod()) { ... } }
7  aspect PrecededAzpect extends AbstractPrecededAzpect{
8    before() : call(* Base.advicedMethod()) { ... } }
9  aspect IrrelevantAzpect {
10   declare precedence : PrecedingAzpect, PrecededAzpect;  }
```

Listing 2.5: Aspect illustrating precedence dependencies

Consider that the execution is suspended at the call to advicedMethod() at line 9, Listing 2.3. By clicking the "Precedence" button on the toolbar of the Join Point view, the graph panel changes to a representation of the precedence dependencies as shown in Figure 2.9.

Labels representing actions are numbered and connected by directed lines. The direction of a connection indicates the precedence between two actions. We

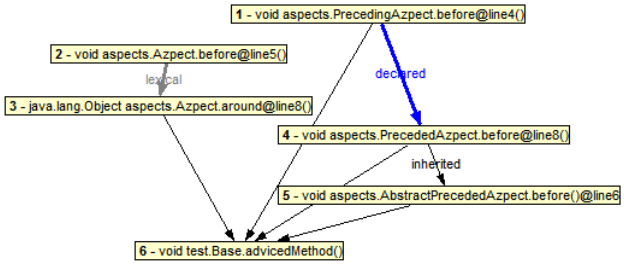


Figure 2.9: The graphical representation of precedence dependencies

use the numbers as substitute for action names in the following paragraph; for example, "action 2" represents Azpect.before@line5().

There are four types of connection representing the types of precedence rules distinguished in ADDI: Precedence may be declared explicitly by means of the **declare precedence** statement, visualized by a bold blue (dark) connection labeled with "declared"; it may be defined by the *lexical order* of advice definitions in the same aspect, visualized by a bold gray (light) connection labeled with "lexical"; it may be implied by the aspect inheritance, visualized by a connection labeled with "inherited"; or it may be determined by the kind of action (i.e., **before**, **after**, **around** advice or the join point action), visualized by a connection without label.

The "declared" precedence is explicitly declared in source, like line 10 in listing 2.5. For the "inherited" precedence, the **extends** clause is the source location, like line 7 in listing 2.5. The location is revealed when the corresponding connection is double-clicked (**T2**). An example of precedence declaration by means of lexical order is shown in listing 2.4: Action 2 is declared on line 5 and, thus, precedes action 3 defined on line 8. The precedence between any two actions without a connection is not specified, such as action 1 and action 2. Therefore, the execution order of the two actions is random at runtime.

2.5.2 Attachments View

In order to dynamically (un-)deploy attachments during runtime, the *Attachments* view is provided. A snapshot of the *Attachments* view is given in Figure 2.10. The top panel shows textual representations of all attachments that are defined in the executing program. Unchecking or checking one of the items will lead to undeployment or deployment of the corresponding attachment in the debugged program (**T10**) and change the state of the attachment accordingly. The middle panel lists the deployment history of the selected attachment in the reversed chronological order. Whether an attachment was (un-)deployed statically ("Statically"), or by the source code ("By code"), or manually through the debugger ("By debugger") is shown in the third column (**T11**). If an (un-)deployment

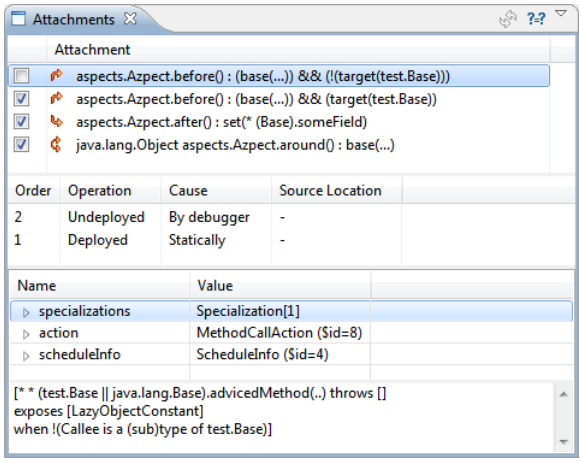


Figure 2.10: A snapshot of the Attachments view

is performed explicitly by the source code, double-clicking the item can highlight the corresponding code. The bottom panel presents details of the selected attachment.

In Figure 2.10, the first attachment, representing the *before* advice declared on line 5 in listing 2.4, is selected. This advice has a pointcut containing a reference to another pointcut declared on line 4. The *Specialization* of the selected attachment describes the related pointcut in the bottom panel and the referred pointcut is inlined in the description (**T4**).

2.5.3 Pattern Evaluation View

To debug patterns used in pointcuts, we visualize the pattern evaluation at the granularity of sub-patterns specified for the separate parts of the join-point signature. Since patterns that do not match at a join point are not shown in the Join Point view, this functionality is accessible through the Attachments view which contains all pointcut-advice definitions in the program.

For illustration suppose we select the third frame representing the call to method test.Base.normalMethod() in Figure 2.4. We find that the **before** advice declared on line 5 in listing 2.4 does not appear in the execution strategy. That means the pattern used in the **before** advice is unsatisfied. To evaluate the method signature against the pattern, we use the item representing the **before** advice in the *Attachment* view. Then, an evaluation result of each sub-pattern is presented in the *Pattern Evaluation* view as shown in Figure 2.11. It gives the evaluation results for each sub-pattern (**T5**).

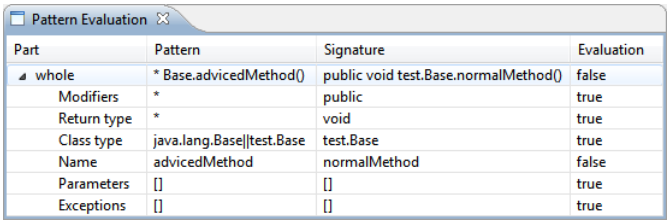


Figure 2.11: A snapshot of the Pattern Evaluation view

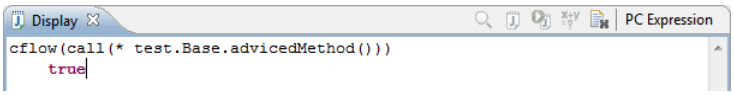


Figure 2.12: The extended Display view for evaluating pointcut expressions

2.5.4 Extended Display View

The pointcut evaluation provided in the *Join point* view shows only expressions existing in the source code. The programmer is unable to test a new pointcut expression unless she modifies and reruns the program. To provide more flexibility in evaluating pointcut expressions (**T3**), we extended the *Display* view. For example, suppose the second frame shown in Figure 2.4 is selected, the programmer evaluates the expression **cflow**(**call**(∗ test.Base.advicedMethod())). The result is shown in Figure 2.12.

2.5.5 Advanced Breakpoints View

We added the *Advanced Breakpoints* view, as Figure 2.13 shows, to allow setting breakpoint at pointcuts (**T1**). We currently provide two types of breakpoints which are line-based and expression-based. In Figure 2.13, the first two breakpoints are line-based and the last two are expression-based.

Setting a line-based breakpoint requires the programmer to select a line in the editor and then use the view to add a breakpoint. The program will be suspended at all join points on this line during debugging. For example, the first breakpoint is set at the line 6 in listing 2.3. There is only one join point on this line—calling method normalMethod. Therefore, when this method is called on this line, the program is suspended.

Setting an expression-based breakpoint requires the programmer to input a valid pointcut expression. The program will be suspended on all join points satisfying this expression. If the input expression is not valid, the corresponding breakpoint does not take effect.

Both advanced breakpoints and conventional breakpoints can an be used in the same debugging session with our debugger. However, some AD and con-

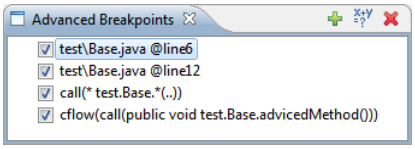


Figure 2.13: A snapshot of the Advanced Breakpoints view

ventional debugging facilities cannot be used if the program is suspended in an unexpected context. When a conventional breakpoint is hit, the AD debugger cannot recognize the suspension place as a join point. Therefore, the *Join Point* view, the *Pattern Evaluation* view, and the extended *Display* view cannot show valid AD information. The *Attachments* view and the *Advanced breakpoints* view can still be used, because their presented information is joinpoint-independent. Similarly, when an advanced breakpoint is hit, the execution is suspended in infrastructure code. Thus, the conventional views, such as the *Variables* view and the *Stack* view, show the debugging information of the infrastructure code instead of the source code.

2.6 Customization of Visualizations

Our debugger is built based on the meta-model *LIAM* which supports many different, advanced-dispatching languages; thus it can be applied to programs written in several languages. While these languages have some overlap in their semantics, they may significantly differ in the syntax. Table 2.3 lists four definitions with the same meaning, but written in different languages. This includes two AOP languages (AspectJ and JBoss AOP[1]), and two predicate-dispatching languages (MultiJava [17] and JPred [61]). All statements specify the dispatch of a call to method Shape Shape.intersect(Shape) in which the first argument should be an instance of type Circle.

Our debugger renders the same presentations for these four languages if no specific customization is provided. But programmers become less productive if descriptions from the debugger do not resemble the source code. In this Section, we describe how to extend our debugger with language-specific customizations (Section 2.6.1), how to choose a customization at runtime (Section 2.6.2), and how to construct customized descriptions (Section 2.6.3).

[1]See `http://www.jboss.org/jbossaop`

Language	Syntax
AspectJ	**call**(Shape Shape.intersect(Shape)) && **args**(Circle)
JBoss AOP	**call**(Shape Shape−>intersect($instanceof{Circle}))
MultiJava	Shape Shape.intersect(Shape@Circle s)
JPred	Shape Shape.intersect(Shape s) **when** s@Circle

Table 2.3: Same dispatching restrictions expressed in different languages.

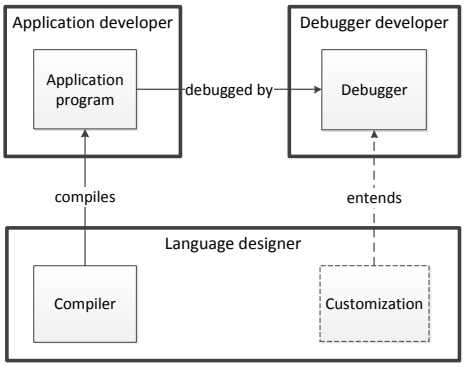


Figure 2.14: Components which are used in debugging are developed by different parties

.

2.6.1 Customizing the Presentation of an Entity in a Modular Way

Figure 2.14 shows the relationships between participants in a debugging session. The top-left part which contains the debuggee programs is developed by application developers, who are also debugger users. The top-right part contains our debugger which takes the compiled application as input and provides interfaces for customizations. The bottom part, which contains a complier and a customization extension for the debugger, is developed by language designers. The dashed line indicates that the customization is not mandatory for debugging.

Our debugger is implemented as Eclipse plug-ins. Eclipse uses the mechanism of extensions and extension points to allow incrementally implementing functionalities in separate plug-ins. Extension points are declared by the extended plug-in and they define contracts how other plug-ins should connect to it.

Figure 2.15 shows how a customization works with our debugger. The debugger component contains two plug-ins. The UI plug-in (*org.alia4j.addb.ui*) im-

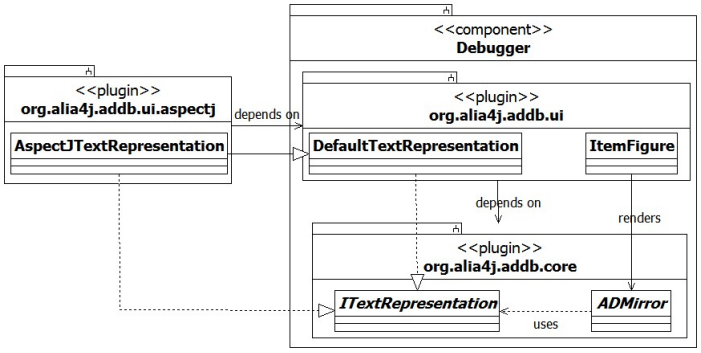


Figure 2.15: The plug-in structure of our debugger. An extension point defines that a customization extension needs to realize the interface *ITextRepresentation*

.

plements the user interfaces. The model plug-in (*org.alia4j.addb.core*) contains the ADDI implementation. The model plug-in provides an interface *ITextRepresentation*, which defines a list of displaying functions for *ADMirror*s. The UI plug-in contains widgets such as *ItemFigure*, which render text representations of *ADMirror*s, and a default implementation of the *ITextRepresentation*. The *DefaultTextRepresentation* is language-independent and it describes entities in a way how LIAM models AD concepts. Texts presented in previous UI snapshots are provided by *DefaultTextRepresentation*.

Each extension point has an identifier and it declares several attributes that its extensions should have. In our implementation, the extension point requires the name of the class that implements ITextRepresentation and the name of the source language to which the customization is applicable. Listing 2.6 shows an extension declaration defined in the plug-in *org.alia4j.addb.ui.aspectj*. The declaration first refers to the extension point by using the identifier (line 1) and then specifies the realizing class and the language name (lines 3 and 4).

```
1 <extension point="org.alia4j.addb.text.display">
2   <TextCustomization
3     class="org.alia4j.addb.ui.aspectj.AspectJTextRepresentation"
4     language="AspectJ">
5   </TextCustomization>
6 </extension>
```

Listing 2.6: An extension declaration for AspectJ textual customization

At runtime, we use the *extension registry* provided by the Eclipse platform to retrieve all desired extensions by specifying the identifier of the extension point. We store the language name and an instance of the customization in a hash map.

2.6.2 Choosing a Customization for an Entity

A Multi-Language Example.

Our debugger can be used for projects written in multiple AD languages. We use the motivating example from the paper introducing *AwesomeDebugger* [4]—which also identified debugging multi-language programs as a relevant problem—to illustrate this. We show this example program in listing 2.7. For brevity, we only show code related to one join point shadow and AD definitions that are applied at that join point shadow. The listing contains three AD units which are written in three different languages respectively: The **aspect** (line 4) is written in *AspectJ*, the **coordinator** (line 7) is written in *Cool*, and the **validator** (line 12) is written in *Validate*.

```
1  public class Stack {
2    public void push(Object obj) { ... }
3  }
4  public aspect Tracer {
5    before() : !cflow(within(Tracer)) { ... }
6  }
7  coordinator Stack {
8    condition full=false;
9    push : requires !full;
10   on_exit { ... }
11 }
12 validator Stack {
13   validate push(Object obj) { ... }
14 }
```

Listing 2.7: A multi-aspect-language example.

When the program in listing 2.7 is suspended at the execution of Stack.push(), AD actions declared at line 5, 10, and 13 may be performed. To illustrate possible program compositions at this join point shadow, Figure 2.16 shows a graphical representation of the execution strategy and a label describing the join point shadow (top left corner). Labels corresponding to elements from AD definitions are tagged with language information: "A" is for *AspectJ*, "C" is for *Cool*, and "V" is for *Validate*. The language information means that the corresponding entity is referred or used in the program written in that language. For example, the atomic predicate *cflow(...)* at the top of the execution strategy is used in the AspectJ program.

In NOIRIn, equivalent entities, which, e.g., originate from different source languages are only evaluated once at the same join point for performance reasons.

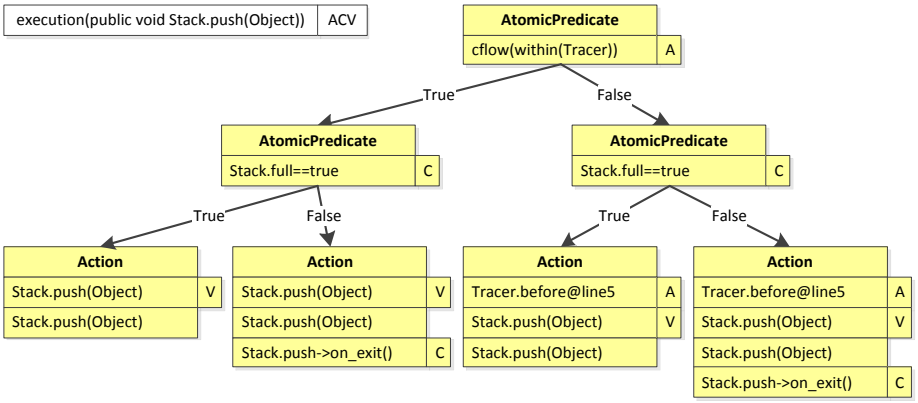


Figure 2.16: Graphical representations tagged with language information

.

Such LIAM entities are *AtomicPredicate*, *Context*, and *Pattern*. To design a neat user interface, we also show the join point, which may be referred by different source languages, only once in the stack trace of the *Join Point* view.

If an entity from one language is rendered by a customization for another language, it is confusing for debugger users. Take *Cool* for example, the method execution is the only available join point. Therefore, language designers may omit the "execution" keyword in describing a join point. This becomes confusing for AspectJ programmers, because advices can be applied not only at method executions but also at method calls. Therefore, when multiple customizations are required, choosing which one and where to apply the chosen one are the main challenges.

Three Customization Approaches.

We can apply a customization for entities either globally or locally. Global strategy means that all entities use the same customization, which can be either language-independent or language-specific. Local strategy means that entities from different source languages use different customizations. We discuss three feasible approaches in the following paragraphs.

Local Customization. This approach uses customizations locally. Recall the ADDI in Figure 2.13: Each location of an entity has a source language name. Therefore, all source languages of an entity can be read from its location information. If an entity has only one source language, the debugger can automatically choose the customization. For those with multiple source locations with different languages, the debugger uses the first available customization. For rectification, programmers may need to manually choose a

Join Point Shadows	Count	Details
execution(Stack.new(..))	1	1AV, 4A
execution(Stack.push(..))	5	1ACV, 7A, 4C, 4V
execution(Stack.pop())	5	1AC, 7A, 6C

customization by using widgets, such as a context menu.

Global Default Customization. This approach is globally applying the default customization which provides sufficient information describing the AD semantics. The default customization uses LIAM terms, which are language-independent. For example, it uses "action" for "advice" and "callee" for "target".

Global Specific Customization. This approach requires debugger users to manually choose a language-specific customization and the chosen customization is applied globally.

An Evaluation of the Three Approaches.

To evaluate the aforementioned three approaches, we analyse the full example used in [4]. The program creates a stack, pushes five elements to the stack, and then pops five elements from the stack. We are interested in only the 6 common types of join points within class Stack, which are constructor-call/execution, method-call/execution, and field-get/set. To find out the precision of the three customization approaches, we count the number of LIAM entities shared between languages at the join points of this program.

There are 69 join points in total and all of them are advised by the AspectJ program. Among them, 11 join points are shared by AD definitions written in at least two languages. The 11 shared join points, which originate from 3 join point shadows, are described in the table below. The "Count" column shows the number of join points corresponding to the join point shadow. The "Details" column specifies the statistics about entities and their source languages at each join point. For example, "1AV" means that there is 1 entity from the *AspectJ* program and the *Validate* program.

The "local customization" approach maximally restores the language-specific descriptions for rendered entities. 97.5% entities at all 69 join points are definitely shown by the appropriate customizations, because each of them has only one source language. To choose the right customization at the shared join points, six join points have entities shared between two languages and thus, require at most 1 manual configuration of the used customization. Five join points have

	Precision	Number of Configurations	Required Implementations
Local Customization	97.5%	< 16	3
Global Default Customization	-	0	0
Global Specific Customization (AspectJ)	82.5%	< 16	> 0

Table 2.4: A comparison between different approaches applying customizations in a multi-AD-language example.

entities shared between three languages, requiring at most 2 configurations. This approach requires that all 3 language-specific customizations are implemented.

The "global default customization" approach reduces the comprehensibility of the representations, because it requires the programmers to get familiar with the mappings from LIAM concepts to the constructs of each specific source language. We do not have exact statistics to quantify to what extent LIAM terms decrease the comprehensibility. The advantages of this approach is that it does not require any customization configuration and implementation.

The "global specific customization" approach needs at least one language-specific customization. We assume that the debugger users use the AspectJ customization, because 82.5% entities are not shared between languages, but only come from the AspectJ program. At the shared join points, if all customizations are available, the maximum number of configurations is the same as for the "local customization".

Table 2.4 summarizes the above discussion. Column 2 and 3 reflect the comprehension and configuration effort spent by the debugger users. Column 4 shows the implementation effort spent by the language designers. Overall, there is no "best" approach that is superior to the other two. The first approach is the most accurate one. The second approach requires the least effort in configuration and implementation. The third approach provides relatively high accuracy without much implementation effort.

According to the comparison above, we have chosen the third approach, which is a trade-off between the other two approaches, in our implementation. We provide a widget which lists all available customizations and programmers can manually select which one to use.

2.6.3 Constructing Descriptions for Entities

A text description may be determined by different factors including the source language, the entity class, the runtime values, and the entity contexts. The source language determines which customization to choose. The entity class determines which specific displaying method to invoke. The runtime values provide contents to the description. However, the description may vary in a different entity context.

The interface ITextRepresentation, as shown on lines 1–6 in listing 2.8, defines a list of functions displaying different *ADMirror*s. Language designers need to realize ITextRepresentation and implement concrete displaying methods.

The DefaultTextRepresentation basically calls mirrorString for each individual displaying method, such as line 10. The method mirrorString returns the textual description of the mirrored object on the debuggee side. By extending DefaultTextRepresentation, language designers only need to override those displaying methods which are different from the default implementation.

The runtime values related to the displayed entity may significantly affect what the textual description looks like. Suppose the method declared on line 15 is executed; that means the entity to be rendered is an InstanceofPredicateMirror written in AspectJ. Line 18 tests if the context is an ArgumentContextMirror. In InstanceofPredicateMirror, the syntax of a value is tied to the kind of value that is tested. For example, its syntax starts with **args** for the arguments context and with **target** for the callee context. Lines 19–30 give a specific implementation for the arguments context. In this case, the AspectJ syntax also requires to encode the position of the tested argument. For this reason the rendered text depends on the index of the restricted argument; the comments on lines 23 and 26 show examples.

The textual representation of the same entity may be different if it is in a different context. Suppose the source code is **args**(Circle, Rectangle), it is transformed to two *InstanceofPredicateMirror*s at runtime. The debugger shows **args**(Circle, ..) and args(∗, Rectangle, ..) separately if it uses the method on lines 15-34. To construct texts for more coarse-grained entities, like a *SpecializationMirror* which contains the args expression, merging the two descriptions is necessary.

Sometimes, rendering part of the source code may lead to loss of semantics. Take the MultiJava expression on row 4 in Table 2.3 for example, the whole expression requires the first argument to be an instance of type Circle. The textual description of the corresponding InstanceofPredicateMirror is "Shape@Circle" which misses the index of the argument. Our solution is to add auxiliary information to variable names, like "Shape@Circle arg#1" that uses "#1" to indicate the index of the argument.

```
1  public interface ITextRepresentation {
2    public String display(ArgumentContextMirror mirror);
3    public String display(AttachedActionMirror mirror);
4    public String display(MethodCallActionMirror mirror);
5    ...
6  }
7  public class DefaultTextRepresentation implements ITextRepresentation {
8    @Override
9    public String display(InstanceofPredicateMirror mirror) {
10     return mirror.toString();
11   }
12 }
13 public class AspectJTextRepresentation extends DefaultRepresentation {
14   @Override
15   public String display(InstanceofPredicateMirror mirror) {
16     ContextMirror context = mirror.context();
17     StringBuffer sb = new StringBuffer();
18     if(context instanceof ArgumentContextMirror) {
19       ArgumentContextMirror argsCtx =
20                   (ArgumentContextMirror) context;
21       sb.append("args(");
22       int index = argsCtx.index();
23       if(index >= 0) { // args(*, *, Clazz, ..)
24         for(int i=0; i<index; i++) { sb.append("*, "); }
25         sb.append(mirror.requiredType()).append(..);
26       } else {   // args(.., Clazz, *, *)
27         sb.append("..,").append(mirror.requiredType());
28         for(int i=index+1; i<0; i++) { sb.append(", *"); }
29       }
30       sb.append(")");
31     } else if(context instanceof CalleeContextMirror) { ... }
32     ...
33     return sb.toString();
34   }
35 }
```

Listing 2.8: The interface *ITextRepresentation* and its two implementations

2.7 Related Work

The related work basically falls into three parts which are debuggers for AOP languages and other development tools for AOP languages. In the following

subsections we present tools in these categories and discuss them according to the requirements listed in this chapter.

2.7.1 Debuggers for Aspect-Oriented Languages

The Java debugger is the most commonly used tool for debugging AspectJ programs which are compiled to pure Java bytecode. Some elements of the aspect definition are partially evaluated during compilation and drive a series of code transformations applied to the aspect and non-aspect modules. Thus, there is no one-to-one mapping between elements in the source code and in the bytecode; because of this and due to limitations of the Java bytecode format, the contained debugging information is not sufficient to store source location information about all aspect-oriented elements that are compiled. Thus, tasks are either only partially supported (T1, T6, T7) or not at all (T2, T3, T4, T5, T8, T9, T10, T11). For example, the stack trace (T7) becomes misleading when it involves the execution of advices. A stack frame representing the execution of an advice indicates that this execution is invoked by the method represented by the previous frame. However, this method does not contain this invocation but the advice is implicitly triggered by a pointcut defined in another piece of code.

The *Aspect Oriented Debugging Architecture* (AODA) by De Borger et al. [21] is built based on the debugging interface *AJDI* which restores some source-level abstractions from the bytecode. Entities in the debugging interface model reflect many AspectJ concepts, such as join points, advices, etc. The debugging interface allows to query advices applied at a join point, the stack trace with advice execution history, and so on. Besides, the AODA contains an aspect-aware breakpoint model which allows programmers to set a breakpoint to aspect-related operations like the instantiation of an aspect. However, their model is not fine-grained enough; it lacks entities which cannot be represented in a non-AO IR like patterns, or precedence declarations. Thus, tasks T2, T3, T6 are partially supported and T5, T9 are not supported by AODA. Due to the compile-time weaving strategy fostered by AODA, it is impossible to exclude AO definitions at runtime (T10, T11).

The *AwesomeDebugger* [4] is a command-line debugger for debugging applications written in multiple domain-specific aspect languages. It uses MDDI which is a debug interface extending *AJDI* with inspection facilities that consider specifications of inter-language composition. The specification includes types of join points that a language can intercept, whether a join point is advisable, and how a language affects a join point. This debugger extends the abilities of AODA in handling multiple languages instead of providing a finer-grained debugging model. Therefore the *AwesomeDebugger* has the same characteristics with respect to our tasks as identified for AODA above.

Wicca [24] is a dynamic AOP system for C# applications that performs source weaving at runtime. For debugging purposes, the woven source code can be inspected, e.g., checking if programs are composed correctly. Wicca also allows to enable/disable aspects at runtime. Though Wicca fully supports T8, and T10, it does not support our other identified tasks because it debugs the woven code and the history of (un-)activation is not tracked. Although the presented C# source code is more easy to understand than woven bytecode, which is available in other systems, it does not contain the AO source-level abstractions anymore.

The identified tasks are not only valid for interactive debuggers, but also for other types of debuggers as long as they are designed for AD languages. Therefore, we also discuss the only related trace-based debugger proposed by Pothier and Tanter [70]. A more detailed discussion about their work is in chapter 4. Pothier and Tanter implemented an AO debugger based on an open source omniscient Java debugger called *TOD*. TOD records all events that occur during the execution of a program and the complete history can be inspected and queried offline after the execution. Programmers can choose to present all, part or none of the aspect activities carried out during runtime. It can show the execution history of join points related to particular AO elements, e.g., where a pointcut matched or did not match. However, the granularity of such elements in TOD is as coarse as in the other presented approaches for debugging woven code. Therefore, TOD only partially supports T1, T2, T6, T7, T8, and it does not support the other tasks at all.

The above discussions is summarized in Table 2.5. *Aws* is short for the AwesomeDebugger, since it is based on AODA and both approaches have the same evaluation results with respect to the identified tasks, they are grouped. In the table, tasks T5 and T9 are not supported at all by any of these debuggers; for tasks T2, T3 and T6 only partial support is provided by the related approaches. The reason for these limitations is the approach that all previous debuggers share: They debug woven code which lost some of the aspect-oriented abstractions. In contrast, our approach introduces an intermediate representation that preserves all source-level abstractions and thus allows observing and interacting with the execution of the debuggee in terms of these abstractions.

2.7.2 Development Tools for Advanced-Dispatching Languages

AO-specific information provided by tools or systems are not only provided in online debuggers. Static tools can be used as auxiliary approaches to understand program behavior or structure during debugging.

Common IDE tools for AOP languages, like the AspectJ Development Tools

Tag	Task Name	Our debugger	JDB	AODA & Aws	Wicca	TOD
T1	Setting AO breakpoints	✓	○	✓		○
T2	Locating AO constructs	✓		○		○
T3	Evaluating pointcut sub-expressions	✓		○		
T4	Flattening pointcut references	✓		✓		
T5	Evaluating pattern sub-expressions	✓				
T6	Inspecting runtime values	✓	○	○		○
T7	Inspecting AO-conforming stack traces	✓	○	✓		○
T8	Inspecting program compositions	✓		✓	✓	○
T9	Inspecting precedence dependencies	✓				
T10	(De-)activating AO definitions	✓			✓	
T11	Inspecting the history of (de-)activation	✓				

Table 2.5: Comparison between different AOP debuggers from the perspective of supporting the identified tasks.

(AJDT)[1], CaesarJ [5] Development Tools (CJDT)[2], JAsCo [77] Development Tools (JAsCoDT)[3], etc., require using the Java debugger. Thus, abstractions inspected during debugging are Java abstractions resulting from the weaving compilation. They provide additional, static features decreasing the effort in understanding and coding corresponding programs. For example, AJDT provides the *Aspect Visualiser* to find places affected by an aspect. JAsCoDT has an *Introspector* which displays the connectors found within the system.

For the ObjectTeams programming language an Eclipse-based IDE exists that enhances the standard JDT Java debugger [39]. The enhancement filters call frames that belong to infrastructural code and adapts the placement of breakpoints. The step-into debugger action is aware of "callin bindings" which correspond to advices in AOP. The ObjectTeams Development Tools (OTDT) provide a view for showing the active and inactive "Teams", their form of aspect declarations, allowing to dynamically enable and disable Teams, similar to the Attachments view of our debugger. While these enhancements hide the details of generated code from programmers, it still falls short in providing additional language-specific functionality.

Some work has been performed on enhancing the visualization of the structure of AO programs. Pfeiffer and Gurd [69] introduced a treemap-based visualization, called *Asbro*. Asbro uses colored and nested rectangles to present the hierarchy as well as the crosscutting structure. It is especially effective in navigating large-scale AO programs. Fabry et al. [28] also use a hierarchical way visualizing how aspects affect the base code. However, their work provides more specific information, such as the precedence between advices, at the granularity level of methods. Coelho and Murphy [18] implemented *ActiveAspect* that can automatically decide which subset of the crosscutting structure should be presented depending on the selected elements in the IDE. Thus, it decreases the complexity of information to be analyzed. These tools or systems aid language users to comprehend programs by simplifying the presentation of the crosscutting structure. Our debugger concentrates on dynamic information, especially for pointcut and pattern evaluation, and program composition.

The JPred Eclipse plug-in[4] provides a view showing implication relationships between predicates used for methods sharing the same signature. This is shown in terms of a Binary Decision Diagram, similar to ALIA4J's dispatch execution strategy. It indicates that a method with a more specific predicate has higher priority to be executed. Compared to this view, the graphical representation of our dispatch function decomposes each predicate into a set of atomic predicates.

[1] See `http://www.eclipse.org/ajdt/`.
[2] See `http://caesarj.org`.
[3] See `http://ssel.vub.ac.be/jasco/index.html`.
[4] See `http://sourceforge.net/projects/eclipse-plug130/`.

It shows the evaluation order of predicates instead of the relationship between them. In contrast to our online debugger, the JPred plug-in only statically shows the decision process of dispatch.

2.7.3 Tool Customizations

To provide highly tailorable user interfaces, many frameworks are created. For example, FlexiBeans [84] uses a component technology to provide a tool with run-time flexibility. Heer et al. [37] found that existing information visualizations like treemaps [14] are difficult to be reused in a different context. Therefore, they proposed the tool *prefuse* on which programmers can interactively compose the visualization of data by using predefined visualization components. Schäfer and Mezini [74] also argued that modern tools lack flexibility of customization and presented an approach implementing a flexible framework for visualizing code *structure*.

2.8 Conclusion

In this chapter, we have investigated four fault models for aspect-oriented programming (AOP) languages and categorized AOP faults related to dynamic features into seven fault categories. To detect all kinds of dynamic AOP faults, we identified eleven tasks that an ideal AOP debugger should be able to perform.

To enable these tasks, the debugging infrastructure must use an intermediate representation of the program to debug which preserves all source-level abstractions. This is necessary to let the programmer inspect and influence the execution of all aspect-oriented program elements in the source code. It must be possible to add source-location information to elements in the IR to be able to localize their source definition during a debugging session. One source construct can be mapped to multiple compiled entities and vice versa. We developed our prototype on ALIA4J which provides an intermediate representation for languages with advanced-dispatching. This IR is expressive enough to represent many-to-many relationships with source code elements, as outlined above. We transform aspect-oriented declarations into AD models and store them in an XML file after compilation. The stored information is available to the debugger by means of the Advanced-Dispatching Debug Interface (ADDI), which allows observing the program executions in terms of AD abstractions. Based on the ADDI, we implemented a user interface in terms of four new and one extended Eclipse views.

Built on the language-independent meta-model, our debugger can be applied to all AD languages. To support language-specific presentations of the model in the user interfaces, we allow programmers to customize the description of an

entity. To enable implementing customizations without changing existing infrastructure we leverage the extension point mechanism provided by Eclipse platform. We discuss three alternative approaches for choosing a customization and where to apply it in multi-language projects: (1) local customization, (2) global default customization, and (3) globally specific customization. We have performed a preliminary evaluation of comprehension, configuration, and implementation effort in the three approaches. According to the evaluation, the third approach provides relatively high precision without much implementation effort. This is the reason why we chose this approach in our implementation. To be able to restore the original source representation for an IR entity as faithfully as possible, customizations can consider the following information: the source language, the entity class, the runtime values, and the context of other IR entities in which it is used.

According to the identified AOP debugging tasks which we generalized from commonly identified AOP faults in the literature, our debugger is the first approach to fully provide the following features.

1. It visualizes all evaluation results of pointcut sub-expressions at a join point, and it represents the constraints defined in the AOP program that lead to a specific composition.

2. It performs evaluations on pointcut and pattern sub-expressions.

3. All elements that rule the execution at a join point are shown by the visual debugger and the source code defining them can be located.

4. The runtime stack is enhanced to present join points as well as all applicable advices at once.

5. It visualizes the declarations leading to a program composition at a join point.

6. It shows all advices defined in the program and allows (un-)deploying them at runtime. Besides, it can show the history of (un-)deployments.

7. While being generically applicable to aspect-oriented and advanced dispatching languages, the user interface of our debugger allows customizing the visualization to a language-specific flavor.

3

A Pointcut Language for Setting Advanced Breakpoints

3.1 Introduction

In interactive debugging, programmers use breakpoints to mark places in the source code where the program should be suspended at runtime. When the debuggee program is suspended, programmers can inspect the program state, observe the program behavior, or perform other debugging tasks by using a debugger. A detailed introduction about breakpoints is given in Section 1.1. How breakpoints are set can significantly affect the efficiency of debugging. An inexperienced programmer may set too many breakpoints; redundant ones distract her attention from those revealing the defect. Or she may set too few, which results in passing the defect.

Breakpoints should not be arbitrarily set, because each suspension has a cost. At least, programmers need to decide whether the suspension is relevant. Observing a symptom, programmers usually first roughly choose a program slice that is most likely to cause the failure, and then set breakpoints to observe the slice. For example, if a field stores a wrong value, breakpoints are set at places where the field is modified. Thus, logically, breakpoints are grouped by programmers according to what they do instead of where they are. However, the traditional breakpoints are mainly based on source lines, which do not embrace any logic of why breakpoints are placed there [22]. Programmers have to mentally sketch the logic relationships between these breakpoints.

Also, breakpoints are independent from each other. At runtime, each breakpoint has its states such as being activated, and contexts such as the value of a variable. Sometimes, a desired suspension requires multiple breakpoints and their states which sequentially form a path leading to the suspension. This may

require non-trivial manual effort, such as recording the state or context at one suspension and using it at another. Limitations of the traditional breakpoints often require programmers to perform many repeated debugging steps. Thus, debugging efficiency is decreased.

We identified five frequently encountered scenarios that the traditional breakpoints cannot handle well. The identified scenarios require breakpoints to be set at places sharing some common characteristics, such as similar syntax. The concept of *pointcut* perfectly fits in this context. These scenarios show that using pointcut-advices to construct places for setting breakpoints is a more convenient and efficient approach than traditional debugging. However, current AOP languages are not specifically designed for solving these scenarios. Thus, pointcut-advices are too verbose. Furthermore, pointcut-advices cannot be used to set a breakpoint to specific advice compositions, which is one of our identified scenarios. Also, programs that are added for debugging may accidentally stay in the project. This will introduce unnecessary maintenance effort in the future.

Therefore, we propose a declarative breakpoint language (BPL). By building on AspectJ, BPL can be used to debug Java or AspectJ programs. The breakpoint, that is the core concept of BPL, is a first-class value. A breakpoint can be defined by AspectJ-like pointcuts which use source-level abstractions. This makes the description of breakpoints more comprehensible than line breakpoints. We extend and improve existing AspectJ pointcut designators with seven novel ones. In BPL, breakpoints are named and can be used to compose higher-level breakpoints. The composition level can be infinite because we treat the primitive and the composed breakpoints in a uniform way. BPL is the first approach providing a pointcut for selecting a specific action composition at runtime.

This chapter is structured as follows. Section 3.2 presents five debugging scenarios and describes how debugging processes are performed in different approaches. Section 3.3 gives a detailed introduction to the new features introduced in BPL. Section 3.4 highlights several implementation considerations in our prototype. Section 3.5 describes a code analysis that measures the probable manual effort involved in each identified scenario. Section 3.6 describes two debugging examples by using the traditional debugging, the program solution, and our solution respectively. Section 3.7 and 3.8 describe related work and conclude this chapter respectively.

3.2 Problem Statement

Debugging is a cognitive process and how it is performed significantly depends on the programmer's observations and experience. A programmer tends to give the same treatment when she observes the same symptom, such as a certain

exception being thrown. In this section, we select several debugging scenarios that are frequently encountered. For each scenario, we elaborate the way of using the traditional debugger. We tag debugging steps in the description like "a.1", in which the letter represents a debugging process and the digit represents the step order of that process.

Each scenario requires multiple breakpoints or suspensions at different locations, which share some common properties, such as similar syntax, relation to the same variable, etc. In AspectJ, a pointcut is used to select places with common properties. This has inspired us to use AspectJ programs during debugging, where pointcuts select the join points at which we want to suspend the execution and where we set a breakpoint in the otherwise empty advice body. In this section, we also demonstrate this approach for the identified scenarios. Actually, this approach is a variant of program instrumentation.

The program solution serves two purposes. First, the program can describe the scenario in a more succinct and clear way than instructions for manual debugging given in natural language. Take pointcut **call**(**void** Shape.set∗(..)) for example, it can be seen as two debugging tasks in this context: finding all places calling methods which satisfy the pattern "void Shape.set∗(..)", and then setting line breakpoints there. Second, the programs will be compared with our solution, which is an AspectJ-like language.

We have two basic criteria for the program solution. First, the program should be in a separate module. AspectJ modularizes scattering concerns and we want to keep this principle in the program solution. Second, the program should be simple. Effort spent on writing the program should be comparable to that spent on the traditional debugger. In most cases, writing a lengthy or a sophisticated program for setting a breakpoint is not desirable.

3.2.1 Scenario 1: Selecting Multiple Locations

Sometimes, it is difficult to decide which specific location is executed at runtime. For example, to debug unexpected behavior in a system, which the programmer is not familiar with, she may deduce roughly which function is executed by matching names of the function with the observed runtime behavior. A function can be implemented as a set of overloading constructors or methods. However, to know which specific one is executed at runtime, she may need to set a breakpoint to each implementation.

The difficulties of using the traditional debugging in this scenario mainly come from finding locations for setting breakpoints and managing breakpoints as logic units.

Finding locations Suppose the programmer observes that a field stores an unexpected value, she needs to monitor the runtime states of this fields. The first option is setting a watchpoint to this field (**a.1**). When the watchpoint is hit, the programmer needs to perform one "step over" to inspect the field value after the modification (**a.2**).

The second option requires the programmer to manually find out the last assignment (**b.1**) to this field before the unexpected value is observed. This assignment can be in any constructor or method modifying this field. She needs to set a line breakpoint to each found place (**b.2**) and specify a condition to check whether the expression on the right-hand side of the assignment equals the unexpected value (**b.3**).

Managing breakpoints Using the *Breakpoint* view provided in modern IDEs, such as Eclipse, the programmer can organize the breakpoints she set. The view can group the breakpoints according to their types, such as line breakpoint or watchpoint, or their locations, such as files or projects. Breakpoints can be (de)activated and deleted at the granularity of groups. However, there is no approach provided to group breakpoints as logic units. Thus, a debugging task applied to a logic operation will possibly required repeating steps.

Listing 3.1 shows how an AspectJ program monitors unexpected assignments to the field Clazz.var. AspectJ can access the value assigned to a field by using **args**(). The pointcut describes the desired places for suspensions and a breakpoint is set in the body of the **before** advice. When the program is suspended in the advice, the programmer can locate the defect by using the stack trace. Usually, the second top frame in the stack trace points to the defect, because the top frame represents the execution of the advice.

```
1 public aspect Scenario1Aspect {
2   before(int val) : set(int Clazz.var) && args(val) &&
3       if(val==/*Unexpected value*/) {
4     // set a breakpoint on this line
5   }
6 }
```

Listing 3.1: An AspectJ program monitoring assignments to a field

3.2.2 Scenario 2: Monitoring Updates on a Field

Listing 3.2 shows a program slice updating a field, which is a HashMap. On lines 4-8, we use ellipsis to indicate that the separated statements may reside in different methods and their execution order is not the same as the lexical order.

```
1 class Scenario2 {
2   private HashMap map1;
3
4   map1.put(key1, value1);
5   ...
6   map1.get(key2);   //returns a null value
7   ...
8   map1.put(key2, value2);
9 }
```

Listing 3.2: Multiple places updating a field

When a value retrieved from the HashMap is wrong, as line 6 shows, the potential defects are places updating this HashMap, such as lines 4 and 8.

To debug this with a traditional debugger, the programmer needs to find all the updating locations (**c.1**), set breakpoints there (**c.2**), and evaluate the values of the expressions for updates at runtime (**c.3**). A watchpoint for the field is not helpful in this scenario, because it can only suspend the program when the field is accessed or modified instead of being updated. Setting a breakpoint to the called method HashMap.put() may result in redundant suspensions, because there may be other HashMaps in the program.

Listing 3.3 gives an AspectJ solution for this scenario. It is a *privileged* aspect which can access protected members of other classes. On line 6, the advice checks whether the callee object t is same as the value stored in the expected field, such as the private field map1 in Listing 3.2.

```
1 public privileged aspect Scenario2Aspect {
2   before(Scenario2 caller, HashMap t, String s) :
3       call(public Object HashMap.put(..)) && this(caller) &&
4       target(t) && args(s, *) &&
5       if(s.equals(/*value of key2*/)) {
6     if(caller.map1 == t) {
7       // set a breakpoint on this line
8     }
9   }}
```

Listing 3.3: An AspectJ program monitoring updates on the object referenced by a specific field

3.2.3 Scenario 3: Finding Null Pointer Dereferences

The dot operator dereferences an object pointer to access a member from that object. A line of code may contain multiple dereference operations as in the

following listing:

```
1 total.getObjects().addAll(current.getObjects());
```

If a *NullPointerException* occurs on this line, the error message only tells the line number where the exception occurs instead of the specific operation. For debugging this scenario, the programmer needs to place a breakpoint at the line where the exception occurs (**d.1**). When the program is suspended, she needs to repeatedly perform "step into" and then "step return" to check each dereference operation until the exception occurs (**d.2**). Meanwhile, she has to manually note which dereference operation the debugger reaches (**d.3**).

Another option is to change the layout of the code so that there is a dereference operation per line, like the following listing shows (**e.1**). After rerunning the program (**e.2**), the error message can accurately tell which line throws the exception. Since this option requires rewriting the source code, it is also not generally applicable.

```
1 total.getObjects()
2   .addAll(
3     current.getObjects());
```

Listing 3.4 presents an AspectJ program corresponding to this scenario. The pointcut is only satisfied if the receiver of a dereference operation is **null** (see line 5). The advice body further restricts the line number on line 7. If a breakpoint is set at line 8, it can suspend the execution before the dereference operation, which ends up with a *NullPointerException*, is about to occur.

```
1  public aspect Scenarios3Aspect {
2    before(Object receiver) :
3        (call(* *.*(..)) || get(* *.*)) && target(receiver) &&
4        withincode(/*a method pattern*/)
5        && if(receiver == null) {
6      int line = thisJoinPoint.getSourceLocation().getLine();
7      if(line == /*expectedLine*/) {
8        // set a breakpoint on this line
9      }
10   }
11 }
```

Listing 3.4: An AspectJ program checking *null* receivers on a source line

3.2.4 Scenario 4 : Recording Execution History

Listing 3.5 shows program slices related to operations on two stream objects. An exception would be thrown when line 6 is executed, because it tries to read data from a closed Stream.

```
1 InputStream s1 = new FileInputStream(...);
2 InputStream s2 = new FileInputStream(...);
3 s1.close();
4 s2.read();
5 s2.close();
6 s1.read();  // An exception is thrown.
```

Listing 3.5: A program performing operations on stream objects

An execution path can lead to unexpected behavior, such as first close then read. Programmers need to track the cause backwards from the point where the symptom is observed. However, most traditional debuggers do not provide backtracking. Using breakpoints, the programmer is likely to suspend the program either before or after the cause. If the cause is passed, the programmer needs to restart a new debugging session. Moreover, debugging Listing 3.5 requires that all events of the path refer to the same object. The programmer has to manually note corresponding information with the traditional debugger.

```
1 tracematch(Stream s) {
2   sym close before : call(* Stream.close(..)) && target(s);
3   sym read before : call(* Stream.read(..)) && target(s);
4   close read {
5     // set a breakpoint on this line
6   }
7 }
```

Listing 3.6: A tracematch specifying an undesired execution path

Tracematch [3] is an AspectJ extension designed for observing execution traces. Therefore, we choose Tracematch as the alternative debugging solution for this scenario. Listing 3.6 shows a Tracematch program. Lines 2 and 3 define two events named close and read. Both of them bind the **target** value to the parameter of the tracematch (line 1). Events with different **target**s are not recorded by the same **tracematch** instance. Line 4 declares the expected, but undesired execution path with the two names. When the path is matched on the same Stream, the instruction represented by line 5 is executed.

3.2.5 Scenario 5: Exploring a Program Composition

The execution of an advice can alter the flow of its base program to any extent. Many works [23, 36, 45, 57, 63] have identified the problem of aspect (or advice) interference. An incorrect composition, which can be either between advices or between advices and the base program, at a join point causes unexpected runtime behavior. Therefore, the programmer needs to inspect the execution of the composition at such a join point. What complicates this task is that pointcuts can include dynamic tests. Thus, when advice share a join point shadow, these advice are not necessarily executed together.

To debug this scenario, the programmer first needs to find join point shadows (JPS) affected by all advices of the expected composition (**f.1**) and then set breakpoints to these shared JPSs (**f.2**). Because in AspectJ, whether an advice is applied can only be seen when it is actually executed, the programmer needs to execute the program once (**f.3**), manually perform bookkeeping of the program composition at each join point (**f.4**) and record the hit count of the line which contains the desired join point (**f.5**). In the second debugging session, setting line breakpoints with the recorded hit counts (**f.6**) leads to suspensions at the expected join point before any advice is executed.

For this scenario, AspectJ cannot provide a clean way for putting debugging code in a separate module. There is no pointcut which can uniquely identify the execution of an advice, because advices are unnamed in AspectJ. Furthermore, an advice using the pointcut **adviceexecution()** cannot easily obtain information about the join point triggering the execution of the advice. Without this information, it is impossible to know whether different advice executions are composed at the same join point. Though there are works, such as Oarta [58] and dependent advice [12], supporting named advices, none of them can use advice names to specify an expected runtime composition.

3.2.6 Summary

In this section, we have described five debugging scenarios that require non-trivial manual tasks such as setting breakpoints, repeating steps, and recording past states. For these scenarios, the program solutions that describe the suspension conditions in a declarative way show their strength and potential.

However, some debugging programs are verbose. In Listing 3.3, comparing the field value and the current target object is always required in debugging scenario 2. In Listing 3.4, most of the parts are generic except the location information. These programs can be more reusable if the configurable parts are parameterized. Besides, there is no solution that treats these scenarios in a uniform way. For example, Tracematch can specify a sequential execution of operations (scenario

4). If one of the operation is expected to be advised by some advice (scenario 5), Tracematch cannot specify an *advising* pattern and compose the new pattern with the sequential one.

Last but not least, we discourage to add code, which is not part of the main functionality, to the source program. The added code may introduce unnecessary maintenance effort if the programmer forgets to remove it after fixing a bug. Even though source control management systems, such as subversion, can be used to tell the differences between two versions, programmers need to distinguish the added debugging programs and the fixed parts.

3.3 Breakpoint Language

Based on the observations described in Section 3.2, we have designed and developed a breakpoint language (BPL) for setting advanced breakpoints. BPL reuses many features of AspectJ and Tracematch. Additionally, it has its own unique functionalities.

The debuggee programs of BPL are Java programs or AspectJ programs. During the debugging, breakpoints specified by the BPL suspend the debuggee program at join points where they are satisfied.

Listing 3.7 shows the grammar rule of a breakpoint declaration. A breakpoint declaration has a name, a parameter list, and a pointcut expression. The rule for PointcutExpr extends the AspectJ pointcut with seven designators. We describe these designators and their usages in the following subsections. A complete grammar definition of BPL is given in Appendix A.

```
1 BreakpointDeclaration :
2   Name '(' FormalParameterList? ')' ':' PointcutExpr ';' ;
```

Listing 3.7: The grammar rule of a breakpoint declaration

3.3.1 The Pointcut call()on()

The pointcut **call**()**on**() is derived from the pointcut **call**(). It matches join points where a method is called on an object referenced by a specific field. The **call**() and **on**() parts take the method and the field specifications respectively. This pointcut can be used at any place where **call**() is applicable. It should be noted that the **on**() part matches based on the referential identity of the values, i.e., it also matches alias of the specified field.

Listing 3.8 shows an example of using **call**()**on**(). Compared to Listing 3.3 in **Scenario 2**, it constrains the callee object of the method.

```
1 bp(String s) :
2   call(public Object HashMap.put(..))on(Scenario2.map1) &&
3   args(s, *) && if(s.equals("key2"));
```

Listing 3.8: A breakpoint declaration using pointcut **call**()**on**()

3.3.2 The Pointcuts location() and checkNPE()

As the following listing shows, the pointcut **location**() takes three parameters which represent the file path, the file name, and the line number in the file respectively. The third parameter takes a list of line numbers or line ranges, e.g., [97, 100..102]. This pointcut can be used jointly with other pointcuts to restrict locations of JPSs, for example **call**(...) && **location**(...).

```
1 bp() : location(``Spacewar", ``SpaceObject.java", [97]);
```

The pointcut **checkNPE**() matches dereference operations where the receiver is *null*. A breakpoint using **checkNPE**() suspends the program just before the satisfied dereference operation is performed, and thus the suspension happens before a *NullPointerException* is thrown. The first breakpoint declaration shown in Listing 3.9 checks whether a line contains a null pointer dereference. The second breakpoint declaration does the same for the specified method body.

Compared to Listing 3.4 in **Scenario 3**, **checkNPE**() omits the fixed parts specifying the cause of *NullPointerException*; only the source location is left to be configured.

```
1 bp1() : checkNPE() &&
2         location("Spacewar", "SpaceObject.java", [97]);
3 bp2() : checkNPE() && withincode(/*a method pattern*/);
```

Listing 3.9: Breakpoint declarations using pointcuts **location**() and **checkNPE**()

3.3.3 The Pointcuts path() and bind()

The pointcut **path**() matches a specific execution path existing in the history. It takes a path expression, which consists of breakpoint references, as the parameter. We use a blank space to represent the sequential order and rectangular brackets to represent the exact expected hit count. For example, **path**(a[2] c) expects that breakpoints a and c are hit in the sequence "aac". Expression c is shortened from c[1]. Besides, the "+" sign, which means 1 or more, and "*", which means 0 or more, can be appended to a breakpoint reference. The **path**() expression is

satisfied when all referenced breakpoints are satisfied in the sequence specified as the path expression.

The pointcut **bind**() is used to bind context values exposed by lower-level breakpoints to the higher-level breakpoint. Listing 3.10 shows our solution for **Scenario 4**, which is about recording execution history.

Lines 1 and 2 declare two breakpoints for read and close operations respectively. Lines 3–5 declare a composite breakpoint. Line 4 describes an expected execution path which requires that breakpoints close and read are hit sequentially. Line 5 binds values from lower-level breakpoints to the parameter declared on line 3. A binding relies on the name of a parameter in the composite breakpoint and the position of a parameter in the lower-level breakpoint. For example, read(s) binds the first parameter of the breakpoint read to the parameter named s of the composite breakpoint closeRead. Wildcards can be used to skip parameters that are not relevant to the breakpoint declarations. For example, read(∗, s) and read(.., s) bind the second and the last parameter respectively. In Listing 3.10, both bindings bind values to the same parameter s and this implies that the bound values must refer to the same object.

```
1 read(Stream t) : call(public * Stream.read()) && target(t);
2 close(Stream t) : call(public * Stream.close()) && target(t);
3 closeRead(Stream s) :
4   path(close read) &&
5   bind(read(s), close(s));
```

Listing 3.10: Declaration of a composite breakpoint

It is also possible to bind values to different parameters, as Listing 3.11 shows. The equality of bound values is specified explicitly in the **if**() expression on line 4. Both breakpoints closeRead and closeRead_if suspend the program at the same times. The former is more succinct and the latter is more flexible with restricting the bound values.

```
1 closeRead_if(Stream rStream, Stream cStream) :
2   path(close read) &&
3   bind(read(rStream), close(cStream)) &&
4   if(cStream == rStream);
```

Listing 3.11: A composite breakpoint using **if**()

Our solution for the path expression is greatly inspired by Tracematch, but there are two fundamental distinctions. In the view of the structure, a primitive event declared in one tracematch cannot be referred to by other tracematches. In BPL, primitive breakpoints are more reusable, because they can be referred

in any number of composite breakpoints. In the view of the join point model, Tracematch is interested in event kinds, such as *before* and *around*. BPL runs with an interactive debugger that provides only forward execution. Therefore, it only suspends the program *before* executions of the satisfied join points.

3.3.4 The Pointcuts adviceexecution() and composition()

In AspectJ, **adviceexecution**() does not take any parameter and it cannot select the executions of a specific advice. BPL provides a backwards-compatible extension of **adviceexecution**(), which can take the fully qualified name of an advice as the parameter. As an example, pointcut **adviceexecution**(GameInfo.guiInitiation) selects the execution of the advice declared in the aspect "GameInfo" and named with "guiInitiation". Section 3.4.3 describes how advices are named.

We use the term "action" to refer to an advice, a method or constructor call, a field access, etc. The **composition**() pointcut designator selects join points with an action composition where actions have the specified relationship. To use this pointcut, the programmer first needs to declare breakpoints that suspend the program at the executions of the desired actions. Then, she can use the names of the declared breakpoints to specify a *composition pattern*. Last, the pattern is used as the parameter of **composition**().

We provide two types of composition pattern. Suppose beforeExe and afterExe are two breakpoints that both use an expression **adviceexecution**(). A breakpoint using **composition**() suspends the program at a join point where the composition satisfies the specified pattern.

Existence - the actions referenced in the specified pattern should exist in the composition. We use commas to list breakpoints corresponding to desired actions, e.g., **composition**(afterExe, beforeExe).

Exclusion - the actions referenced in the pattern should not occur in the composition. We put an exclamation mark before a breakpoint reference for this relationship, e.g., **composition**(!afterExe).

3.4 Implementation Considerations

In Chapter 2, we introduced a interactive debugger for advanced-dispatching (AD) languages, which is built on top of the execution environment NOIRIn. BPL is implemented to work together with the AD debugger. When a breakpoint is hit at a join point, the programmer can use the AD debugger to observe the suspended program. At runtime, the breakpoint declarations are sent to NOIRIn.

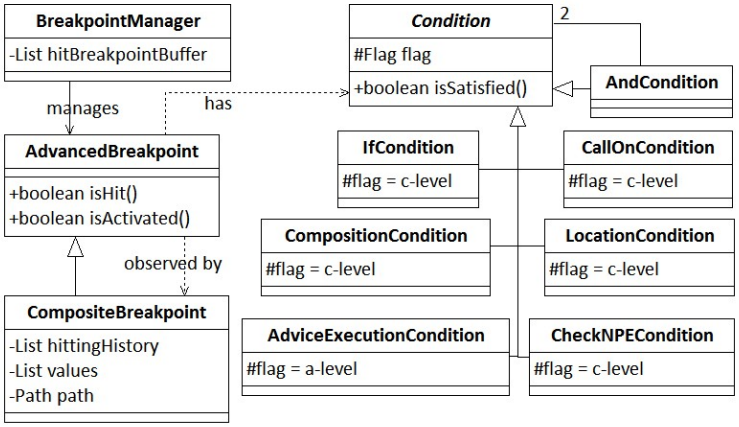


Figure 3.1: A class diagram of classes related to the breakpoint in BPL.

They are evaluated in the context at a join point along with the execution of rest of the program.

3.4.1 Evaluation of Breakpoints

Figure 3.1 shows a diagram of the classes that are used in our implementation to represent breakpoints in BPL in the execution environment. *AdvancedBreakpoint* represents the breakpoint and it is managed by a *BreakpointManager*. Each breakpoint has a *Condition* specifying in what condition the breakpoint can be hit. The figure includes only the conditions related to the pointcut designators introduced in section 3.3.

When the program reaches a join point at runtime, NOIRIn first analyzes the call context, then computes the action composition performed at this join point, and finally executes actions in the composition. Breakpoints are evaluated when an action is about to be executed after all context information, including the call context, the composition, and the executing action, is prepared.

For all primitive conditions except *AdviceExecutionCondition*, it is enough to be evaluated once at a join point. To distinguish this different evaluation frequency, we put a flag to *Condition* and its subtypes. The flag has two values, which are composition-level (*c-level*) and action-level (*a-level*). A breakpoint with a c-level condition is evaluated once at a join point and one with an a-level condition is evaluated at every action in the composition. A binary condition such as *AndCondition* is c-level if and only if its two operators are c-level.

Multiple breakpoints may suspend the program at the same join point. We use a buffer *hitBreakpointBuffer* to store the hit breakpoints at a join point. A hit breakpoint sends a message with required debugging information to the buffer.

When the evaluations of all breakpoints are finished, the *BreakpointManager* checks whether there is any message in the buffer. If the buffer is not empty, the manager emits a suspending request and releases all messages stored in the buffer. The buffer is cleared when the next evaluation process starts.

3.4.2 Evaluation of Composite Breakpoints

Using a **path**() expression, a composite breakpoint can be composed of primitive breakpoints or other composite breakpoints. It may use **bind**() to access values from lower-level breakpoints and further restrict its suspending condition by using **if**().

In Figure 3.1, there are two lists in class CompositeBreakpoint. The list *hittingHistory* records the hit history of the lower-level breakpoints. The list *values* records values bound to the parameters. A *CompositeBreakpoint* is an observer of all its lower-level breakpoints. Whenever one of its lower-level breakpoints hits, the evaluation of the composite breakpoint starts. The evaluation first updates lists *hittingHistory* and *values*. Then, it explores the *hittingHistory* list and tries to find an expected path. If an expected path exists, the composite breakpoint starts to evaluate the *if()* condition. If the *if()* condition is true, the composite breakpoint is hit and it produces a message containing all the debugging information, such as locations and bound values, of its lower-level breakpoints.

For illustration, take Listing 3.5 as the debuggee program, Listing 3.11 as the breakpoint declarations. Figure 3.2 shows a complete evaluation process of the breakpoint closeRead_if. Column *Code* lists code where the primitive breakpoints hit. Columns *Hit History* and *Values* describe the runtime states of the lists *hittingHistory* and *values* respectively. Column *Evaluation* has two sub-columns which represent the two-stage evaluation respectively. Sub-column *path* represents matches on the execution path and sub-column *If expr.* represents the test of the **if**() expression. "T" and "F" stand for true and false. When a path is found, a "T" is put in the sub-column *path*. A list representing the indexes of the hit history is put after "T", as in like T{0,1}.

The breakpoints are hit sequentially from top to bottom in the table and the evaluation is performed accordingly. When the program reaches s2.read(), a path is found with indexes {0,1}. Then, the composite breakpoint uses the indexes of the path to retrieve values, which are required by the **if**() condition, from the *values* list. However, the condition "cStream==rStream" does not hold. When the program reaches s1.read(), the composite breakpoint finds a path with indexes {2,3} but the **if**() condition again does not hold. Then, another path with indexes {0,3} is found and satisfies the **if**() condition. The composite breakpoint closeRead_if is hit and it produces a suspension message.

Code	Hit History		Values		Evaluation	
	Index	Bp. ref.	Index	Map	path	If expr.
s1.close( )	0	close	0	“cStream” →s1	F	-
s2.read( )	1	read	1	“rStream” →s2	T{0,1}	F
s2.close( )	2	close	2	“cStream” →s2	F	-
s1.read( )	3	read	3	“rStream” →s1	T{2,3} T{0,3}	F T

Figure 3.2: A table showing how a composite breakpoint stores the hit history and the bound values

3.4.3 Named Advices

Bodden et al. [12], as well as Marot and Wuyts [58] proposed named advices for the purpose of uniquely identifying an advice. They extended the AspectJ syntax to achieve this goal. We name advices for referring to them in pointcut **adviceexecution**(). Besides, we do not intend to extend the syntax of the debuggee program, because the effort integrating it with the rest of our debugging infrastructure [87] is not trivial. Annotations are not supported in the *abc* compiler, which is part of our tool chain. Therefore, we choose to use comments as the approach for naming advices.

```
1 aspect Azpect() {
2   /**
3    * @advicename=firstBefore
4    */
5   before() : call(...) {}
6 }
```

Listing 3.12: A named advice

```
1 <map>
2   <entry>
3     <virtual>firstBefore</virtual>
4     <compiled>before$1</compiled>
5   </entry>
6 </map>
```

Listing 3.13: A naming map

Listing 3.12 shows a **before** advice with a comment naming the advice as firstBefore. The programmer can refer to this advice in the breakpoint declaration. For example, **adviceexecution**(Azpect.firstBefore). During compilation, methods

with unique identifiers as method names are created. For simplicity, we do not use the specified name as method name, but let the compiler decide the name as usual. Instead, we keep a mapping between the advices' *virtual names* as specified in the comment and their *compiled name*, as Listing 3.13 shows. During compilation, a validator checks whether there are ambiguous virtual names and prints an error message, if so.

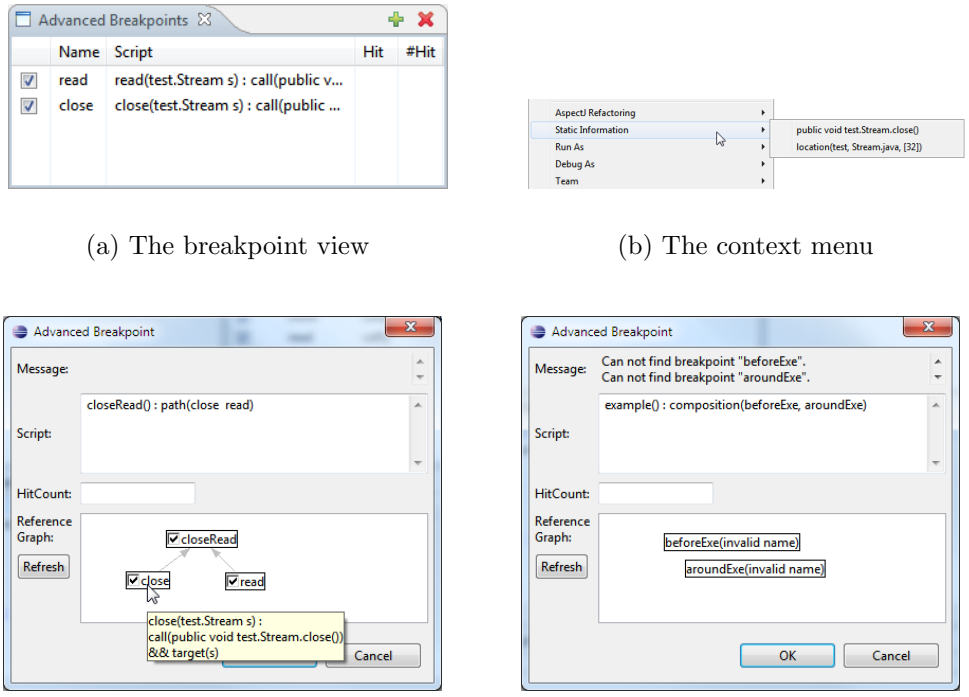
When a breakpoint declaration using **adviceexecution**() is sent to NOIRIn, which reads the virtual advice name and replaces it with the compiled name by using the naming map. When this breakpoint is hit, a hitting message is produced. The creation of the message replaces the compiled name with the virtual name.

3.4.4 User Interface

We implemented a dedicated user interface to manage and set breakpoints. The snapshots are given in Figure 3.3. The breakpoint view (Figure 3.3(a)) lists all the breakpoints and it has five columns, which are for (de)activation, presenting the breakpoint names, giving the complete declarations, showing whether a breakpoint is hit, and counting the hits respectively.

The panel shown in Figure 3.3(c) is for creating or editing a breakpoint. From top to bottom, it consists of four parts. The *message* part is for showing error or warning messages. The *script* part is for writing the breakpoint declaration. The *hit count* part is for specifying the expected hit count. The *graph* part is for presenting the reference relationships to other breakpoints. The panel in Figure 3.3(c) constructs a breakpoint which refers to breakpoints *close* and *read* in its expected path. The corresponding graph shows their hierarchical relationship. The declaration detail is shown when hovering the mouse over a label in the graph. Each label has a check box where programmers can (de)activate the corresponding breakpoint. If the declaration refers to breakpoint names which do not exist, as shown in Figure 3.3(d), an error message is shown on the *message* part and names in the graph are labeled with "invalid name".

Moreover, we provide functionalities to ease the burden of typing static information, such as signatures and line locations. By using the context menu (Figure 3.3(b)) in the editor, text can be generated according to where the cursor is. For example, if the programmer selects a method name, the signature of the method can be generated. The generated text is appended to the *script* part of the breakpoint panel. Therefore, programmers do not have to manually type such information and can further customize the generated texts, such as replacing part of the text with wildcards.



(a) The breakpoint view

(b) The context menu

(c) The breakpoint panel

(d) The panel with invalid names

Figure 3.3: Snapshots of the user interface

3.4.5 Runtime Interactivity

We allow programmers to add, delete, and update breakpoints at editing time and during the execution of the debuggee program. The addition, deletion, or update of a breakpoint does not only changes itself but may also propagate the effect to other breakpoints. A breakpoint is invalid if its declaration contains invalid reference names. When a composite breakpoint becomes invalid, it does not make sense to keep its recorded history. Therefore, the runtime changes may alter the behavior of breakpoints. Other operations, such as additions and deactivations, do not affect the validity of other breakpoints. In the following list, we discuss these effects in detail.

- If a breakpoint is deleted, all breakpoints directly or indirectly referring to it become invalid. An invalid breakpoint discards the information it has recorded. This effect propagates until no more valid breakpoint can become invalid.

- The addition of a breakpoint triggers a re-compilation of all invalid break-

points. If the added breakpoint matches the missing part in previously invalid breakpoints, the invalid breakpoints become valid. This effect propagates until no more invalid breakpoint can become valid.

- If the script of a breakpoint is updated, this can be deemed as a deletion followed by an addition.

3.4.6 Performance

To get an indication of the runtime overhead imposed by our proposed debugging approach, we performed preliminary micro benchmarks. For this purpose, we use an application that sorts 600 elements by performing *bubble sort*. To create *the worst case* of using our approach, we create a *BubbleSort* class with a method that sorts an array field. Therefore, the evaluated program intensively accesses the array field.

We run the program in three different environments.

- First, we run it on a plain Java Virtual Machine and it takes around 500ms.
- Second, we added a breakpoint, which select all field access, in BPL and executed the program on NOIRIn+BPL. To avoid measuring the time spent during the suspensions, we change the infrastructural code to let breakpoints only print a short message when they are hit. Though breakpoints do not cause any suspension throughout the whole execution, the evaluations whether each breakpoint is hit are actually performed. The second run has a slow-down of 20 times.
- Third, we changed the code layout to put each join point on a separate line, set conditional breakpoints to lines that are selected by the BPL breakpoint in the second run, and debug the program in Eclipse. The conditions set on the conditional breakpoints intentionally fail to avoid any suspension. The third run has a slow-down of 25 times.

The slow-down of NOIRIn+BPL shown in the second run is already acceptable for very short running debug sessions. Since we have focused on the semantics rather than on an efficient implementation, currently many breakpoint evaluations are redundant. In future work, we will also consider optimizations to avoid these redundant evaluations.

The time used in the third run shows that BPL has a better performance than a common debugging facility for conditional breakpoints. This is because of the different mechanisms of evaluating breakpoints. As introduced in Section 2.4, the architecture of the Java debugger consists of a debugger side and a debuggee side. Conditional breakpoints in existing techniques are evaluated on

the debugger side. To retrieve involved values in a condition, it may require several iterations of communications with the debuggee side. In our solution, the evaluations are performed on the debuggee side. Therefore, the time spend on the communication is saved.

We have not yet analyzed the memory overhead which may especially be imposed by the **path**() pointcuts. Based on preliminary experiments, nevertheless, we do not expect this memory overhead to be limiting in typical, short-running debugging sessions.

3.5 Code Analysis

To measure the how likely programmers need unnecessary manual effort of debugging each scenario in the traditional way, we defined several metrics and measured them in 9 Java projects and 5 AspectJ projects. This section discusses how the metrics were defined, how data were collected and analysed.

Except scenario 4 which requires sufficient domain-specific knowledge to detect the buggy code patterns. All other scenarios described in section 3.2 are performed on fixed code patterns. To explore how likely these fixed code patterns occur and how much debugging effort needs to be paid for each code pattern, we define the following seven metrics.

3.5.1 Metrics

Scenarios 1 contains an open list of concrete cases, such as setting breakpoints to all methods that have the same signature or to all constructors of classes that inherit some interface. We considered only three commonly encountered cases and defined three metrics accordingly. The **Modified Fields (MF)** metric measures to what extent a field is modified in multiple constructors and methods in a class. If a field has a wrong value, programmers need to suspend the program in those members and evaluate the expressions assigned to the field. This metric tells us how many breakpoints a programmer is likely to set to find the place where a field is assigned with the wrong value. The **Overloading Constructors (OC)** and the **Overloading Methods (OM)** metrics measure to what extent a constructor or a method is overloaded. The two metrics shows us how many breakpoints a programmer is likely to set to observe how an object is created or how a *function* is performed. We call a set of overloading methods with the same fully qualified name as a *function*.

For scenario 2, we considered only fields with a collection type, such as *map* and *set*. This is mainly because (1) the predefined collection types are frequently used; (2) there is an open list of customized types and most of them are not as

general as the collection types. The **Updated Collection fields (UC)** metric measures how likely a collection field is updated at multiple places in a class. It can tell us how many breakpoints are likely to be set to find the defect which updates the collection field with the wrong value.

Regarding to scenario 3, the **Dereference operations per Line (DL)** metric measures the number of dereference operations on a line. This metric indicates how many suspensions or code movements are needed for finding the problematic dereference operation on a source line.

For the composition problem mentioned in scenario 5, the **Advice Compositions (AC)** metric measures to what extent a join point shadow (JPS) is advised by multiple advices. It reveals the potential that a condition is needed to suspend the program at an expected composition consisting of a combination of advices.

3.5.2 Data Collection

In this and the following subsections, we describe how data is collected and analysed for measuring the defined metrics. All the raw data, the analysing files, and the used tool can be downloaded from our project homepage[1].

To measure the metrics for plain OO programs, we used a simple tool that scans binary class files using the ASM bytecode toolkit[2]. The tool processed all class files in a provided Java archive (jar file) and inspected all method, constructor and static initializer declarations as well as all instructions their bodies contain. Of all instructions, only the method/constructor invocation and field accesses were considered. The raw data is a comma separated values (CSV) file; for every encountered declaration, or invocation or access instruction, one row is output whereby the columns have the following meaning:

className. The fully qualified name of the containing class.

methodName. The name of the declared method or of the method containing the current instruction. Constructors and static initializers are internally represented as methods with the name "<init> " and "<clinit>", respectively.

methodDescriptor. The descriptor of the declared method or of the method containing the current instruction.

kind specifies whether the current row represents a method declaration ("d"), a method call ("c"), a field get ("g") or a field set ("s").

[1] See `http://alia4j.org/alia4j-debugging/`.
[2] See `http://asm.ow2.org/`.

accessedMember. The fully qualified name of the invoked method or accessed field, including its declaring class, the name, and the descriptor.

receiver. A specification of the origin of the receiver for the current method call or field access. This is identified with a simple data-flow analysis.

modifiedCollectionField. If the method invoked by the current instruction is one that modifies a collection object, e.g., adding an element to a list, and the receiver of the method invocation has been read from a field, the fully qualified name of that field is stored in this column.

To measure metrics for AO programs, we compiled our selected AspectJ projects in Eclipse. The *AJDT Event Trace* view can print information about the cross-cutting model for each AspectJ project. We implemented a simple interpreter to analyse the printed information. The interpreter also generated a CSV file for each cross-cutting model. Each row in a file is a relationship triple which consists of a subject, a relationship type, and an object. For example, the subject is an advice, the type is "advises", and the object is a JPS.

3.5.3 Data Analysis

We imported all CSV files to a database and ran a list of queries on the data tables. We explain how we performed the analysis by the example of the OC metric. The corresponding SQL query is shown in Listing 3.14. The query for the OC metric consists of three nested SELECT queries. The innermost query (lines 3–4) selects lines that contain a method declaration where the method is actually a constructor; [*TableName*] is a placeholder representing a concrete table name holding the data for one of the investigated projects. The second query groups these constructor declarations by the declaring class (line 5) and calculates the number of rows in each group (line 2), i.e., the number of overloading constructors named as c for each class. Finally, the outermost SELECT query groups the data by the number of overloading constructors c (line 6) and counts the occurrences of each value (line 1), giving us a frequency distribution.

We further analysed the reason for extreme values; e.g., in the data for the DL metric, we found some lines with over 50 dereference operations. We found that those lines are compiled from source code composing a *String*. The compilation transforms additions between *String* fragments to a list of consecutive *append* method calls on *StringBuffer*. It is impossible that an executed *append* returns a null value. Therefore, we added a filter in the query to exclude this case: NOT (accessedMember LIKE '∗StringBuffer∗').

```
SELECT c, count(c) FROM (
  SELECT className, count(className) AS c FROM (
    SELECT className, methodName FROM [TableName]
    WHERE kind='d' AND methodName = '<init>' )
  GROUP BY className )
GROUP BY c;
```

Listing 3.14: A query for the frequency distribution of the constructor count

3.5.4 Case Studies

To select Java projects, we considered the size, the code quality, and the domain of projects. Table 3.1 lists nine Java projects from *JFlex* with the fewest classes to *Antlr* with the most. All of them are well-known and well maintained. Among them, *JUnit* is a framework, *JHotdraw* has a significant part handling graphics, *Jigsaw* is a web server platform, and *Antlr* is compiler-related. Table 3.1 gives some general information about the listed projects.

For AspectJ projects, the size of the projects is our only criterion because there are only a handful of options. Table 3.2 lists five AspectJ projects from *Tetris* with the fewest source lines to *AJHSQLDB* with the most. We also collected the number of advices advising the same JPS for supporting metric AC.

In Figures 3.4–3.9, an average value is written on top of the bars for each group and it is rounded to the nearest integer.

Modified Fields (MF), Overloading Constructors (OC), and Overloading Methods (OM)

Figure 3.4 shows the statistics collected for the MF metric. The x-axis represents the number of different members modifying the same field. It shows that on average 32% of fields are modified at least in two members. Among them, 8% involve in more than three members. In *Ant*, a field is modified at 30 different members of the class that declares it. This field is a boolean variable and it used as a state flag. The way of using such a field is not rare.

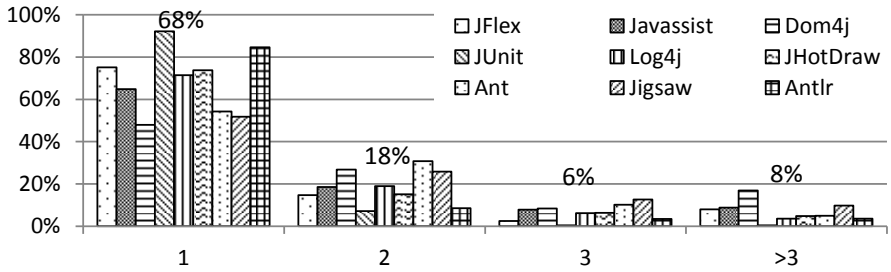


Figure 3.4: Fractions of fields modified by different numbers of constructors and methods

	JFlex	Javassist	Dom4j	JUnit	Log4j	JHotDraw	Ant	Jigsaw	Antlr
version	1.4.3	2.1	1.6.1	4.10	1.2.6	6.0	1.7.1	2.2.6	3.4
# of classes	87	113	185	226	255	395	750	933	980
# of modified fields	417	307	455	281	743	661	2,575	3,248	6,081
# of updated collection fields	25	10	43	16	41	51	225	122	237
# of functions	508	668	2,067	922	1,327	2,856	5,476	5,794	6,532
# of lines with dereferencings	1,766	400	2,768	994	2,245	4,163	9,805	15,667	27,884

Table 3.1: General information about analysed Java projects

	Tetris	Spacewar	HealthWatcher	AJHotDraw	AJHSQLDB
# of source lines	1,030	3,052	6,949	22,104	75,556
# of advices	20	11	17	37	97
# of adviced JPS	24	35	70	96	4,494

Table 3.2: General information about analysed AspectJ projects

Figure 3.5 is for the OC metric. The x-axis represents the number of overloading constructors in one class. It reveals that there are 24% classes which have more than one constructors. In both *Antlr* and *JHotDraw*, we found classes with 9 constructors. Typically, a java class may have multiple fields that need to be initialized when an instance of that class is created. Some fields have default values so that they can be ignored. However, Java does not provide named and optional parameters to specify the usages of the given inputs. Therefore, constructors with different parameter lists are needed to initialize different combinations of fields.

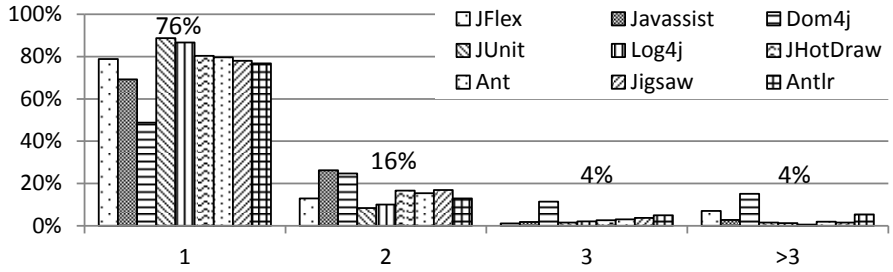


Figure 3.5: Fractions of classes with different numbers of overloading constructors

Figure 3.6 is the result of the OM metric. The x-axis represents the number of overloading methods for each *function*. Unlike constructors, methods are much less likely to be overloaded and there are only 6% of functions with multiple implementations. In *JUnit*, the assertEquals function has 20 overloading methods. Among them, many methods evaluate the equality between two primitive values, because primitive types cannot be handled uniformly in Java. This is also likely to happen in our daily development.

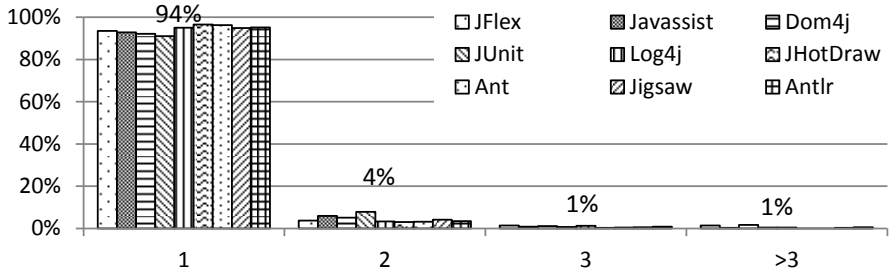


Figure 3.6: Fractions of functions with different numbers of overloading methods

Updated Collection Fields (UC)

In Figure 3.7, the x-axis represents the number of different locations updating the same collection field. It shows that almost half of the collection fields are updated at more than one location in a class and 12% involve more than three locations. Compared to other metrics, the fraction of single locations is relatively low, because a collection field is typically accessed by multiple producers and consumers. The maximum case contains 21 places modifying the same collection field. In *Dom4j*, we found a class that delays to perform all "events". Instead, it uses a list to record events in corresponding methods and dispatches the list of events to a dedicated handler.

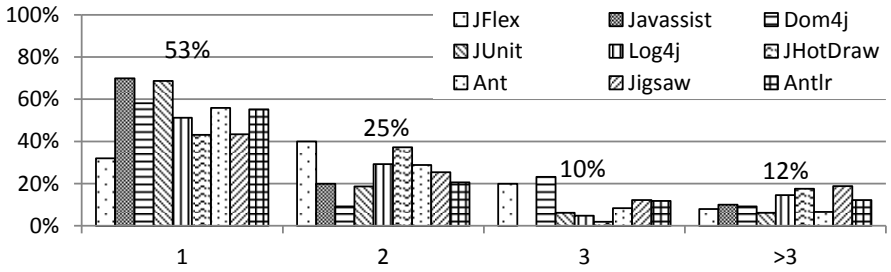


Figure 3.7: Fractions of collection fields updated by different numbers of locations

Dereference Operations per Line (DL)

In Figure 3.8, the x-axis represents the number of dereference operations at one line in the source code. We found 22% lines with dereference operations have more than one such operation. In *Log4j*, a line with 10 dereference operations are performed on 6 different receivers. If a *NullPointException* occurs on that line, there are 6 suspicious places. A threat to the validity of the statistics is that our approach cannot distinguish different receivers. A line can invoke multiple methods on the same receiver.

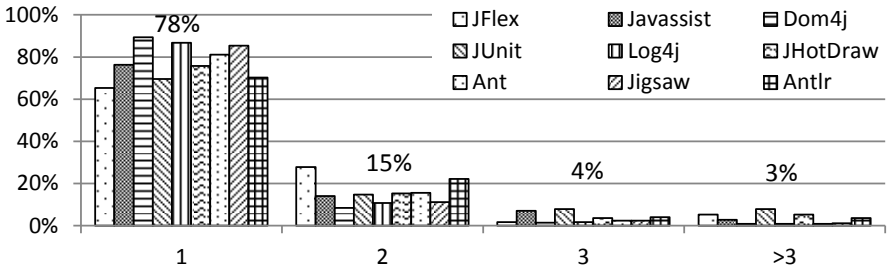


Figure 3.8: Fractions of lines with different numbers of dereference operations

Advice Compositions (AC)

In figure 3.9, the x-axis represents the number of statically woven advices at one JPS. It shows that there are 22% of JPSs advised by more than one advice. Though the general trend of fractions is decreasing when the amount of advices in one composition is increasing, the fractions of compositions with 2 advices in *Spacewar* and 3 advices in *HealthWatcher* are outstandingly high. In *Spacewar*, a **before** advice and an **after** advice use the same named pointcut and the pointcut selects 16 JPSs. Therefore, 16 JPSs are advised by at least 2 advices. Similarly in *HealthWatcher*, a **before** advice, an **after returning** advice, and an **after throwing** advice use the same named pointcut.

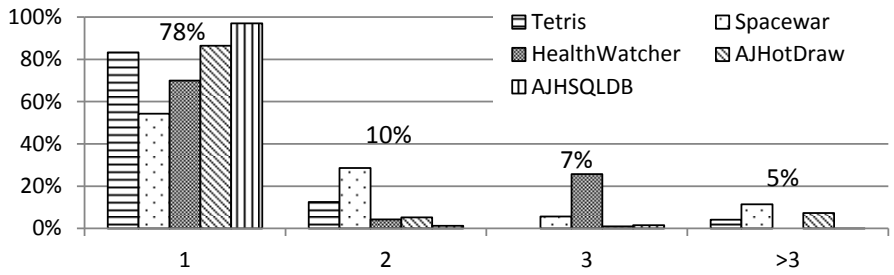


Figure 3.9: Fractions of JPSs affected by different numbers of advices

3.5.5 Summary

Table 3.3 summarizes the aforementioned measurements. For each metric, it lists the percentage of cases that involve more than one locations and the maximum number we found in the analysis. The statistics of metrics MF, OC, OM, UC, and LD are for OO programs. Except OM, they show that there is a significant portion of cases involving more than one source location. For these cases, debugging would require repeated manual steps like setting a breakpoint at each location. Metric AC is for AO programs and its statistics shows that the probability of involving multiple advices is high. As we have discussed in Section 3.2.5, selecting a specific composition is difficult. Programmers can save a lot of effort if a specific composition can be selected by a breakpoint.

The significant portions of cases that involves more than one locations reflect the value of using the BPL solution, not to mention that the worst cases of MF, OM, UC, and LD even exceed 20. The BPL solution programmatically removes the repeated manual effort, which includes finding locations, setting breakpoints, managing breakpoints, etc. It also saves programmers effort in analysing irrelevant suspensions, which are decreased or even eliminated at runtime.

Name	Acronym	Involving more than one locations (%)	Maximum
Modified fields	MF	34	30
Overloading Constructors	OC	24	9
Overloading Methods	OM	6	20
Updated collection fields	UC	47	21
Dereference operations per line	DL	22	22
Advice Compositions	AC	22	5

Table 3.3: A summary of the measurements.

3.6 Comparisons of Debugging Processes

This section shows the processes of debugging two examples in the traditional solution that uses line breakpoints, the program solution that uses auxiliary debugging program, and the BPL solution respectively. The description of each process may contain preparations before and a short summary after the concrete debugging steps.

3.6.1 Debugging Action Compositions

In this example, we add a new requirement to the **Scenario 4**, which is about recording execution history. As Listing 3.15 shows, a closed stream can be reopened before it reads data in the context of the vital parts of the program.

```
1  aspect SafeStream {
2    pointcut VitalPart() : ...;
3    Object around(Stream s) :
4          call(* Stream.read()) && target(s) && cflow(VitalPart()) {
5      if(s.isClosed()) { s.open(); }
6      Object result = proceed();
7      return result;
8    }
9  }
10 Stream stream;
11 ...
12 stream.close();   // the root cause
13 ...
14 stream.read();  // an exception is thrown.
```

Listing 3.15: An aspect with an advice that checks whether a Stream is closed and a program slice that performs operations on a Stream

In Listing 3.15, the **around** advice (lines 3–8) in Listing 3.15 implements this new requirement. Lines 10–14 contain a bug, which is reading data from a closed stream. The ellipses mean that statements may not be close to each other. We use the italic font for the variables *stream* to indicate that they refer to the same object but may use different names. The symptom of the bug is observed at line 14 but the root cause is the premature close() at line 12. The purpose of debugging is to intentionally suspend the program at the root cause. In the following paragraphs, we describe how a typical debugging process is performed in different solutions.

The Traditional Debugging

Before starting the debugging, the programmer should consider where to put the breakpoints. Reading the exception message, she knows that the root cause must be an invocation of close(). A natural thought is setting a breakpoint to the call site of close(). However, there may be many invocations of close() in the program and setting breakpoints to each of them is troublesome. Therefore, putting the breakpoint to the body of close() and then tracing back to its call sites is more feasible.

Debugging steps:

1. Set two breakpoints to the body of methods close() and read(). Start debugging.
2. When the program is suspended, note the hit count of the breakpoint and the object identity of the Stream object. Resume debugging.
3. Repeat step 2 until the exception occurs. The first debugging session terminates.
4. Check the note and find out where the problematic Stream was closed by comparing the object identity. Record the corresponding hit count of the breakpoint set in close().
5. Delete or deactivate the breakpoint set in read(). Use the recorded hit count to restrain the breakpoint set in close(). Start debugging again.
6. When the program is suspended, it is in the execution of close() invoked by the root cause. The source, which is located at the second top stack frame in the stack trace, is the root cause.

Besides finding appropriate places to set breakpoints, this process spends significant effort on manually noting record and searching history.

The Program Solution

In this solution, the programmer realizes that using the program in Listing 3.6 will result in many false positives. The **around** advice interrupts the matching of the trace pattern "close read" because it calls Stream.open(). We see two ways of handling this. First, Stream.open() can be considered in the pattern. Second, and more straightforward, calls to Stream.read() which are advised by the **around** advice can be excluded. Suppose the programmer selects the latter way, the desired condition should contain the negation of the pointcut **cflow**(VitalPart()) (Listing 3.15, line 4).

Debugging steps:

1. Add the following aspect and tracematch to the program. Replace the ellipsis on line 11 with the definition of the pointcut VitalPart() (Listing 3.15, line 2). Set a breakpoint at line 13. Start debugging.

```
1  Aspect debugging {
2    private int hitcount=0;
3    before() : execution(* Stream.close()) {
4      hitcount++;
5      print(hitcount);
6    }
7  }
8  tracematch(Stream s) {
9    sym close before : call(* Stream.close()) && this(s);
10   sym read before : call(* Stream.read()) && this(s)
11     && !(cflow(...));
12   close read {
13     // set a breakpoint on this line
14   }
15 }
```

2. When the program is suspended, read the printed hit count produced by line 5. The first debugging session terminates.

3. Delete or deactivate the set breakpoint. Use the recorded hit count to set a breakpoint in the body of Stream.close(). Start debugging again.

4. When the program is suspended, it is in the execution of close() invoked by the root cause. The method execution, which is located at the second top stack frame in the stack trace, is the root cause.

This process automates the manual work in the previous process. The most effort spent concentrates on writing the program, especially the condition that excludes

the **around** advice. Designing such a correct condition may be non-trivial. For example, a condition requires to exclude or include multiple advices.

The BPL Solution

The programmer has the same flow of thought as she does in the program solution. She needs to exclude the calls to Stream.read() where the **around** advice is applied.

Debugging steps:

1. Name the **around** advice as "aroundAdvice" and define the following five breakpoints. Line 6 shows how the **around** advice is excluded. Method signature such as Stream.close() can be generated by using the user interface. Activate only close_readNoAround. Start debugging.

```
1 close(Stream s) : call(* Stream.close()) && target(s);
2 read(Stream s) : call(* Stream.read()) && target(s);
3 aroundAdvice() :
4     adviceexecution(SafeConnection.aroundAdvice);
5 read_NoAround(Stream s) :
6     composition(read, !aroundAdvice) &&
7     bind(read(s));
8 close_readNoAround(Stream s) :
9     path(close read_NoAround) &&
10    bind(read_NoAround(s), close(s));
```

2. When the program is suspended, close_readNoAround prints the following information on the console. The first debugging session terminates.

```
1 ********** close_readNoAround **********
2 matched path (close read_NoAround)
3 close examples\StreamTest.java(line 10, hitcount 3)
4 read_NoAround examples\MyLogger.java(line 30, hitcount 1)
5 --read examples\MyLogger.java(line 30, hitcount 2)
```

3. Delete or deactivate close_readNoAround. According to the printed information, activate the breakpoint close and configure the hit count with 3. Start debugging again.

4. When the program is suspended, the root cause is found.

This process overcomes the shortcomings of the previous two processes. It not only automates manual works but also constructs the condition in a straightforward way. The **composition**() expression is an intuitive way for expressing a certain action composition and it does not require a sophisticated analysis.

3.6.2 Debugging Dereference Operations

JabRef is an open source bibliography reference manager. We have scanned the commits in its subversion repository to find all the revisions with reports containing the keyword "bug". Revision #25 reports a fixed null pointer bug. Listing 3.16 and 3.17 show the buggy program and the revised program. Line 4, which tests whether frame.basePanel() is null, is added in the revised program. By reverse engineering, we can deduce that a *NullPointerException* is thrown in the execution of line 5 of the buggy program. There are two possible root causes, one is the field frame and another is the expression frame.basePanel(). The exception occurs on the line where frame is accessed the first time in method actionPerformed. Therefore, the possibility that frame stores a null value cannot be excluded. The programmer needs to check both dereference operations during debugging.

```
1 class SearchManager {
2   public void actionPerformed(ActionEvent e) {
3     if (e.getSource() == escape)
4
5       frame.basePanel().stopShowingSearchResults();
6     ...
7   }
8 }
```

Listing 3.16: A buggy program

```
1 class SearchManager {
2   public void actionPerformed(ActionEvent e) {
3     if (e.getSource() == escape)
4       if (frame.basePanel() != null) // added in the revised version
5         frame.basePanel().stopShowingSearchResults();
6     ...
7   }
8 }
```

Listing 3.17: A revised program

The Traditional Debugging

Debugging steps:

1. Set a breakpoint on line 5. Start debugging.
2. When the program is suspended, inspect the value of the field frame.

3. The frame is not null. Deduce that the expression frame.basePanel() returns a null value.

4. The root cause is found. Terminate debugging.

In this process, most effort concentrates on finding the expression that returns null. This effort increases with the number of dereference operations on the same line. The programmer has to inspect each dereference operation until she can decide which one is the root cause. There are usually two ways of inspection. One is to copy and evaluate an expression. Another is to repeatedly perform "step into" and "step return" and meanwhile note which dereference operation the debugger comes to.

Another solution is putting each problematic dereference operation in a separate line and rerun the program. Then, the line number from the error message indicates that the dereference operation on that line is the cause. However, the chopped format of code is not as readable as it was. After the bug is fixed, the code fragments need to be put back together. Similar to the other traditional solution, it does not scale well when the number of dereference operations increases. Besides, this option requires changing the source code, which is not generally possible.

The Program Solution

Debugging steps:

1. Manually code the following aspect, add it to the debuggee project, and set a breakpoint at line 9 in the added program. Start debugging.

```
1  public aspect ProgramSolutionAspect {
2    before(Object receiver) :
3         (call(* *.*(..)) || get(* *.*))
4         && target(receiver) && withincode(public void
5           SearchManager.actionPerformed(ActionEvent))
6         && if(receiver == null) {
7      int line = thisJoinPoint.getSourceLocation().getLine();
8      if(line == 5) {
9        // set a breakpoint on this line
10     }
11   }
12 }
```

2. When the program is suspended, inspect the variable **thisJoinPoint** and find out the signature of the method call or the field access.

3. The signature contains stopShowingSearchResults(). Deduce that the expression frame.basePanel() returns a null value.

4. The root cause is found. Terminate debugging.

This process automates the task of finding the root cause but complicates the way of setting breakpoints. The program describes the task and constructs a place for setting the breakpoint. It needs non-trivial designing and implementation effort.

The BPL Solution

Debugging steps:

1. Set a breakpoint with the following breakpoint declaration using the user interface to generate the **location**() expression. Start debugging.

```
1 bp() : checkNPE() && location(
2         "net\sf\jabref", "SearchManager.java", [5]);
```

2. When the program is suspended, the breakpoint prints the following information on the console. Deduce that the expression frame.basePanel() returns a null value.

```
1 ********** bp **********
2 Panel.stopShowingSearchResults() has a null receiver.
```

3. The root cause is found. Terminate debugging.

This process combines the advantages of the previous two processes. It not only automatically detects the root cause, but also requires only trivial effort for setting the breakpoint. This comparison highlights the great convenience and efficiency of using BPL.

3.7 Related Work

We categorize the related work into three groups, which are breakpoints, debuggers, and pointcut languages.

3.7.1 Breakpoints

Modern IDEs have developed some advanced breakpoints. The IntelliJ Java debugger supports temporal dependency between two breakpoints: If a breakpoint A depends on another breakpoint B, then A cannot be hit until B is hit. Visual Studio allows setting breakpoints on a specific call to a function by using the stack trace. Such a breakpoint suspends the program when the call stack is exactly the same as the one it was set on. Nevertheless, these advanced breakpoints are developed based on line breakpoints, which hardly show why breakpoints are placed there. BPL uses programs to specify breakpoints and the intention of using the breakpoints, like suspending the program at method calls or field accesses, are explicit.

Chern and De Volder [16] proposed the control-flow breakpoint to suspend the program according to the state of the stack trace. The control-flow breakpoint can specify that an event should or should not occur in the control-flow of another event. The breakpoint specification can be gradually refined at runtime until only the expected suspensions occur. In our approach, control flow refinements result in a breakpoint declaration which consists of multiple *cflow* expressions. In addition to the control flow, we also support sequential execution patterns.

The stateful breakpoint [11] allows programmers to suspend the program when some line breakpoints are hit in an expected order and certain values at those hits are coincident. A stateful breakpoint consists of three parts: a set of named line breakpoints, variables bound by the line breakpoints, and an execution trace composed by the names of the line breakpoints. Our composite breakpoint has two main differences to the stateful breakpoint. First, we provide more flexible ways specifying conditions on the bound variables by explicitly using `if()`. Second, a primitive breakpoint in BPL, once it has been defined, can be referenced by multiple composite ones. A primitive breakpoint is not reusable in stateful breakpoints.

3.7.2 Debuggers

Bugdel [79] is an AO debugging system in which programmers can set AO breakpoints by using dedicated graphical user interfaces. It can insert statements at a breakpoint to specify what to do when the breakpoint is hit. Its breakpoint model is join-point-shadow based, which is more fine-grained than line breakpoints. However, breakpoints defined in Bugdel are independent from each other. Thus, they cannot be used to compose higher-level breakpoints.

JavaDD [32] is a declarative debugger on which programmers can perform queries over the recorded execution history. Like other query-based debuggers, JavaDD records all the salient events, such as method calls, or field assignments,

at runtime. In our approach, the breakpoint declarations are similar to queries but they are written before and applied to the following execution. Our approach records only the interesting values and, thus, programmers cannot query values which were not recorded.

Ducassé [22] complained that line-based breakpoints do not have semantics and, therefore, proposed Coca, which is a debugger using only events related to source abstractions as queries. Our approach does not throw away line-based breakpoints, because they are sometimes easier to be specified and more straightforward than breakpoints using source abstractions. Besides, Coca uses Prolog, which is completely different from the language of the debuggee programs, as the query language. Learning Prolog increases the cost of using debugging facilities. Our approach aims at minimizing the learning effort by using a pointcut language using abstractions that are natural to programmers of object-oriented and aspect-oriented programs, namely one similar to AspectJ.

3.7.3 Pointcut Languages

Tracematch [3] uses regular expressions consisting of references to primitive pointcuts to specify an expected execution trace. It supports free variables in specifying a trace. Therefore, a matched trace depends on not only the order of events but also the associated variables of the events. The design of the composite breakpoint in BPL is greatly inspired by Tracematch. However, there are two fundamental differences. First, a *tracematch* is a standalone unit and other *tracematch*es cannot reuse its members including definitions of primitive events. In BPL, a primitive breakpoint can be referred by any number of other breakpoints. Second, there are several event kinds, such as *before* and *after returning*, in Tracematch. BPL is only interested in *before* because we want to suspend the program before the interesting events occur.

Oarta [58] extends AspectJ with features, which are similar to some of BPL. It supports *named advices* by putting a name in the declaration of an advice. It also allows declaring precedence at the advice level. Our approach does not change the syntax of AspectJ and it names advices by using Java comments. Besides, our composition specification targets finding advice compositions at runtime instead of defining precedence rules for weaving.

3.8 Conclusion

In this chapter, we identified five scenarios of using breakpoints. These scenarios are frequently encountered but not supported sufficiently by existing breakpoints. Programmers need to manually perform some repetitive tasks, thus debugging

efficiency is decreased.

Targeting all scenarios, we proposed a breakpoint language (BPL) which models the breakpoint as first-class values. Breakpoints are named and they are defined by AspectJ-like pointcuts which use comprehensible source-level abstractions. We devised five completely new pointcut designators and improved two of AspectJ's pointcut designators. In our language, primitive and composite breakpoints are treated uniformly and the composition level can be infinite. It is the first language to support selecting join points with a specific advice composition. Furthermore, a preliminary evaluation shows that the performance of the BPL solution is better than existing techniques for conditional breakpoints.

To validate our work, we performed a code analysis on 9 Java projects and 5 AspectJ projects. The results of the analysis shows that the BPL solution can benefit programmers in a significant portion of cases. Besides, we illustrate the usage of our approach by means of two example walkthroughs. The examples show that BPL has the following advantages over the traditional debugging and the approach using other languages.

- It allows programmers to describe the logic relationship between multiple breakpoints with succinct code.
- It allows using pointcut **composition**() to express an advice composition in a straightforward way.
- It automatically records and prints information, such as the source location and the hit count, of a hit breakpoint and its referenced breakpoints. The information is helpful for localizing the root cause in an additional debugging sessions.

The pointcut **composition**() can also be used for other purposes. For example, it can verify whether a certain advice composition exists or not in the program. Another example is handling the fragile pointcut problem. Sometimes, changes to a pointcut may unexpectedly exclude or include some join points. Therefore, behavior occurring at these join points becomes undesired. To find the join-point differences, the following breakpoint declarations can be used.

```
1 oldOne() : adviceexecution(someAzpect.oldAdvice);
2 newOne() : adviceexecution(someAzpect.newAdvice);
3 inOldNotInNew() : composition(oldOne, !newOne);
4 inNewNotInOld() : composition(newOne, !oldOne);
```

According to the above specification, breakpoint inOldNotInNew is hit on the excluded join points and breakpoint inNewNotInOld is hit on the newly included ones. In this way, the program execution is suspended at the join points which

are (not) advised in both the old and the new versions of the program. The two breakpoints narrow down the potential places causing the undesired behavior.

4

Trace-based Debugging for Advanced-Dispatching Languages

4.1 Introduction

A failure usually does not appear immediately after the defect has been executed. Marc Eisenstadt [26] calls this phenomenon *cause-effect chasm*. A survey [35] in 2002 pointed out that over 50 percent of the debugging difficulties resulted from the time and space chasm between failure and defect or inadequate debugging tools.

Interactive debuggers, as discussed in Chapters 2 and 3, are powerful in accessing the runtime information. However, the cause-effect chasm reflects that the nature of debugging is a backward process, which is from the observed failure to the defect. But in interactive debugging, to inspect a state in the past execution, programmers have to terminate the current debugging session, start a new one, and suspend the debuggee program earlier than the last session. This may require several iterations until the defect is found.

Trace-based debugging is a debugging technique that records interesting runtime information during the program execution and performs debugging by using the recorded information after the execution. Trace-based debugging allows programmers to navigate the runtime information backwards. Therefore, it naturally fits the necessity of backward tracking in debugging. However, few works have explored trace-based debugging for AD languages. To our best knowledge, the TOD extension [70] is the only work in this area but it is only for AOP, which is classified as one type of AD mechanism.

In this chapter, we present ALIA-TBD, which is a generic trace-based debugger for AD languages. The eleven AD-specific debugging tasks proposed in Section 2.2 are also applicable for validating the trace-based debugging. Similar

to the AD interactive debugger, ALIA-TBD needs to collect information at runtime and present it through a user interface. Therefore, its structure is basically similar to the one proposed in Section 2.4, except that it requires to store the collected information.

The overall structure of ALIA-TBD consists of two parts: a back-end and a front-end. The back-end collects and records runtime information during the execution of the AD program. Two key components in the back-end are trace model and storage model. The trace model defines which information needs to be collected at runtime. The storage model defines how the collected information is structured on the storage media. The runtime information, which is also interchangeably called *execution trace* in this chapter, is written to a storage media. The front-end provides a user interface to visualize, navigate, and query the execution traces. To present the abundant information compactly, we have chosen to use a tree map to visualize the execution trace. Information is layered in different levels according to the calling depth. We provide a top-down navigating strategy—from higher levels to lower levels. In addition to step-wise navigation, programmers can perform queries to have overviews of the trace. To decrease the effort for programmers of learning how to use queries, we have implemented a library with several query templates that need to be frequently used.

The primary contributions of our work are listed below.

- A trace model for AD languages. It identifies the required activities that need to be captured in the execution of AD programs. The activities are mainly related to reconstructing object states and tracing AD activities.

- The tree-map visualization of AD-specific traces. Our work is the first attempt to visualize runtime information in a tree map. The visualization allows programmers to conveniently navigate and explore AD activities.

- The implementation of stepping actions in XQuery. This allows programmers to control the debugger in a programmatic way and customize new stepping actions. An advantage of XQuery is that it does not require compilation.

The remainder of this chapter is structured as follows. Section 4.2 describes two AD-specific debugging scenarios to motivate this work. Section 4.3 introduces the design and implementation of the back-end. Section 4.4 describes the front-end and Section 4.5 introduces the query library. Sections 4.6 and 4.7 validate our work by an evaluation and a case study. Sections 4.8 and 4.9 discuss related work and conclude this chapter respectively.

4.2 Motivation and Requirements

In this section, we motivate our work by analysing problems in debugging AspectJ programs. These problems can be generalized and they may also occur in debugging programs written in other AD languages.

When introducing a new aspect into a system, the aspect can influence the execution of both the base program and other aspects. Rinard et al. [72] classified the interactions between advice and methods into two main categories: *control-flow interaction* and *data-flow interaction*. The control-flow interaction represents how an advice changes the execution of an advised method. The data-flow interaction represents how an advice uses data shared with the advised method. Many works [36, 45, 57, 63] have discussed *aspect interference*. The newly introduced aspect can also interfere with the control-flow and data-flow of the existing aspects.

The following subsections discuss two concrete debugging scenarios with respect to the aforementioned two types of changes. We found that these scenarios are difficult to be debugged without using the execution history. To illustrate, we use the base program in Listing 4.1 for both scenarios. It is an Account class and its field value is decreased when the customer withdraws money.

```
1 public class Account {
2   private float value;
3   public void withdraw(float amount) {
4     value −= amount;
5   }
6 }
```

Listing 4.1: An example base program.

4.2.1 Control-flow Change

An advice can augment, narrow, or replace the execution of the advised program. For example, a *before* advice always augments the method execution, because it only adds functionality to the method; an *around* advice can replace the execution of other advices with lower-priorities at a shared join point. However, such control-flow changes are not always desired.

Take Listing 4.2 for example, the around advice conditionally proceeds to advice or method. This aspect is supposed to constrain the amount of the withdrawn money. When the amount exceeds a threshold, the transaction cannot be committed. However, the programmer specified the condition on line 5 wrong.

```
1  public aspect TransactionLimits {
2    Object around(float amount) :
3      call(* Account.withdraw(float)) && args(amount) {
4      //FIXTO: if (amount <= THRESHOLD)
5      if (amount >= THRESHOLD)
6        return proceed(amount);
7      else
8        return null;
9    }
10 }
```

Listing 4.2: An aspect with an **around** advice that conditionally invokes proceed().

In this scenario, programmers observe that expected withdraw operations are not performed. The cause is that the **around** advise unexpectedly does not call **proceed**(). This problem can be generalized to a wrong composition, which consists of a sequence of advice executions and the advised computation, at a join point. The composition sometimes cannot be statically determined due to the runtime tests in pointcuts. To analyse a composition, programmers need to inspect the complete execution at a join point.

Furthermore, programmers may want to verify whether all the around advices handle the **proceed**() call properly. Hence, programmers need to find all the join points with required compositions in the execution. Examples of such join points are those advised by an **around** advice without calling **proceed**() and those where two different advices are applied. Without using the execution history, it is difficult for programmers to find those join points.

4.2.2 Data-flow Change

An advice execution can modify context variables that will be used or has been used by the advised method. For example, AspectJ allows to modify or replace arguments and/or the returned value of a method call by using an around advice. This may lead the program execution to an unexpected state. In such situations, the programmer needs to know whether the problematic variables are modified by advices.

Suppose the aspect TransactionCost in Listing 4.3 runs with the base program, the **around** advice deducts the transaction fee from the withdrawn money on line 5. However, the calculation of the transaction fee is wrong. At a certain point after paying the transaction fee, the programmer observes that the account state is unexpected. To debug this problem, she first inspects the execution of withdraw() but finds no defect in it, then notices that the method is advised by an aspect, and finally finds the defect in the **around** advice.

A context variable can be accessed by multiple advices at one join point. Analysing how the variable is modified requires to know how the multiple advices are composed and inspect the whole execution of such an composition. Without the execution history, the programmer need to perform these tasks manually.

```
1 public aspect TransactionCost {
2   Object around(float amount) :
3     call(* Account.withdraw(float)) && args(amount) {
4       //FIXTO: amount -= amount*RATE;
5       amount -= RATE;
6       return proceed(amount);
7   }
8 }
```

Listing 4.3: An aspect with an **around** advice that alters the state of the advised program

4.2.3 Requirements

When debugging programs written in main-stream languages like Java, programmers may keep asking questions, like "where is the previous assignment to this variable?", "why is this method executed?", etc. These questions are important to be solved and they are supported by most of the trace-based debuggers. However, they are not the focus of this chapter. We need to identify which AD-specific questions need to be supported in our approach. The two debugging scenarios intuitively show which AD-questions might be needed to find defects. Examples of such questions are:

- Which computations were skipped by advices?
- Which contexts of computations had been modified by advices?
- Which advices were actually applied to this join point?
- What was the evaluation result of this predicate?
- Which join points were advised by both advices A and B?

Requirement 1 : Collecting Information. To answer the above questions, our debugger needs to trace the following information. The first two points are applicable to all programs and the last two are AD-specific.

1. The static information of each entity in the source code, which includes source locations, identifiers, signatures, static structures, etc.

2. The runtime information of execution, which includes variable values, object states, identities of entities, the interrelationships between runtime entities, the state changes, etc.

3. The interactions between the base program and the AD entities, which include the join points where an advice is applied, the compositions of advices at a join point, the control-flow and data-flow modifications to the base-program in *around* advices, etc.

4. The activities of AD entities, which include predicate evaluation, precedence evaluation, action execution, etc.

Requirement 2 : Accessing Information. Our debugger needs to organize and provide the information in a way that programmers can efficiently find what they need. This requirement can be further divided into three sub-requirements.

1. The base program is not aware of the existence of AD entities. Therefore, the debugger should be able to present information of the program as if no AD entity has been applied. On the contrary, the debugger should also be able to present all the information associated with AD entities.

2. A program can typically generate thousands of events within a few seconds. It is not possible to present all of them on one screen. The debugger should be able to present part of the information at a time according to the programmers' focus.

3. The debugger should allow programmers to specify questions. The way of asking questions should be flexible so that programmers can customize their navigation on the recorded information.

4.3 Back-end Design and Implementation

4.3.1 Model Design

To fulfil Requirement 1, ALIA-TBD needs to collect the required information during the execution and store it in a way that is convenient for future accesses. To avoid duplication in storing, each entity needs a unique identifier to be referred. Therefore, we propose a trace model, an identifier model, and a storage model in this section. The trace model defines which information needs to be collected at runtime. The identifier model defines how different entities are identified in a uniform way. The storage model defines how the collected information is structured on the storage media.

Some trace-based debugging approaches, such as TOD [71] and ODB [53], use the term *event* to refer to the fundamental computations captured at runtime. In this work, we use *join point* instead, because join points are modelled as first-class objects in our execution environment NOIRIn [10]. Besides, we want to distinguish our work from existing trace-based debuggers.

Trace Model

Figure 4.1 presents a simplified AD-specific trace model in a UML class diagram. The join points supported by NOIRIn are put in the hatched area. We add three types of join points, which are *ThrowJoinPoint*, *ArrayWriteJoinPoint*, and *LocalVariableWriteJoinPoint*. The AD-related activities, which occur during the execution of NOIRIn, are also included in the trace model, and they are put in the cross hatched area. The trace model includes join point types for all entities defined in LIAM except *ScheduleInfo* and *Pattern*. Instances of *ScheduleInfo* and *Pattern* are only evaluated once during the deployments of their corresponding attachments, and the evaluation results cannot be changed at runtime.

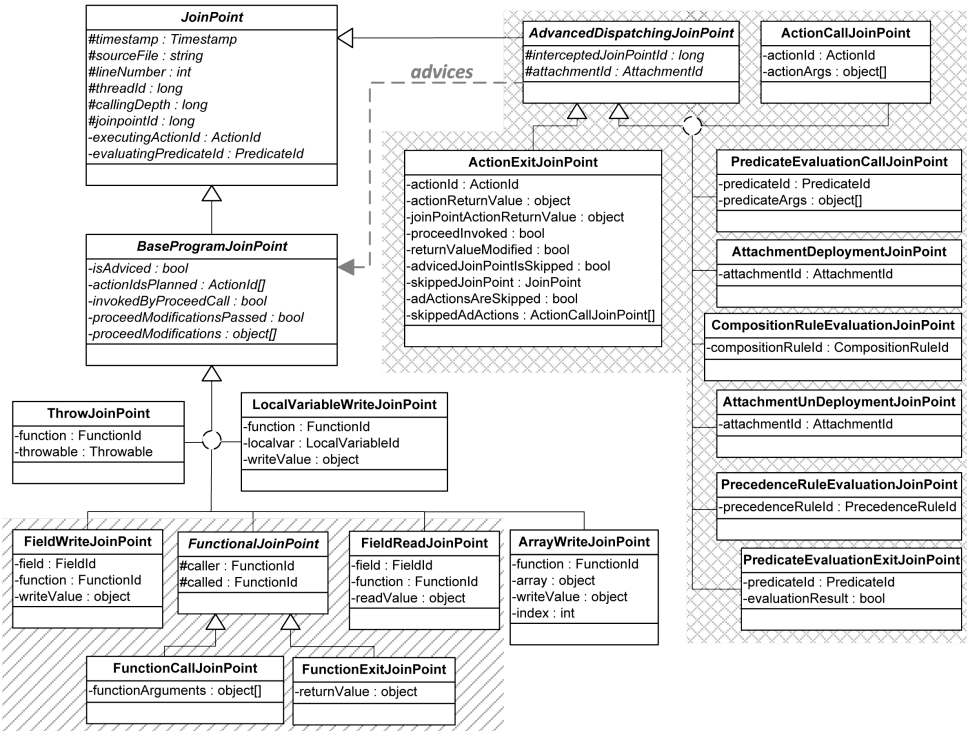


Figure 4.1: A simplified UML class diagram of the trace model.

The class *JoinPoint* is the root class and all other join points inherit it. It

contains fields representing common information, such as the source location of a dispatch site. The class *FunctionalJoinPoint* contains the caller and the callee of the corresponding function. It has two subclasses *FunctionCallJoinPoint* and *FunctionExitJoinPoint*, which represent the call to and the return from a function respectively. The former contains the passed arguments and the latter stores the returned object.

The class *AdvancedDispatchingJoinPoint* reifies an AD-specific activity. The execution related to *Action* and *Predicate* entities may include a subroutine. Therefore, the trace model also defines join point types for their calls and exits. The class *ActionExitJoinPoint* reifies the exit of an *Action* and it contains fields that represent different types of modification to the advised computation. For example, if an *Action* skipped the advised computation, the field *proceedInvoked* is false and the field *skippedJoinPoint* refers to the skipped join point. In LIAM, *precedence rules* define the priorities between several AD entities, such as the declare-precedence statement in AspectJ. At a dispatch site, precedence rules need to be evaluated to prioritize the applied AD entities. Even though most implementations of AD languages statically evaluate precedence rules, we include *PrecedenceRuleEvaluationJoinPoint* as a dynamic activity in the trace model to allow querying it.

Identifier Model

We use an *identifier model* to identify every entity in the recorded trace. It separates the abstractions of entity identities from the trace model. Figure 4.2 presents a simplified identifier model in a UML class diagram. Each instance of an *Id* subclass is used as both a reference to and a representation of a source construct. Take the class *FunctionId* for example, it uniquely identifies a function execution. Suppose there are two distinctive objects `x` and `y` and an instance method `m()`, `x.m()` and `y.m()` are represented by two different instances of *FunctionId*. As the fields of *FunctionId* suggest, each instance contains the static information, such as its modifiers, about the function `m()`.

Storage Model

We extend the model proposed by De Pauw et al. [68] with AD-specific concepts. Our storage model produces a *call tree* representation of a program execution. Figure 4.3 shows a simplified call tree for the execution of the program in Listing 4.4. For simplifying the explanation, each node is labelled with a number. The nodes ①, ②, ③, ⑦, and ⑨ are inner nodes and they represent subroutine calls. The nodes are layered in different levels, which correspond to *calling depths*. The path from the root to a node in the tree is the *call stack* of that node. For example, the call stack of node ⑩ is : "main(args)/Main.init()/n()/o()".

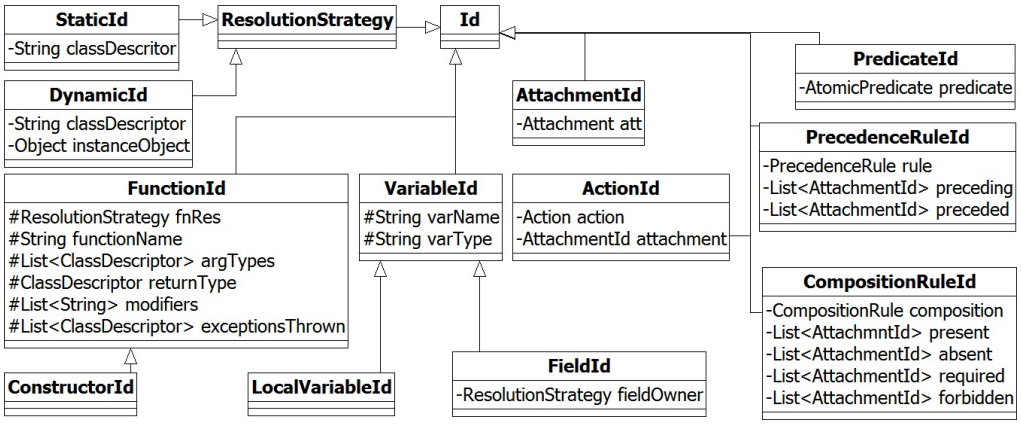


Figure 4.2: A simplified UML class diagram of the identifier model.

```
1  public class Main {
2    public static void main(String args[]) {
3      new Main();
4    }
5    private static int f;
6    public Main() {
7      m();
8      n();
9    }
10   public int m() {
11     int a = 1;
12     int b = 2;
13     return a + b;
14   }
15   public void n() {
16     int d = 4;
17     o();
18   }
19   public void o() {
20     int e = 5;
21     f = 6;
22   }
23 }
```

Listing 4.4: An example of Java program.

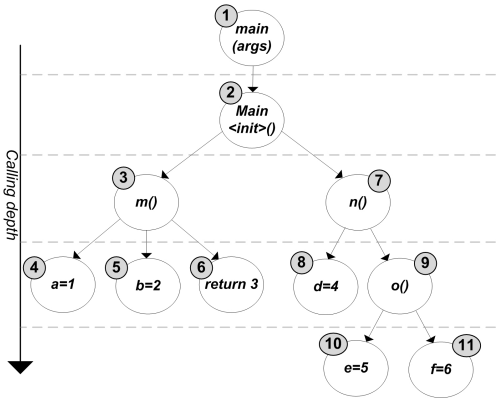


Figure 4.3: Call tree representation of the execution of the program in Listing 4.4.

Figure 4.4 depicts the simplified storage model in a class diagram. A *Trace* consists of a list of *JoinPoint*, whose structure follows the composite pattern [31]. The *CallJoinPoint* reifies a call to a subroutine and it consists of *JoinPoint*s that occurred in the execution of that subroutine. NOIRIn performs advices, and predicate evaluations in the same way as normal methods. Therefore, *CallJoinPoint* has subclasses *FunctionCallJoinPoint*, *ActionCallJoinPoint*, and *PredicateEvluationCallJoinPoint*. The *ExitJoinPoint* reifies the exit of a subroutine. Accordingly, it has subclasses for normal methods, actions, and predicate evaluations. The *OtherJoinPoint* in the dashed box is a pseudo class, which represents any other join point, such as *FieldWriteJoinPoint*.

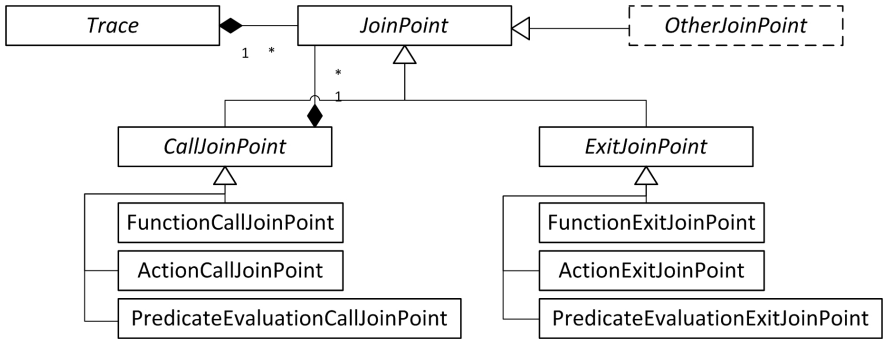


Figure 4.4: A simplified UML class diagram of the storage model.

4.3.2 Collecting and Storing Runtime Information

The trace model is implemented as an extension to NOIRIn [10], where AD concepts are modelled and run as first-class objects. We used three dedicated approaches for collecting runtime information defined in the trace model. First, NOIRIn instruments byte code to get call backs for join points in the base program. We hook call back method for collecting runtime information. Second, for join points, which are *ThrowJoinPoint*, *ArrayWriteJoinPoint*, and *LocalWriteJoinPoint*, not in NOIRIn's join point model, we instrumented byte code as NOIRIn does to get call backs. Third, for AD-specific join points which are generated during the execution of NOIRIn, we instrumented the interpreter in NOIRIn to collect information. In any case, when a join point is reached, collected dynamic information is passed to the storage model as entities defined in the trace model.

We chose to use XML to implement the storage model, because it is well supported by many techniques and tools. We implemented a trace writer to transform runtime trace entities to XML elements. The model structure starts

from the root element *Trace*, which contains of a list of *JoinPoint*s. As introduced in Section 4.3.1, the *CallJoinPoint* reifies a call to a subroutine, and thus it contains a list of *SubJoinPoints*.

The XML file stores values for the primitive entities and identifiers for objects. The trace writer manages a hash map that stores objects and their identifiers as key-value pairs. When an object needs to be serialized to the XML file, the writer searches the identifier for that object in the hash map. If nothing is returned, a new identifier is generated and stored with the object to the hash map.

4.4 User Interface

ALIA-TBD reuses BaseX [1] to parse the trace file and process queries. BaseX is an open source, light-weight, high-performance, and scalable XML database engine and XPath/XQuery 3.0 Processor. BaseX indexes the trace file during parsing to efficiently perform future queries on the parsed file. We also extend a module from BaseX to visualize the trace file in a tree map. It takes the source code and an XML-format trace as input.

A snapshot of ALIA-TBD is shown in Figure 4.5. Its four main components are *Source-code view*, *Tree-map view*, *Query editor*, and *Joinpoint-info view*, and they are labelled by numbers 1-4 respectively. Source-code view can display the source code and highlights a specified range of source code according to the selection in the Tree-map view. Tree-map view visualizes the recorded trace in a multi-staging tree map. Query editor is a text editor where queries can be specified. Joinpoint-info view displays the detailed information of a single join point in a tree viewer according to the selection in the Tree-map view.

4.4.1 Tree-map view

Tree map is a visualization for displaying hierarchical data by using nested rectangles. In our context, each rectangle represents a join point. For a *CallJoinPoint*, its rectangle is tiled with smaller ones that represent sub-joinpoints that occurred in its execution. For other types of join points, they are represented by leaf rectangles.

Take Listing 4.5 for example, its execution trace is shown in Figure 4.6. We use colored rectangles for AD-specific join points and grey rectangles for others. The deeper they are in the calling depth, the darker their color is. As the dashed arrow indicates, a join point at the top-left corner occurs earlier than one at the bottom-right conner. For example, the constructor of class `Main` calls two methods `m()` and `n()` sequentially in its body. Accordingly, the rectangle labelled

[1] See `http://basex.org/`

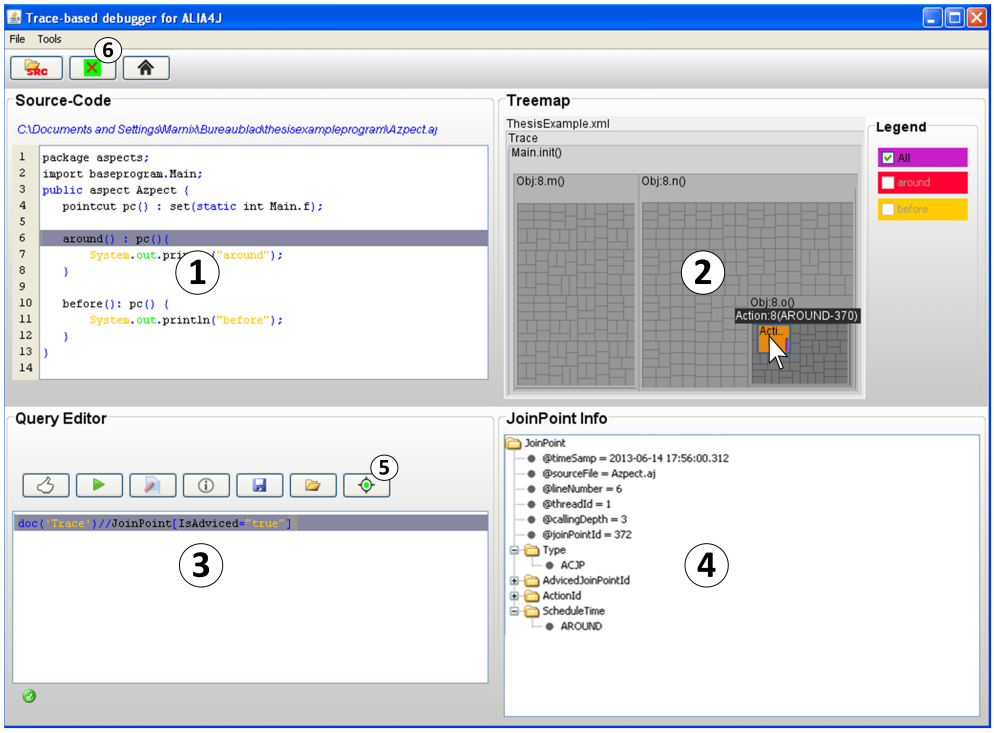


Figure 4.5: A snapshot of ALIA-TBD.

with *Main.init()* has two nested rectangles, which are labelled with *m()* and *n()* respectively, and they are placed from left to right.

```
1  public class Main {
2    private static int f;
3    public Main(){ m(); n(); }
4    public int m() { ... }
5    public void n() { ... o(); }
6    public void o() { f = 6; ... }
7    public static void main(String args[]){
8      new Main();
9    }
10 }
11 public aspect Azpect {
12   pointcut pc() : set(static int Main.f);
13   Object around() : pc(){ return proceed(); }
14   before(): pc() { ... }
15 }
```

Listing 4.5: An example of AspectJ program

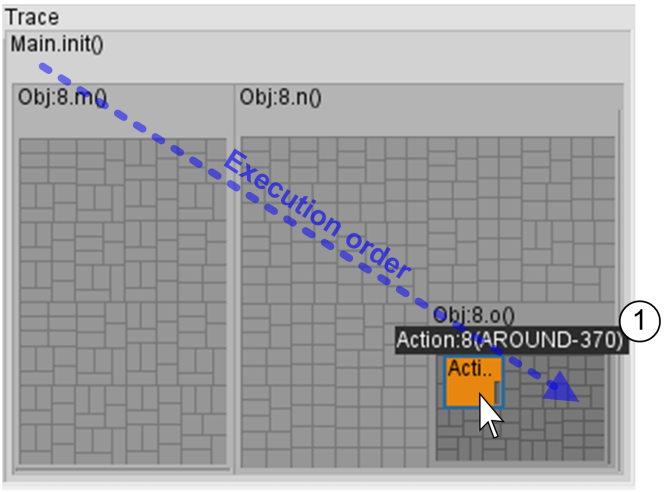


Figure 4.6: A snapshot of the tree-map view with an overview of an execution trace.

Figure 4.6 shows the initial overview of the trace and low-level join points are abstracted as empty rectangles. To inspect a low-level join point, programmers can double-click the corresponding rectangle and then the tree map of that rectangle substitutes the current visualization. Suppose programmers choose to inspect the join point as the mouse pointer shows, the visualization in Figure 4.7 comes in. Information that was missing in the overview is presented. The root rectangle represents the execution of the **around** advice and it contains the execution of the **before** advice. In this way, programmers can navigate information from the root to any leaf of the trace.

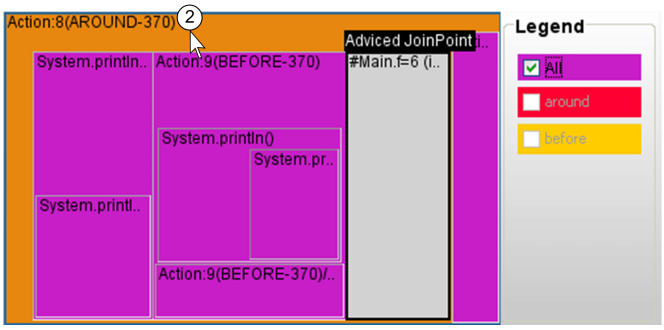


Figure 4.7: A snapshot of the tree-map view, which is zoomed in from Figure 4.6.

4.4.2 Join Point Representations

Rendering plain rectangles in the tree-map view does not help programmers to comprehend the execution trace and localize interesting join points. Therefore, we label each rectangle with a brief description of the corresponding join point. Due to the limited space of a rectangle, we only show the crucial information to roughly tell programmers what the join point is about. Take *FieldWriteJoinPoint* for example, we describe it in a form "<fieldOwner>.<varName>=<writtenValue>". Appendix B gives a detailed description of the representation for each join point defined in the trace model.

The brief textual description of a join point may still exceed the rectangle representing that join point. Programmers can hover the mouse pointer over the rectangle and a tooltip with the textual description is shown. As Figure 4.6 illustrates, a tooltip for the pointed *around* action is rendered. To examine all details of a join point, programmers can use the *Joinpoint-info view*. This view is updated when programmers select a join point in the tree-map view. It shows information of the selected join point in a tree structure.

4.4.3 Query View

Navigation with the tree map can only inspect one join point at a time. If programmers needs to inspect multiple join points that scatter throughout the whole trace or a single join point with a long execution path, they need to perform repeated manual navigation steps, such as selecting and zooming in rectangles. In addition to step-wise navigation, we support querying in ALIA-TBD. Querying allows programmers to express complex navigation requests on the trace. The query result is a list of join points, which satisfy the query and are located in different places in the trace.

The query results executed from the query editor are highlighted in the tree-map view. Each highlighted rectangle indicates that it contains at least one join point satisfying the query. The satisfied join point can be the one represented by the highlighted rectangle, or any of its nested rectangles. Projecting query results on the trace visualization increases the comprehensibility of the result, because it clearly shows the context of each satisfying join point.

ALIA-TBD also allows programmers to perform a step-wise query. Once a query has been executed, programmers can ask the tree-map view to show only the query results by clicking Button (5) in Figure 4.5. Then, the current query result becomes the search domain of further queries. In this way, programmers can focus one querying requirement at a time until the expected results are selected.

Programmers can also invoke stepping commands prepared in the query library, which will be introduced in Section 4.5. When a join point is selected

in the tree-map view, executing a stepping command transfers the focus to the reached join point. Stepping commands can be used either separately or with other XQuery constraint like a scope requirement. For example, suppose there is a loop with 100 iterations and the programmer wants to inspect the execution after the 75th iteration, she can add a constraint to step.next, as Listing 4.6 shows, to avoid many irrelevant manual steps.

```
1 step.next[LocalVariableId/VariableName="i" and NewValue/Int > 75]
```

Listing 4.6: A stepping command with an additional XQuery constraint.

4.5 Query Library

In Section 4.2.3, Requirement 2 requires that ALIA-TBD allows programmers to specify questions about the recorded runtime information. The questions can be formulated as queries and we chose XQuery as the query language. Programmers can specify any XQuery-conform query and the result is a list of XML elements satisfying the query. To improve the efficiency, tools typically provide a command for a frequently performed function. We also prepared a query library for frequently encountered tasks, especially for the stepping actions and the AD-specific queries.

Stepping with XQuery

As introduced in Section 1.1, a typical interactive debugger supports several stepping actions, such as step over and step into. Programmers only need to use a breakpoint to specify the starting point of suspensions, and then they can inspect the program states with full control of the execution by using the stepping actions. We use XQuery code to realize stepping actions. Therefore, programmers can use ALIA-TBD to analyse a program in a similar way as they use interactive debuggers. We implemented six types of stepping actions and Figure 4.8 shows a call tree as well as what each stepping action means. The types of stepping actions are not limited to these six. Programmers can customize any new type of stepping action with XQuery according to specific domains.

Step-next & Step-back These two stepping actions are represented by one step in the pre-order tree traversal forward or backward respectively. For example, starting from node (5), step-next goes to node (6) and step-back goes to node (4). The corresponding implementation in XQuery is shown in Listing 4.7. The current node, which is node (5) in this example, is represented by XML element

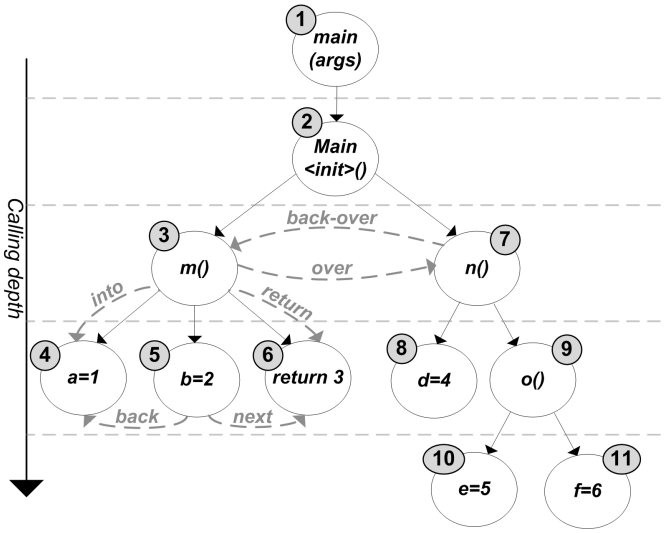


Figure 4.8: A call tree and stepping actions supported in ALIA-TBD.

jp. Keywords following and preceding are two XPath axes. An axis defines a node-set relative to the current node. The part JoinPoint[1] means to select the first element in the specified set.

```
1 (: STEP−NEXT :)
2 jp/following::JoinPoint[1]
3 (: STEP−BACK :).
4 jp/preceding::JoinPoint[1]
```

Listing 4.7: XQuery code for step-next and step-back

Step-over & Step-back-over These two stepping actions are represented by one step in the sequential traversal at the same calling depth forward or backward respectively. For example, starting from node (3), step-over skips its children and directly goes to node (7). Similarly, starting from node (7), step-back-over goes to node (3). The corresponding implementation in XQuery is shown in Listing 4.8. Keywords following−sibling and preceding−sibling are two XPath axes.

```
1 (: STEP−OVER :)
2 jp/following-sibling::JoinPoint[1]
3 (: STEP−BACK−OVER :).
4 jp/preceding-sibling::JoinPoint[1]
```

Listing 4.8: XQuery code for step-over and step-back-over

Step-into & Step-return Step-into and step-return goes to the first child and the last child of the current join point respectively. For example, starting from node (3), step-into goes to node (4) and step-return goes to node (6). The corresponding implementation in XQuery is shown in Listing 4.9. Keyword child is an XPath axis. Predicate last() refers to the last node in the selected node set.

Here, we define the *step-return* action differently from the traditional one that jumps to the first instruction after the method is returned. Our *step-return* is more atomic, because the tradition one can be substituted by sequentially composing our *step-return* and *step-next*.

```
1 (: STEP−INTO :)
2 jp/SubJoinPoints/child::JoinPoint[1]
3 (: STEP−RETURN :).
4 jp/SubJoinPoints/child::JoinPoint[last()]
```

Listing 4.9: XQuery code for step-into and step-return

AD-specific Queries

Queries can be formulated based on the AD-specific information that is included in the trace-model. Such queries can be useful to determine if AD entities behaved as expected during the entire program execution.

Querying the behaviour of predicates In AspectJ, an advice can be unexpectedly passed because of an unmatched pointcut. However, it is difficult to find where the problem occurs and analyse which specific atomic pointcut has failed, because it is impossible to directly observe an element that is absent in the execution.

This task can be easily achieved by using the queries shown in Listing 4.10. The first query (lines 1 and 2) searches all failed predicate evaluations. Code tr represents the root of the trace and $umJp refers to the unmatched join point. The second query (lines 4 and 5) is a more strict version of the first query and it searches all failed evaluations of a specific predicate.

```
1 tr//JoinPoint[Type="PEJP" and InterceptedJoinPointId=$umJp/@joinPointId and
2     EvaluationResult="false"]
3 (: Specific predicate :)
4 tr//JoinPoint[Type="PEJP" and InterceptedJoinPointId=$umJp/@joinPointId and
5     EvaluationResult="false" and PredicateId=predicateId]
```

Listing 4.10: XQuery code for searching failed predicate evaluations.

Querying the behaviour of actions In AspectJ, an *around* advice can bypass the original computation by not calling *proceed*. Section 4.2 has shown a debugging scenario for the control-flow change. Listing 4.11 presents a query that searches join points skipped by *around* advices. Line 1 stores all the *around* advices that do not call *proceed* in a variable $act_skipped. The value of attribute ProceedInvoked (line 2) is pre-computed at tracing time. Line 3 returns the join points skipped by the *around* advices found on lines 1 and 2.

```
1 let $act_skipped := tr//JoinPoint[Type="AEJP" and
2     ScheduleTime="AROUND" and ProceedInvoked="false" ]
3 return $act_skipped/SkippedJoinPoint
```

Listing 4.11: XQuery code for searching join points skipped by *around* advices

Querying precedence rules at a shared join point Since AD allows to advice a join point with multiple attachments, the execution sequence of these attachments may affect the outcome of the program. The missing of an explicit precedence declaration, among such attachments, can cause the program to behave unexpectedly, as demonstrated by Zhang and Zhao [91].

Listing 4.12 presents a query template to search join points, where multiple advices are applied but not all required precedence rules can be found. First, the shared join points of the trace are retrieved (lines 1-2). Then, the query iterates over each shared join point (line 3) and locates the corresponding *PrecedenceRuleEvaluationJoinPoint* (PRJP) (lines 4-5). Finally, if no corresponding evaluation of a precedence rule was found (line 6), the shared join point is added to the result set of the query (line 7).

```
1 let $shared_jps := tr//JoinPoint[IsAdviced="true" and
2   count(ActionIdsPlanned//ActionId) > 2]
3 for $shared_jp in $shared_jps
4   let $precedence := tr//JoinPoint[Type="PRJP" and
5     AdvicedJoinPointId=$shared_jp/@joinPointId]
6   where empty($precedence)
7 return $shared_jp
```

Listing 4.12: XQuery code for searching precedences evaluated at a join point.

4.6 A Performance Evaluation

4.6.1 Two Dependent Variables

The performance evaluation has two dependent variables, which are environment settings and join point types.

Environment Settings

To measure the portion of overhead caused by each component, we enabled our infrastructure part by part from a pure Java virtual machine (JVM) to the full functioning ALIA-TBD and we obtained the following four types of environment settings.

Original setting uses the officially released JVM.

NOIRIn setting adds NOIRIn on top of the original setting.

Trace-enabled setting adds ALIA-TBD on top of the NOIRIn setting. However, join points are collected at runtime but are not written to any storage.

Storage-enabled setting enables the information recording on top of the trace-enabled setting.

Join Point Types

Table 4.1 categorizes join points in ALIA-TBD into four types according to two orthogonal criteria: how the join point is collected and whether the join point contains sub-joinpoints. The two criteria are listed in columns and rows respectively. The table gives for each type a representative example and only the given join point examples are measured in the evaluation.

4.6.2 Results

With each different combination of the two dependent variables, we measured 4 baseline time in the original setting and 12 different kinds of overhead in total. In each single run, we use a loop with 100K iterations in source code to generate expected join points. The evaluation is conducted on a computer with Windows NT 5.1 and GB RAM. The version of the Java runtime environment is 1.6.0_24.

Table 4.2 gives all required data measured in evaluation. The first four row shows the time used for each combination in millisecond. The last row shows the sizes of traces generated in measurements of the storage-enable setting. Two

	Bytecode instrumentation	Interpreter instrumentation
Atomic join point	LocalVariable -WriteJoinPoint (LVWJP)	FieldWriteJoinPoint (FWJP)
CallJoinPoint	FunctionCallJoinPoint (FCJP)	ActionCallJoinPoint (ACJP)

Table 4.1: Four types of join points that are measured in the evaluation.

	LVWJP	FWJP	FCJP	ACJP
Original setting(ms)	0.29	0.38	0.35	0.35
NOIRIn setting(ms)	0.29	5076.68	4397.86	2752.19
Trace-enabled setting(ms)	69.85	5647.82	5250.91	3698.13
Storage-enabled setting(ms)	8576.84	15671.45	26175	40688.56
Trace Size(MB)	98	106	222	307

Table 4.2: Results of overhead measurements with different environment settings and join point types.

points should be noted in the table. First, LVWJP caused no performance overhead in NOIRIn setting, because it is not supported in NOIRIn. Second, we chose to reuse the baseline measurement of FCJP in original setting as that of ACJP, because the simulation of ACJP only includes a *TruePredicate* that is always evaluated to be true. Therefore, ACJP is interpreted exactly the same as a function call in NOIRIn.

According to the definition of the four types of environment settings, a later defined setting is always a superset of earlier defined ones. Therefore, an overhead caused by a later defined setting includes all overhead caused by earlier defined ones. Figure 4.9 intuitively shows the portions of overheads that each setting individually increases in the fully functioning ALIA-TBD. In each bar, the overhead increased by each setting is separately marked. The parts for the original setting is too thin to be visible. The black parts shows that NOIRIn has already caused a significant performance overhead. The light grey represents how overhead increased by the trace-model setting. LVWJP causes much less

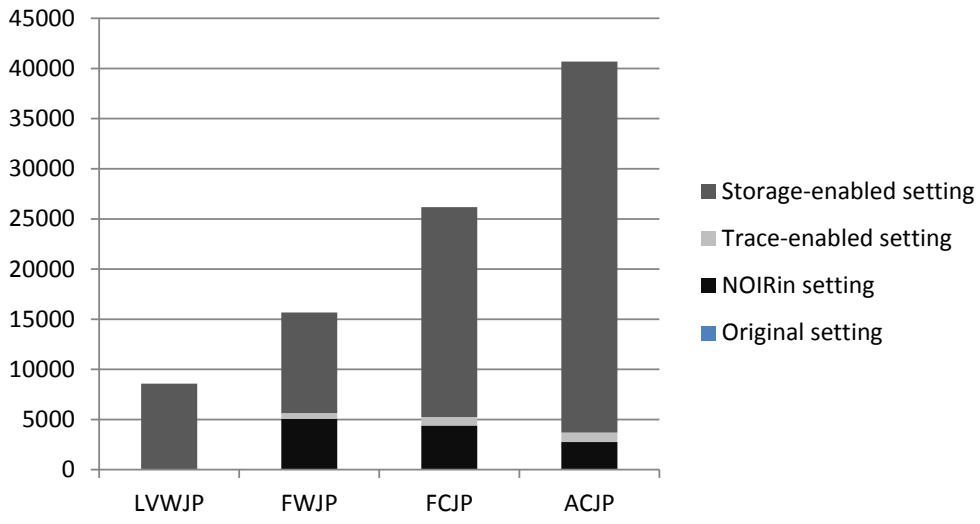


Figure 4.9: A chart that compares the overhead increased by different environment settings for each join point type.

trace-enabled overhead than the other three, because we extend NOIRIn for only collecting information of LVWJP, and thus NOIRIn does not provide dedicated dispatching on LVWJP. The storage-enabled setting, which is represented by the dark grey, increases the overhead tremendously, because the executions need to write all collected information to disk.

This evaluation analyses executions with dense join points and they represent almost the worst cases for our prototype. In practise, programs generate more sparse join points and not all the join points are required to be recorded. Therefore, the overhead caused by the trace-enabled setting would become much less than it shows in the evaluation. Besides, the current XML-format trace is excessively verbosity and we can substitute many frequently used strings with abbreviations. Therefore, we are optimistic about the future optimization.

4.7 A Case Study

4.7.1 Program and Defect Description

The programs simulate a supermarket. When a customer rings up a product, the product as well as its price is added to a list. The system performs addPrice(Product.getPrice()) to accumulate the prices of products. The supermarket allows two types of price reduction. When a product is on sale, its price is reduced by a percentage. Listing 4.13 presents aspect DiscountAspect, which contains an *around* advice that calculates the discounted price.

```
1 public aspect DiscountAspect {
2   public final static float DISCOUNT = 0.75;
3
4   Object around() :
5     call(public float Product.getPrice()) && withincode(...) {
6       return proceed() * DISCOUNT;
7   }
8 }
```

Listing 4.13: The source code of DiscountAspect

Customers can have 1 euro bonus when they bought enough products in the past. The discounted price can be further deducted by the bonus. Listing 4.14 shows aspect BonusAspect.

```
1 public aspect BonusAspect {
2   public final static float BONUS = 1;
3
4   Object around() :
5     call(public float Product.getPrice()) && withincode(...) {
6       return proceed() - BONUS;
7   }
8 }
```

Listing 4.14: The source code of BonusAspect

Listing 4.15 presents aspect OrderingAspect, which defines the precedences between the two aforementioned aspects. As the comment on line 2 of Listing 2 shows, the precedence declaration is problematic, because the DiscountAspect has a higher priority. Consequently, the final price of a product is first deducted by the bonus and then discounted. The expected ordering should be reversed.

```
1 public aspect OrderingAspect {
2   //FIXTO: declare precedence: BonusAspect, DiscountAspect;
3   declare precedence: DiscountAspect, BonusAspect;
4 }
```

Listing 4.15: The source code of OrderingAspect

4.7.2 Debugging with ALIA-TBD

Suppose the original price of the product is 10 euro, the problematic price after processing bonus and discount is $(10 - 1) \times 0.75 = 6.75$ euro, but it should be

$10 \times 0.75 - 1 = 6.5$ euro. When the programmer observes that the problematic price is recorded in the list, she can specify a query like the following listing after the execution.

```
1 let $symptom := $tr//JoinPoint[Type=''FCJP'' and
2       CalledId//FunctionName=''addPrice'' and
3       .//FunctionArgument[@argIndex=0]/Float=''6.75'']
```

After executing the query, the treemap view shows an overview of the trace with a highlighted rectangle, which contains a join point that satisfies the query. The programmer keeps zooming in the highlighted rectangles until she finds a rectangle representing a call to Product.getPrice().

From the colouring of rectangles, she observes that the execution of advice *Discount.around* embraces the execution of advice *Bonus.around*. She concludes that the precedence between these two advices is problematic. Then, she imports the query template, which is introduced in Section 4.5, to find whether a precedence rule was applied at the current dispatch site. Finally, she can locate and fix the problematic precedence declaration in Listing 4.15 according to the query result.

In interactive debugging, suspending the program at the place where the price (6.75 euro) is returned is not difficult. However, the following steps that require to inspect how the price was calculated is impossible, because interactive debugging can only perform forward analysis. Furthermore, the ordering of two advices matters in the price calculation. The feature of locating related precedence rules have never been supported in any other debuggers, except ALIA-based debuggers which are ALIA-TBD and the interactive debugger introduced in Chapter 2.

4.8 Related Work

To our best knowledge, the TOD extension [70] is the only existing work about trace-based debugging for AOP. It is built based on TOD [71], which is an omniscient debugger for Java. To identify AO-specific activities, such as aspect instance selection and advice execution, in the recorded execution history, it uses a tagging scheme. AO-specific activities can be completely folded or expanded in the view of the history. Therefore, programmers can inspect the history at appropriate intimacy levels. Besides, it provides an aspect mural view to highlight the activity of aspects in the history. We list the main differences between the TOD extension and ALIA-TBD.

- The TOD extension works on the woven code which does not preserve AO-specific information, such as precedence rules. Therefore, it cannot fully restore the runtime information to source-level abstractions. ALIA-TBD is

developed based on a ALIA4J framework [10], which models AO-specific concepts as first-class values. It also records and present information in AD-specific terms.

- The TOD extension serves only AOP, which we classified as one type of advanced-dispatching (AD) mechanism. ALIA-TBD targets AD languages and, thus, it is more generic.

- The TOD extension stores the runtime information in a woven form, that means it does not separate AO-specific information from the rest. Therefore, we can deduce that its underlying storage model does not contain AO-specific attributes. We use a dedicated AD-specific storage model to record information.

- The TOD extension provides limited ways for querying AO-specific information in the recorded execution history. ALIA-TBD allows programmers to perform arbitrary queries on the recorded trace.

- The TOD extension uses a mural view where all recorded events are presented at one dimension. We use a tree-map visualization to divide information into different levels.

In addition to the TOD extension, we categorize other related work into three groups, which are respectively associated with the three main features of ALIA-TBD: trace recording, query-based debugging, and trace visualization. In the following subsections, we discuss related work of each group.

4.8.1 Related Work of the Trace Recording

Chronon[1] is a commercial debugger that creates exhaustive traces. It mainly uses three approaches to decrease the performance overhead. First, it does not record traces for libraries including Java libraries. Second, it creates a static template of the trace before running and records only the information that is different from the template at runtime. Third, it employs dedicated threads running in the background to process trace information.

IntelliTrace[2] is a debugger developed by Microsoft and it records information on demand to decrease the runtime overhead. For example, it records program states only at the places where programmers set breakpoints in the source code. This requires programmers not to miss any required breakpoint. Otherwise, they

[1]See `http://chrononsystems.com`.
[2]See `http://msdn.microsoft.com/en-us/library/dd264915.aspx`

need to place additional breakpoints and run the program multiple times until all required information is provided.

There are some debuggers, such as ODB [53] and Unstuck [40], that use in-memory storage. The storage keeps only the latest information in a limited memory. This approach significantly decreases the runtime overhead of trace-based debugging. However, if the information generated during the cause-effect chasm exceeds the available memory, it increases the difficulty of fixing the bug because the provided information is not complete.

Above mentioned works weigh differently in information completeness and performance. These two requirements are conflicting and we cannot judge which one is more crucial. Researchers need to weigh them properly in their work according to their specific goal and context. The current stage of our work does not consider performance as a prior requirement. Therefore, our implementation provides traces as complete as possible and does intentionally not optimize the storage.

4.8.2 Related Work of the Query-based Debugging

According to the time at which queries are performed, we can divide existing works into pre-query, in-query, and post-query approaches. Pre-query approaches, such as Lencevicius's debugger [52], PQL [59], PQTL [33], and Squirrel [82], allow programmers to specify queries before runtime. Queries are typically compiled with the source code and become a part of the compiled code. In-query approaches, such as on-the-fly debugging [51] and Querypoints [62], support more dynamic queries. They allow programmers to query the program state when the execution is suspended. However, the querying scope is limited by the information provided at that suspension.

Post-query approaches including ALIA-TBD first record the complete trace and then perform queries over the recorded trace. WhyLine [48] is a graphical debugger on which programmers can ask "why did" and "why didn't" questions about the program state. The questions are generated automatically from static and dynamic analyses and presented as a list of choices on the user interface. WhyLine, on one hand, releases the burden of learning the query language. On the other hand, it limits the flexibility of questions, especially those that are not about "why". JHyde [38] combines declarative debugging, which is a variant of query-based debugging, with trace-based debugging. The data abstractions for the declarative debugging are at the level of method calls. Therefore, the declarative debugging cannot find the exact defect in a problematic method execution. JHyde uses the trace-based debugging, which records execution of statements, to compensate this limitation. JavaDD [32] is another typical declarative debugger, which uses a prolog-like query language. It not only allow programmers to pose

textual queries, but also provide context-aware queries through UI widgets like WhyLine. Even though we build a library to increase debugging efficiency, using context-aware queries seems more user-friendly in certain circumstances and this is part of the future work.

4.8.3 Related Work of the Trace Visualization

To visualize execution traces, one straightforward way is using a sequence-diagram. A sequence diagram shows object interactions arranged in time sequence. Such approaches include Jinsight [67], JIVE [20], JavaDD [32], and so on. However, if no proper graph reduction is applied, presenting the entire execution trace in one sequence diagram can increase the difficulty of navigation, because it requires to list all the created objects and the visualization of interactions between objects may span over several screens. Besides, it is impossible to visualize activities that have no interaction with other objects, such as writing a local variable.

Bohnet et al. [13] propose a technique that prunes less relevant information and detect repeated patterns from the trace. They also used a stage-like visualization in which an event can be expanded to a list of sub-events. The complete expanding path from the root to the current level is presented. ALIA-TBD reduces the graph at the view level instead of the data level. Its expansion focusses on the information at the current level.

4.9 Conclusion

In this chapter, we presented the design and implementation of a trace-based debugger for AD languages, called ALIA-TBD. The work is motivated by two orthogonal motivations. First, interactive debugging facilities developed in chapters 2 and 3 cannot allow programmers to freely inspect a past state in the execution. Second, since AD programs are typically transformed during the compilation, dedicated debugging tools should be aware of the transformation and present information at the abstraction level used in the source code.

We first showed two AD-specific debugging scenarios where defects are difficult to be found without using history. Based on the observations in the debugging scenarios, we proposed two key requirements for building trace-based debugging for AD languages: recording required information and providing information properly.

To fulfil the first requirement, we extended NOIRIn with two core components at the back-end of ALIA-TBD: a trace model and a storage model. The trace model defines which information needs to be collected at runtime. The storage model defines how the collected information is structured on the storage me-

dia. AD-specific concepts are modelled and run as first class objects in NOIRIn. Therefore, we can collect and store AD-specific information. This raises the abstraction level of debugging and thus increases the comprehensibility of using ALIA-TBD.

To fulfil the second requirement, we allow programmers to not only navigate but also query the recorded information. The information is visualized in a tree map that consists of nested rectangles. Each rectangle represents an event, which is called *join point* in our solution. Programmers can also search join points by using queries. The queries can be freely specified by programmers, and thus the means of navigation are not limited. Query results are highlighted in the tree map. To increase debugging efficiency, we built a library that contains frequently used queries. The front-end is entirely language-independent and it can be applied to any language as long as the trace is written in XML format.

We performed an evaluation on 4 types of joinpoints in 4 types of environment settings. The result shows that writing runtime information to the XML-format file takes the biggest portion of overhead. However, the example code used in the evaluation generate extremely dense join points. In practise, program executions have much less join points to be recorded. Moreover, the information written in the XML file is not optimized and thus excessively verbose. Therefore, we are optimistic about the future optimization. We used a debugging walkthrough to show that programmers need less effort in tackling debugging scenarios that require to use history. This increases the efficiency of debugging, and thus the quality of software built in AD languages can be enhanced.

5

Slicing Aspect-Oriented Programs

5.1 Introduction

In software development, program analysis is a prerequisite of many other activities, such as comprehending programs, optimizing structures, verifying properties, and finding defects. In the previous chapters, we have introduced debugging approaches, which can be used to analyse programs and then find defects. However, analysis performed in these approaches is mostly manual.

After observing a failure, programmers usually first read the program and then select a subset of the program that is relevant to the failure [34, 60, 90]. This can significantly reduce the scope of the program that needs further analysis like runtime inspection. Slicing is a technique that automatically selects relevant instructions according to the failure observed by programmers.

Slicing is performed on dependency graphs (DG). A considerable amount of research about operating on dependency graphs has been proposed, e.g., slicing [43, 50], testing [8], change impact analysis [78], program differencing and integration [41, 42]. The majority of them analyse programs written in mainstream languages, such as procedural languages and object-oriented languages. Few works explore other programming paradigms. In this chapter, we are particularly interested in constructing dependency graphs that are specific to aspect-oriented programming (AOP) and applying dedicated static slicing to aspect-oriented (AO) programs.

As introduced in Chapter 1, an AO language is usually developed as an extension of a base language. Existing research [85, 92] chose to extend the traditional DGs to accommodate AO-specific features. Handling constructs and working mechanisms that are absent in the base languages are the main challenges in ex-

tending the traditional DGs. We list the three most crucial AO-specific features that need to be considered.

Join point shadows Advices are applied at *join point shadows* (JPSs). Therefore, JPSs are inevitable elements included in tracing dependencies from advices to advised programs.

Program compositions Multiple advices can be executed at the same JPS in a specified order. A different order can change the execution result.

Non-argument context values Advices can access non-parameter context values, such as the callee object of a method call.

In 2002, Zhao [92] first proposed a preliminary approach, which hardly took the three features into account. A more recent and comprehensive study is from Xu and Rountev [85]. However, their solution implicitly transforms the source code and introduces elements that are not necessarily required.

In this chapter, we aim to propose a pure source-level and clean dependency graph, which can help programmers not only to slice AO programs but also to comprehend the dependencies in AO programs. We call the dependency graph in our solution *AODG*. We extend the traditional DG regarding to the three features and we develop a slicing algorithm on an AODG accordingly. A prototype of our approach is implemented as a standalone Java application in which AODGs and slicing results are visualized. We evaluated two aspects of this work: the efficiency of building a DG and the effectiveness of the slicing algorithm. The efficiency evaluation shows that the compilation overhead increases little when the size of the project is small. The effectiveness evaluation shows that our slicing algorithm can include all relevant nodes.

This chapter is structured as follows. Section 5.2 motivates this chapter by explaining the challenges of slicing AO program and the drawbacks of existing approaches. Section 5.3 presents newly introduced AO-specific features in the DG. Section 5.4 explicates the slicing algorithm and shows how it is conducted with two examples. Section 5.5 describes a preliminary user interface for performing slicing on AODGs. Section 5.6 depicts two evaluations of the efficiency of building AODGs and the precision of our slicing algorithm. Sections 5.7 and 5.8 present related work and conclude this chapter.

5.2 Challenges of Slicing AO Programs

Section 1.1 introduced background knowledge about slicing and showed how slicing is performed by using a code example. In this section, we add AO features to

the code example in AspectJ and the modified code is shown in Listing 5.1. The aspect PointProtocol (lines 20-33) contains a pointcut (lines 21-24) and two advices: **around** (lines 25-29) and **after** (lines 30-32). The pointcut consists of three parts: describing a join point shadow (line 22), accessing an argument value (line 23), and accessing the callee object (line 24). When setX() is called on line 16, both advices are applied. According to their lexical order, the **after** advice has a higher priority than the **around** advice. Therefore, the **after** advice is executed sequentially *after* the **around** advice. The output of the listing is "Set x with -2 and x becomes 2". If the precedence is reversed, the **after** advice accesses the context passed by the **proceed** computation in the **around** advice and the output becomes "Set x with 2 and x becomes 2".

```
1  public class Point {
2    private int x;
3    private int y;
4    public int getX() {
5      return x;
6    }
7    void setX(int _x) {
8      x = _x;
9    }
10   void setY(int _y) { y = _y; }
11   public String toString() {
12     return "(" + getX() + "," + getY() + ")";
13   }
14   public static void main(String[] args) {
15     Point p = new Point();
16     p.setX(−2);
17     p.setY(1);
18   }
19 }
20 aspect PointProtocol {
21   pointcut settingX(int i, Point p) :
22     call(void Point.setX(int))
23     && args(i)
24     && target(p);
25   Object around(int i, Point p) : settingX(i, p) {
26     if(i<0)
27       i *= −1;
28     return proceed(i, p);
29   }
30   after(int i, Point p) : settingX(i, p) {
31     println("Set x with " + i + " and x becomes " + p.getX());
32   }
33 }
```

Listing 5.1: A sample AspectJ program

AOP not only introduces new source constructs, such as contexts and advices, but also new concepts, such as join point shadows and advice compositions. To slice AO programs, we need to handle these AO-specific features properly. The following three features are crucial to the execution of AO programs but have no support in the traditional DGs at all.

1. JPSs are the place where advices are applied, like the call to Point.setX(**int**) on line 16. However, JPSs are inaccessible in the source code, because they are not designed as first-class constructs in base languages. Statements, which are modelled as the finest elements in the traditional DG, can contain multiple JPSs. For example, line 12 contains statement, but two JPSs: calling Point.getX() and Point.getY().

2. Multiple advices can be applied at the same join point. Their executions as well as the execution of the advised program form a *composition* at that join point. A different composition results in a different execution, and thus a slice involving that composition should be changed accordingly. An advice execution is normally transformed to a method call after compilation. However, it is impossible to describe a "call" to multiple elements from one place as well as the priorities between the called elements in traditional DGs

3. Advices can access, modify, and even replace context values at join points, like the **around** advice on line 25 in Listing 5.1. Take a method call for example, the context includes arguments, callee, caller and returned value. However, traditional DGs only provide nodes explicitly representing arguments.

Considering the above mentioned features, we developed a dependency graph for AO programs as well as a dedicated slicing algorithm.

5.3 Dependency Graph for Aspect-Oriented Programs

Considering the discussed problems, we propose the aspect-oriented dependency graph (AODG). Figure 5.1 shows the AODG for the program in Listing 5.1. Based on Figure 1.2, it adds the DGs of the **after** and the **around** advices. In the following paragraphs, we introduce the AO-specific extensions to the traditional DG with respect to the three features.

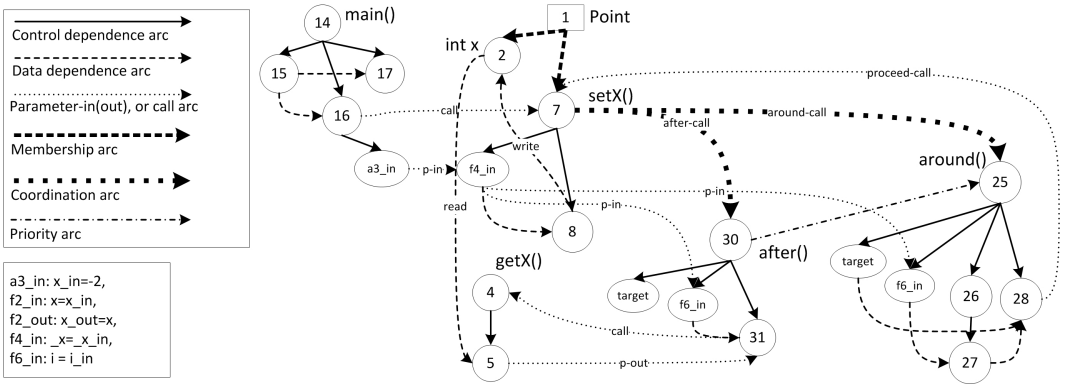


Figure 5.1: The AODG of program in Listing 5.1

Join point shadows Table 5.1 lists all the join point categories of AspectJ in the first column. Other main AOP languages such as JBoss AOP[1] and Spring AOP[2] support a subset of the listed join point categories. Each category requires a signature in its pointcut syntax and the signature specifies the source location of the JPSs. Pointcut "adviceexecution()" can be deemed as taking a signature selecting all advices as the parameter. We call a node representing the entrance of a member as a "declaration node", such as nodes 2, 4, and 30. We reuse declaration nodes as JPSs, and the third column in Table 5.1 shows which declaration node is used for each join point category.

This improvement leads to changing field accesses in traditional DGs. In the traditional way, field accesses are modelled by using auxiliary parameter nodes, like nodes *f2_in* and *f2_out* in Figure 1.2 (page 7). In AODG like Figure 5.1, arcs of field writes and reads respectively go to and from the field node through data dependency arcs, like arcs *2→5* and *8→2*. The arc is labelled with "read" or "write" according to its specific operation. This change has two benefits over the traditional DG. First, the graph is cleaner and more comprehensible. Second, we can locate all the places accessing a specific field.

Advice compositions To build the dependency between an advice and its advised programs, we use *coordination arcs*. A coordination arc connects declaration nodes of the advised element and the advice, like arcs *7→25* and *7→30*. For comprehension, each coordination arc is labelled with a schedule information and a pointcut designator, e.g., "after-call" on arc *7→30*. Schedule information can be *before*, *after*, or *around*. In AOP, multiple advices can be applied to one JPS and one advice can be applied to multiple JPSs. Our improvements preserve

[1]See `http://www.jboss.org/jbossaop`
[2]See `http://static.springsource.org/spring/docs/2.5.5/reference/aop.html`

Join point category	Pointcut syntax	Declaration node
Method execution	execution(*MethodSignature*)	method
Method call	call(*MethodSignature*)	method
Constructor execution	execution(*ConstructorSignature*)	constructor
Constructor call	call(*ConstructorSignature*)	constructor
Class initialization	staticinitialization(*TypeSignature*)	class
Field read access	get(*FieldSignature*)	field
Field write access	set(*FieldSignature*)	field
Exception handler execution	handler(*TypeSignature*)	class
Object initialization	initialization(*ConstructorSignature*)	constructor
Object pre-initialization	preinitialization(*ConstructorSignature*)	constructor
Advice execution	adviceexecution()	advice

Table 5.1: Join point categories in AspectJ and corresponding declaration node for each category in our solution.

the many to many relationship between advices and JPSs in source code.

An **around** advice can perform the advised program by calling **proceed()**. A **proceed** call can be deemed as a special method call. Therefore, we use a call arc with a label "proceed-call" to represent a **proceed()** call, like the arc *28→7*.

To express the priorities between advices, we introduce *precedence arc*, like arc *30→25* in Figure 5.1. The arc connects two advice declaration nodes and points to the advice with a lower priority.

Non-argument context values As discussed in Section 5.2, advices can access non-argument values, such as caller and callee objects of a method call, at join points. To support this, we propose *context nodes*, which are *target node*, *this node*, and *return node*, in AODG. For now, we only build data dependencies between context nodes with nodes using context values in advices, like arc *target→28*. Parameter-in-like arcs, which connect the origins of context nodes, are not provided for context nodes due to two difficulties. First, a designator refers to different contexts in different join point categories. Take a *this node* for example, it refers to caller in a method call but to callee in a method execution. Second, to determine which object performs a function needs runtime information, like dispatching overridden methods. However, context nodes are not necessarily used in slicing on AODG and this will be discussed in Section 5.4.

5.4 Slicing Algorithm

In Section 1.1, we have introduced a traditional slicing algorithm for inter-procedural programs. Briefly speaking, the algorithm consists of two traverses: ascending traverse and descending traverse. The first traverse visits procedures calling the

procedure where the investigated statement resides. The second traverse visits procedures called by the investigated statement. We can extend this algorithm to process the introduced AO-features in AODG, because Java—the base language of AspectJ—is an inter-procedural language.

AODG has several new types of arcs, which are field-read, field-write, and coordination arc. Field reads and writes can be deemed as method returns from *getters* and calls to *setters* respectively. Therefore, backtracking *field-read* arcs is descending and backtracking *field-write* arcs is ascending. A coordination arc starts from a declaration node, which can be deemed as the context where the corresponding advice is "called". Thus, we classify backtracking a coordination arc as ascending.

This subsection describes the slicing algorithm. For clarity, we divide the entire algorithm into three parts and refer each of them as "Algorithm #". Section 5.4.2 shows two slicing examples. Section 5.4.3 discusses important design choices that handle the crucial features introduced in Section 5.2.

5.4.1 Slicing Algorithm for AODG

Algorithm 1 describes the overall work flow of the slicing process. It takes two parameters: the dependency graph G which consists of a set of nodes and a node sn where the slicing starts. The algorithm traverses the graph in two phases. The first phase (line 7) traverses ascend and marks all the reached nodes. The listed arc types in the parameter list are excluded from this traverse. The second phase (line 9) traverses descend from the marked nodes (line 8) and marks all the newly reached nodes. Finally, the algorithm removes nodes representing un-executable source elements (line 10), such as parameters and member declarations, from all nodes marked during the two phases. The resulting list of the marked nodes is the slicing result.

Algorithm 1 The overall work flow

```
 1: procedure MarkVerticesOfSlice(G, sn)
 2: declare
 3:     G: an aspect-oriented dependence graph
 4:     sn: the starting node of the slicing process
 5:     S: a set of nodes in G
 6: begin procedure
 7:     MarkReachingNodes(G, {sn}, {parameter − out, field − read})
 8:     S := all marked nodes in G
 9:     MarkReachingNodes(G, S, {parameter−in, call, field−write, coordination})
10:     Remove unexecutable nodes from S
11: end procedure
```

Algorithm 2 shows how a single traversing phase is performed. It continuously selects a node v from a work list (line 11) and analyses connected nodes of v until the work list becomes empty (line 10). The initial value V of the work list is specified in Algorithm 1. It is the starting node sn in the first phase and the marked nodes S in the second phase. For each arc that ends at v, the algorithm marks the source node w of the arc if the kind of arc is not excluded and w is unmarked (line 20). However, if the arc is a *proceed* arc, the algorithm marks w conditionally (line 17).

Algorithm 2 A single traverse

```
 1: procedure MarkReachingNodes(G, V, Kinds)
 2: declare
 3:     G: an aspect-oriented dependency graph
 4:     V: a set of nodes in G
 5:     Kinds: a set of kinds of edges
 6:     v, w: nodes in G
 7:     WorkList: a set of nodes in G
 8: begin procedure
 9:     WorkList := V
10:     while WorkList ≠ ∅ do
11:         Select and remove node v from WorkList
12:         Mark v
13:         for each unmarked node w such that an arc w → v whose kinds is not in
    Kinds do
14:             if w → v is a proceed arc then
15:                 HandleProceedArc(G, sn, v, w)
16:                 if w is marked then
17:                     Insert w into WorkList
18:                 end if
19:             else
20:                 Insert w into WorkList
21:             end if
22:         end for
23:     end while
24: end procedure
```

To influence the execution of a composition, the only option is to add an *around* advice with a priority that is higher than any advices already in that composition. As a result, the execution of the composition is performed in the control flow of the added *around* advice. Based on this observation, Algorithm 3 sketches how a *proceed* arc is handled. A *proceed* arc starts at a statement node (variable f) in an **around** advice and ends at the declaration node (variable t)

of a member, which is a field, a method, a constructor, or another advice. The algorithm first finds all marked advices applied to the JPS associated with the member (line 10) and then chooses the one (variable h) with the highest priority (line 12). If f, which is the node calling *proceed*, is from an advice with higher priority than h, the algorithm marks h.

Algorithm 3 Handling a proceed arc

```
 1: procedure HandleProceedArc(G, f, t)
 2: declare
 3:     G: an aspect-oriented dependency graph
 4:     f: the starting node of a proceed arc
 5:     t: the ending node of a proceed arc
 6:     h: the declaration node of an advice
 7:     C: a set of nodes in G
 8: begin procedure
 9:     for each marked node w such that an arc t → w whose kinds is coordination
    do
10:         add w to C
11:     end for
12:     h := the declaration node of the advice with the highest priority in C
13:     if f is in an advice with a higher priority than h then
14:         Mark f
15:     end if
16: end procedure
```

5.4.2 Two Slicing Examples

We use the example described in Section 5.2 to illustrate how the algorithm works. To recap, the **around** advice has a lower priority than the **after** advice. Therefore, the **after** advice can access the original context of calling Point.setX(). However, the context received by the called method may be modified by the **around** advice. The following two paragraphs show the slicing process that starts from the **after** advice and the method Point.setX() respectively.

Slicing from after advice

Suppose the execution of statement on line 31 in Listing 5.1 is problematic, slicing starts from node 31. In the first phase, nodes 31, 30, 7, 16, 15, 14, $f6\_in$, $f4\_in$, and $a3\_in$ are marked. When the traversal reaches node 7, it finds node 28 through a *proceed* arc. To determine whether node 28 should be marked, the Algorithm 3 is performed. All marked declaration nodes of advices that are applied to the

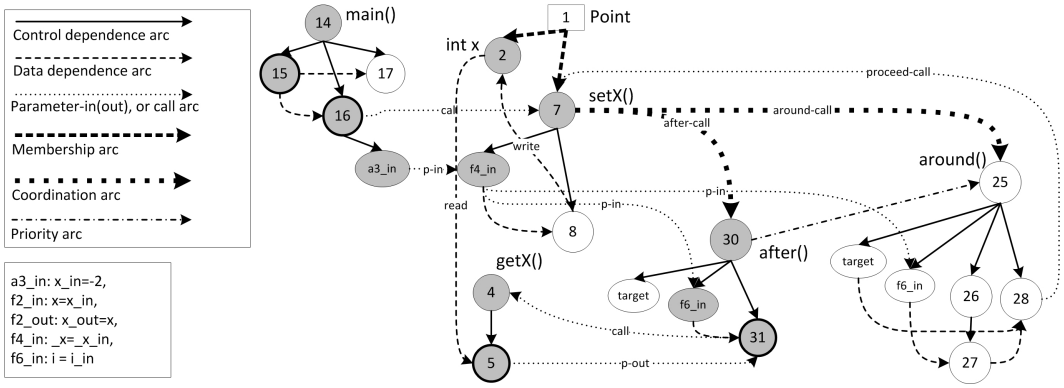


Figure 5.2: An AODG and a slice starting from node 31. Bold circles represent the slicing result.

JPS represented by node 7 are found. The result is a list only containing node 30, which represents the **after** advice. Node 28, which calls **proceed**(), is from an advice with lower precedence than the **after** advice. Therefore, node 28 should not be included in the slice. In the second phase, the algorithm can traverse arcs that were excluded in the first phase. Therefore, node 5, which connects to node 31 with a parameter-out arc, is marked. Then, 4 and 2 are marked. Figure 5.2 shows the AODG with marked nodes during the slicing. The bold circles represent the final slicing result after removing nodes for un-executable source elements.

Slicing from Point.setX()

Suppose the execution of statement on line 8 in Listing 5.1 is problematic, slicing starts from node 8. In the first phase, nodes 8, 7, 16, 15, 14, $f4\_in$, and $a3\_in$ are marked. Again, the algorithm needs to inspect node 28 when the traversal reaches node 7. Different from the previous scenario, the algorithm does not find any marked advice. This indicates that the **around** advice, where node 28 is from, has the highest precedence among all the advices connecting node 7 with coordination arcs. Therefore, node 28 is included in the slice and slicing proceeds to mark nodes 27, 26, 25, $target$ (of **around** advice), and $f6\_in$. The second phase does not mark any new nodes. Figure 5.3 shows the AODG with marked nodes during the slicing. The bold circles represent the final slicing result after removing nodes for un-executable source elements.

The **after** advice, which has a higher priority than the **around** advice, is not executed in the control-flow of the **around** advice. However, method setX() is "proceeded" in the control-flow of the **around** advice. Thus, the **around** advice influences the execution of the setX() but does not for the **after** advice. The first

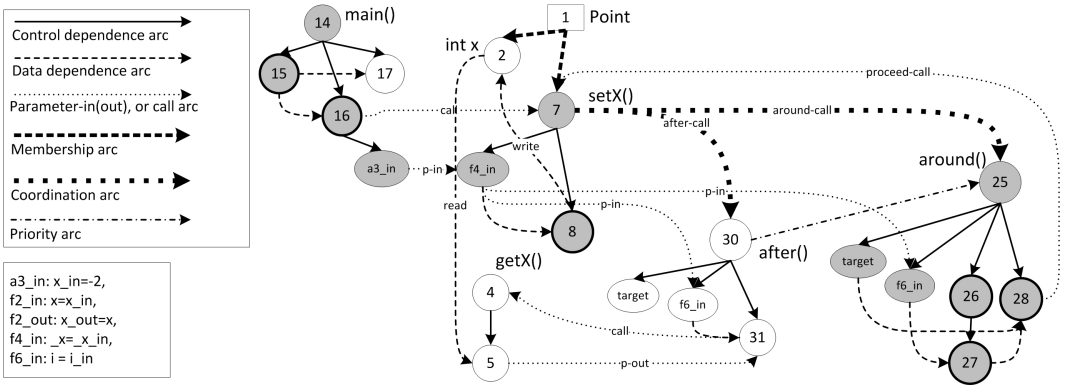


Figure 5.3: An AODG and a slice starting from node 8. Bold circles represent the slicing result.

slicing result does not contain any node from the **around** advice but the second does. The results exactly fit the semantics of AspectJ.

5.4.3 Design Choices

Some AO-specific features introduced in Section 5.3 are for increasing comprehensibility of the AODG and they not used in slicing. The following subsections explain why they are not necessary in the slicing algorithm.

Uses of Schedule Information

The algorithm does not use the schedule information of advices explicitly. It only uses the *proceed()* arc to classify advices into ones with a *proceed()* call and ones without.

If the context values are primitive values or unmodifiable objects, modifications to the context values in the advices without calling *proceed()* cannot influence the original context values. Therefore, when a slicing starts outside of such an advice, the resulting slice is supposed to not include statements of the advice.

If the context values are modifiable objects, advices can change the state of the objects by calling methods or writing fields of those context objects. Therefore, a slicing on the states of the context objects in the advices can be substituted by a slicing that starts from a node belongs to the classes of those objects. Listing 5.2 shows an example. Both the **before** and the **after** advice call methods that modify the state of the same Point instance. If such modifications on a Point instance in the advice composition is unexpected, programmers can perform slicing from class Point and find where the operations that modify the state of the Point instance are unexpectedly called.

```
1 aspect PointProtocol2 {
2   pointcut settingX(Point p) :
3     call(void Point.setX(int)) && target(p);
4   before(Point p) : settingX(p) { p.moveTo(1, 1); }
5   after(Point p) : settingX(p) { p.reset(); }
6 }
```

Listing 5.2: A sample AspectJ program

If an *around* advice does not call *proceed*, it skips the original computation at its advised JPSs. In AODG, this is modelled as an *around* advice which does not have a *proceed* arc connected to the advised JPSs. Therefore, when a slicing reaches at the advised JPSs, it cannot trace back to the *around* advice, because of the missing *proceed* arc. Slicing at the JPS indicates that the JPS is actually executed. Thus, the *around* advice is failed to be applied at runtime and this exactly fits the AspectJ semantics.

Uses of Non-Argument Context Values

AspectJ uses the designators *this*, *target*, and *return* to refer to non-argument context. These designators refer to different objects according to the join point category. Table 5.2 lists all join point categories as well as their **this**, **target**, and **return** contexts. Items "caller" and "catcher" refer to the objects invoking the corresponding computation specified by the join point. Item "callee" refers to the object eventually performing the computation. Some contexts do not exist and they are represented by a dash "-" in the table.

It is not necessary to explicitly build data dependencies from nodes representing creations of context values to context nodes. The creations can be always included in slices on AODG when a slicing starts from context nodes. Figure 5.4 abstracts away irrelevant information and shows a typical path of calling an advised method in AODG. When the statement at node 5 is executed, it calls method b() (node 6) and the call is advised by advice c (node 8). The callee of b() is created (node 4) in the control flow of method a() (node 3). The caller of b(), which is also the callee of a(), is created at node 1. The object returned by b() is created in the control flow of b() (node 7).

Suppose the slicing starts from node 9, which represents a context node, it can definitely reach the declaration node of the advice, which is represented by node 8. Then the slicing algorithm can backwards traverse and mark all nodes in Figure 5.4 as long as they exist in AODG. Context values are included in the slice no matter if they have explicitly data dependencies to context nodes or not. This also holds for other join point categories listed in Table 5.2.

Join point category (AspectJ)	this	target	return
Method execution	callee	callee	-
Method call	caller	callee	returned object
Constructor execution	callee	callee	-
Constructor call	caller	-	-
Class initialization	-	-	-
Field read access	caller	callee	field
Field write access	caller	callee	-
Exception handler execution	catcher	-	-
Object initialization	callee	callee	-
Object pre-initialization	-	-	-
Advice execution	callee of advice	callee of advice	-

Table 5.2: Objects referred by **this**, **target**, and **return** designators in different join point categories.

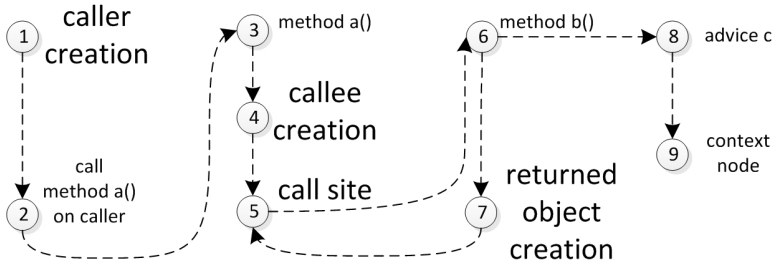


Figure 5.4: A typical flow of calling an advised method

5.5 User Interface

To increase the efficiency of using AODG, we implemented a preliminary graphical user interface (GUI) and it is shown in Figure 5.5. It consists of two parts: a panel on the left and a canvas on the right. The panel is for navigation and showing slicing results. The canvas renders the AODG according to the configuration in the panel.

Six group widgets are on the panel from top to bottom. The "Project" group is for specifying the path of the analysed project. The "Classes" group is for filtering nodes shown on the canvas. It lists all compiled classes and only nodes from checked classes are rendered on the canvas. The "Dependency" group is for filtering arcs. It lists four types and only arcs with the checked types are

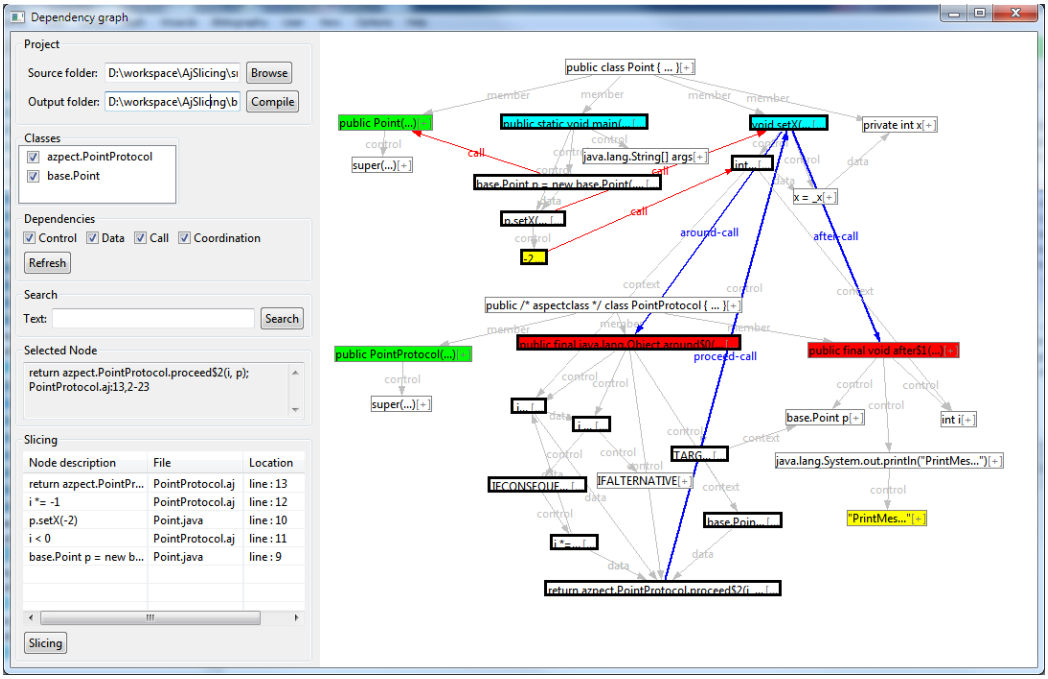


Figure 5.5: A snapshot of our tool for slicing AO program on AODGs

shown. The "Search" group provides functionality of searching nodes that are already shown on the canvas. By pressing the "Search" button, nodes containing specified text are highlighted. The "Selected Node" group presents descriptions of the selected node. Once a node is selected, it becomes the slicing criterion. The "Slicing" group provides a table for listing all nodes for executable statements in a slice. The visualization of a slice in AODGs is described in the following paragraph.

In theoretical analysis, we use line numbers and different shapes as notations to keep AODGs clean because of limited space. The AODGs shown on the canvas use different notations. Each node is a rectangle labelled with a description, which is mostly the source code fragment that node represents. Nodes are distinguished by using colors, e.g., declaration nodes of advices are in red. Each arc is a labelled straight arrow instead of being in a particular line style. The nodes with bold edges represent marked nodes, which includes executable and un-executable source elements, in slicing.

5.6 Evaluation

We evaluate two aspects of this work: the efficiency of building an AODG and the effectiveness of the slicing algorithm. The two evaluations are introduced in the following two subsections.

5.6.1 Performance

The first evaluation is to measure the time overhead of building AODG during compilation. There is no overlap between the original part of the compiler and the augmented part for building an AODG. Therefore, the time of the original compilation and building an AODG can be measured independently in one run. We compiled four projects, which are *Mediator*, *Tracing*, *Telecom*, and *Spacewar*. Project *Mediator* is one of design patterns written in AspectJ and the other three are from the examples in the AspectJ Development Tools (AJDT)[1]. Some projects are refactored, because our implementation currently does not support inner classes. Our evaluations were run on an HP laptop with Windows 7. The laptop has a CPU of Intel Core i5 M430 2.27Hz and 4GB RAM. The development platform we worked on is Eclipse Helios. The elapse time of running each project sometimes has some fluctuations mainly due to JVM working mechanisms such as garbage collection. To increase the integrity of our evaluation results, we run each project 8 times, excluded the noise that excessively deviates from other records, and then averaged the relatively close 5 remaining records.

Table 5.3 lists all the results. To measure the size of a compiled project, we only count the lines that are used in constructing AODGs. Therefore, lines with package imports and comments are excluded. To give an intuition how complex an AODG is, we measured the amount of created nodes and arcs (columns 3 and 4). In an AODG, nodes represent source code fragments and arcs represent relationships between these code fragments. Therefore, the amount of nodes grows linearly ($O(n)$) along with the increasing size of the project and the amount of dependencies grows quadratically ($O(n^2)$). However, the amount of created entities does not necessarily indicate the time of building that graph (column 5). Besides creating entities, searching entities for building arcs also contributes a significant portion in the graph construction. For example, when a variable is found in a method or a constructor, the graph construction first search the table with local variables. If nothing is found, it then searches the table with fields in that class. Even though *Spacewar* is 10 times as large as *Tracing*, such variable searches appear in the former only 5 times as many as in the latter. Besides variable searches, there are call site searches, join point searches, etc.

We cannot draw a precise conclusion about the overhead for large projects due to the limited amount of evaluated projects. However, the data can show that the compilation overhead for small projects is acceptable. A threat to the validity of our result is that we built the tool based on AspectBench compiler (*abc*)[2], which is known to be not the most efficient compiler. Therefore, using *abc* can result in long normal compile time and little time overhead ratio.

[1] See `http://www.eclipse.org/ajdt/`

[2] See `http://www.sable.mcgill.ca/abc/`.

Project	#LOC	Normal compile time(ms)	# of nodes	# of arcs	Graph building time(ms)	Time overhead ratio(%)
Tracing	106	1792	319	675	19	1.06
Mediator	119	2730	248	573	23	0.84
Telecom	226	2124	600	1276	46	2.17
Spacewar	1044	4091	2542	6618	102	2.49

Table 5.3: Result of performance evaluation about building AODGs.

5.6.2 Effectiveness

To evaluate how effective our slicing algorithm works, we investigated code in three cases with representative AO-specific features. In each case, we intendedly inject one statement that will eventually cause unexpected behavior or value. The injected code can be seen as the *defects*.

Case 1 : Intertype declaration The purpose of choosing this case is to evaluate how well our approach support the intertype declaration. The slicing is performed on the project *Telecom* and Listing 5.3 shows the most relevant part of the source code. In Listing 5.3, aspect Billing introduces a new field totalCharge to class Customer (line 2). It reads the field in its own method getTotalCharge() (line 4) and updates the field in a intertype method Customer.addCharge() (line 6). The injected code causes that getTotalCharge() returns an unexpected value. Therefore, the algorithmic slicing should start on line 4.

```
1 public aspect Billing {
2   public long Customer.totalCharge = 0;
3   public long getTotalCharge(Customer cust) {
4     return cust.totalCharge; /* slicing starts here */  }
5   public void Customer.addCharge(long charge){
6     totalCharge += charge;
7   }
8 }
```

Listing 5.3: The core program in slicing case 1

Case 2: Multiple advices This case traces from a single entity to multiple advices. It uses the project *Spacewar* and Listing 5.4 shows the relevant code. Listing 5.4 contains a class Display (line 1) and an aspect DisplayAspect (line 16). Class Display declares a method noticeSizeChange() that calls another method

initializeOffImage(), which modifies fields offImage and offGraphics (lines 11 and 12). DisplayAspect declares two **after** advices and both of them call noticeSizeChange(). The injected code causes that initializeOffImage() is unexpectedly executed. Therefore, we need to find how the method is called and the algorithmic slicing should start from initializeOffImage().

```
1  class Display {
2    Image offImage;
3    Graphics offGraphics;
4    void noticeSizeChange() {
5      initializeOffImage();
6    }
7    private void initializeOffImage () { //slicing starts here
8      int w = getSize().width;
9      int h = getSize().height;
10     if ( w > 0 && h > 0) {
11       offImage = createImage(w, h);
12       offGraphics = offImage.getGraphics();
13     }
14   }
15 }
16 aspect DisplayAspect {
17   after() returning (Display display): call(Display+.new(..)) {
18     display.noticeSizeChange();
19   }
20   after(Display display) returning (): call(void setSize(..)) && target(display) {
21     display.noticeSizeChange();
22   }
23 }
```

Listing 5.4: The core program in slicing case 2

Case 3: Multiple dispatch sites Following the previous case, this case searches from a single advice to multiple advised entities. The evaluation is performed on the project *Spacewar* and Listing 5.5 shows the relevant source code. To increase generality of this case, we add **this**() to the pointcut expression (line 7) and use the *this*-context value in the advice (line 8). The injected code causes a problematic *this*-context value. Therefore, we need to perform the algorithmic slicing on the statement on line 8.

```
1  aspect RegistrySynchronization {
2    protected pointcut synchronizationPoint():
3      call(void Registry.register(..)) ||
4      call(void Registry.unregister(..)) ||
5      call(SpaceObject[] Registry.getObjects(..)) ||
6      call(Ship[] Registry.getShips(..));
7    before(Registry r): synchronizationPoint() && this(r) {
8      r.getClass().getName();  //slicing starts here
9    }
10 }
```

Listing 5.5: The core program in slicing case 3

Experiment & Result We use *precision* and *recall* to measure the effectiveness of our approach. Precision is defined by the fraction of the selected node that is actually relevant. High precision means that our approach selected substantially more relevant nodes than irrelevant. Recall is defined by the fraction of the relevant nodes that is actually selected. High recall means that our approach returned most of the relevant nodes.

$$precision = \frac{r}{n} \qquad recall = \frac{r}{R}$$

r : number of relevant nodes selected
n : number of nodes selected
R : total number of relevant nodes

In the first phase, we performed a forward runtime analysis, which starts from the execution of the injected code and ends when the failure is observed. If statement B is influenced by another statement A, either B uses a value created or modified at A, or execution of B depends on execution of A. We used comments to mark statements that are influenced by the injected code or statements that are already marked. When the failure was marked, we removed marks from the marked statements that cannot influence the failure. In the end, the marked statements S we obtained is the minimum set of statements involved in the infection chain from the defect to the failure.

In the second phase, we used our tool to accomplish the same slicing tasks performed in the first phase. In each case, we collected the numbers of selected nodes by the tool and they represent n. Then, we found all nodes corresponding to S in AODG, collected the numbers of the marked nodes and they represent R. Last, we count the number of nodes that were selected by in both phases, and it represents r in our measurement.

Case	R	n	r	*Precision*	*Recall*
1	12	31	12	0.39	1
2	12	83	12	0.14	1
3	32	167	32	0.19	1

Table 5.4: Result of effectiveness evaluation about our slicing algorithm

Table 5.4 lists our evaluation results. All recalls are 1 but precisions are relatively low. The full recall means that the algorithm can select all the nodes selected by us. The low precision means that the algorithm selected many more statements than we did. This is mainly due to the following two reasons.

- The algorithm selects statements without a specific goal and it only stops when AODGs have no more reachable nodes. The analysis we performed in the first phase only limit in the scope from the defect to the failure.
- We selectively used dependencies, but the algorithm uses all types of dependencies. Actually, programmers rarely analyse code in a systematic and thorough way as the algorithm does. Take Case 2 for example, it requires to select statements that are responsible for an unexpected method execution. We traced the call hierarchy and marked all the traced statements.

5.7 Related Work

Zhao's work [92] is the first attempt of slicing AO programs. It is a conceptual proposal that is not implemented and evaluated. It augments the traditional DG with only a handful of AO-specific elements, such as coordination arcs that represent advising in AOP. Therefore, very limited AO-specific features are supported. However, all following works, including ours, employ the graph building process introduced in Zhao's work—first constructing DGs for OO and AO code separately and then weaving individual DGs according to the weaving information.

Sereni and Moor's work [75] focuses on increasing the efficiency of compiling AO programs. They use a lightweight AO language, called AJD, to analyse whether the weaving of an advice with dynamic pointcuts can be statically determined at compile time. AJD only supports join points of procedure calls, procedure executions, and advice executions. Their analysis builds a call graph at each procedure call. All advices modifying a call are added to the graph and chained in the order that they are specified in the source code. Precomputations are performed for some pointcuts and infeasible edges can be pruned in the graph according to the computation result. In this way, computations at compile time eliminate unnecessary evaluations of pointcuts at runtime. Our work focuses on

program comprehension instead of optimization. Therefore, we try to keep the consistency between source code and graphs and avoid any transformation of code.

Parizi and Ghani implemented AJcFgraph [66], which is a tool for visualizing control-flow dependencies of AspectJ programs. AJcFgraph can only build control-flow dependencies. As we discussed in Section 5.2, other dependencies, especially data-flow dependencies, are also important in comprehending and analysing AspectJ programs. Besides, AJcFGraph builds a composition at a JPS by using the complete graphs of advices. As a result, an advice that is applicable to multiple JPS has duplicated DGs at each JPS.

Ishio et al. [44] extended the traditional call graph by regarding an advice execution as a kind of method call. A call graph visualizes control dependence relations between objects and aspects and supports the detection of an infinite loop. They performed a user study and the result shows that using DG can effectively help programmers to understand AO programs.

Xu and Rounteve [85] proposed a framework for analysing AspectJ programs at source level by using a dedicated DG. Their approach is the first one that thoroughly considered and solved all three challenges discussed in Section 5.2. With respect to the three challenges, their approach extends the traditional DG in the following ways:

1. It introduces auxiliary nodes as the place holders of JPSs. Advices applied to a JPS interact with its place holder.

2. It models compositions as *interaction graphs* (IGs), and IGs are separated from DGs. In IG, each advice has auxiliary nodes for its entry, exit, returning point of the nesting computation, and result of the pointcut evaluation. The exact execution order of the composition is also computed in the IG.

3. It considers the non-argument context values as parameters of advices. For an *around* advice, it takes all context values of all join points as parameters, no matter whether they are actually used in the execution of the around advice.

Xu and Rountev's approach introduces several kinds of auxiliary nodes to build DGs for AO programs. It transforms the source code, e.g., putting computed compositions at JPS place holders and filling *around* advices with undeclared parameters. Our approach is closer to the source code than theirs, because our DG has almost no auxiliary constructs and does not require to transform source code.

Table 5.5 summarizes the mentioned related work. Symbol "N" means the corresponding feature is not supported. Some items are appended with plus '+'

Work	JPS	Advice Composition	Context values
Ours	declaration node	Precedence dependency & labelled coordination arc	auxiliary nodes [+]
Zhao's	statement node	N	N
Ishio et al.'s	statement node	N	auxiliary nodes [+]
Sereni and Moor's	JPS node [+]	placeholder [+]	N
Parizi and Ghani's	JPS node [+]	duplication [++]	N
Xu and Rounteve's	JPS node [+]	placeholder [+]	auxiliary nodes [+]

Table 5.5: A comparison of our work and the related work.

notations, which means the corresponding technique adds nodes to the DGs that are built completely in the traditional way. The item "duplication" is appended with a double-plus, which means duplication introduces a considerable amount of nodes. From the table, we know that only our approach and Xu and Rountev's approach support all three features. However, our approach introduces less auxiliary nodes.

5.8 Conclusion

Even though slicing AO programs has been studied by several work, we are the first to explicitly identify and discuss the three crucial AO-specific features in slicing. The three features are join point shadows (JPSs), advice compositions, and non-argument context values. They can either influence control flow or change program states in execution. Without handling them properly, slicing AO programs cannot be correct.

Considering the three features, we proposed aspect-oriented dependency graph (AODG) which is an extension of the traditional dependency graph (DG). We extended the declaration nodes of members to be JPSs and added the traditional DG with coordination arcs, precedence arcs, and proceed-call arcs. For comprehensibility, we labelled the added arcs with information reflecting their attributes. Accordingly, we developed a slicing algorithm on AODG. We have shown that some extended features are not necessarily used in the algorithm. However, keeping them in AODG increases the consistency between the AODG with the source code.

We implemented a graphical user interface with functionalities for navigating and slicing AODGs. We evaluated the efficiency of building an AODG and

the effectiveness of the slicing algorithm. The efficiency evaluation shows that the compilation increases little overhead for small projects. The effectiveness evaluation shows that our slicing algorithm can include all relevant nodes.

We have not investigated the problems of slicing other advanced-dispatching paradigms, especially predicate dispatching. However, AOP introduces more complex concepts, such as precedence and advice composition, than predicate dispatching. We believe that our approach of designing and implementing AODG can be easily generalized and applied to other AD paradigms.

6

Conclusion and Future Work

6.1 A Generic Debugging Process Model

We are facing the next generation of the "debugging scandal" after Henry Lieberman proclaimed the first generation in 1997. AD languages are implemented as an extension of a main-stream language, which is also called the base language. After the compilation, AD programs are transformed to the compiled form of the base language so that it can run in the execution environment for the base language. Debugging facilities for the base language are also applicable for the AD languages. But when using these, programmers have to inspect, comprehend, and analyse abstractions which are absent in the source code during debugging.

At the beginning of the thesis (Section 1.1), we described a debugging process model (DPM) that includes fundamental elements required in debugging. Traditional debuggers cannot *reflect* AD-specific information correctly. This thesis introduced four works to rebuild the *reflect*-relation and we put them in three approaches, which are related to interactive debugging, trace-based debugging, and static slicing respectively. In the following subsections, we discuss how each of these approaches instantiates the DPM and illustrate this in a figure. The figures label some components with numbers of the corresponding chapters or sections. "S" represents a section and "C" is for a chapter. Subsection 6.1.4 discusses how the introduced approaches can be combined to give a thorough view of the AD-specific debugging.

6.1.1 Interactive Debugger with Advanced Breakpoints

To find which AD-specific information is required during debugging, we investigated four dedicated fault models and identified eleven tasks that an ideal AD-specific debugger should support. We showed that existing debuggers are not powerful enough to support all identified tasks, because the AD abstractions are

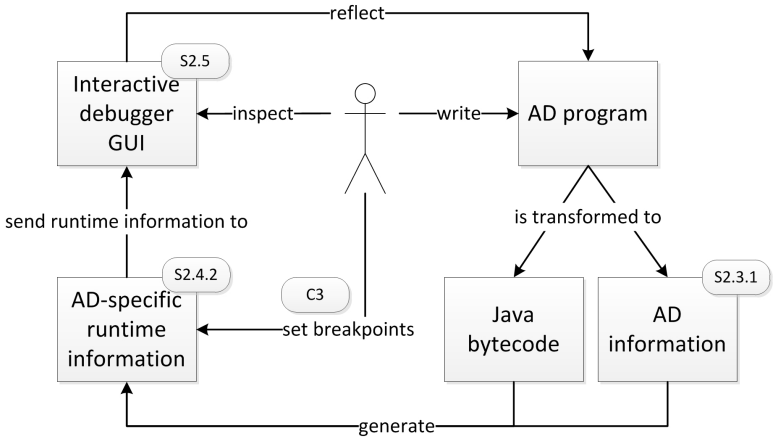


Figure 6.1: A debugging process model of the interactive debugging

lost after compilation. Chapter 2 described the design and implementation of an AD-specific debugger. We performed the following three critical improvements on the traditional way of debugging AD programs with interactive debuggers. The instantiated DPM is shown in Figure 6.1.

1. We modified a compiler to prevent weaving at compilation and put all required AD-specific information as well as weaving information in a file after compilation.

2. We extended NOIRIn, which is an execution environment that models and runs AD concepts as first-class objects, to import the file with AD-specific source information and collect AD-specific debugging information at runtime.

3. We implemented a dedicated graphical user interface (GUI) to render AD-specific information and locate source code corresponding to a rendered entity.

To select information in interactive debugging, an essential step is to set breakpoints specifying where a program should be suspended at runtime. A debugging session may use multiple logically related breakpoints so that the sequence of their (de)activations leads to the expected suspension with the least irrelevant suspensions. However, existing breakpoints, which are mainly based on line locations, are not expressive enough to describe the logic relations between breakpoints. Programmers have to manually perform some repeated tasks, thus debugging efficiency is decreased.

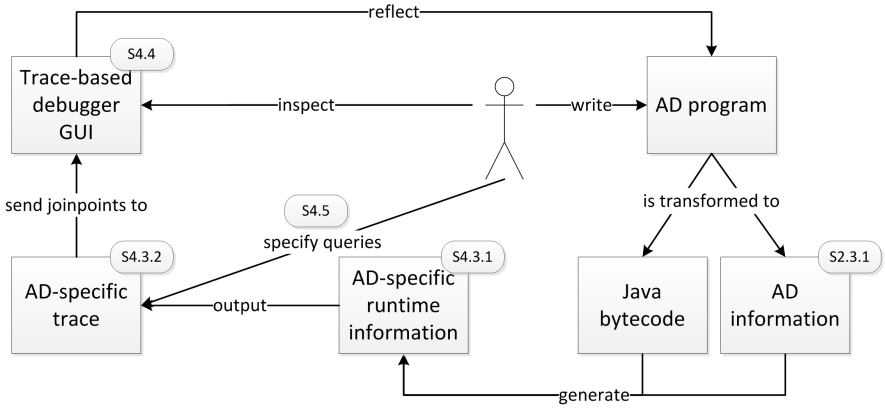


Figure 6.2: A debugging process model of the trace-based debugging

In Chapter 3, we identified five frequently encountered debugging scenarios that require to use multiple breakpoints. Inspired by them, we designed and implemented a breakpoint language that uses pointcuts to select suspension times in the program. The language allows programmers to use comprehensible source-level abstractions to define breakpoints. Also, multiple breakpoints can be freely composed to express their collaboration. In this way, an expected suspension can be expressively programmed and reached with less or even no irrelevant suspensions. We performed a code analysis over 13 representative projects and the results show that the language can benefit programmers in a significant portion of cases.

6.1.2 Trace-Based Debugger

Execution of a defect may cause a failure to occur in the following execution. Debugging is a backward process that traces from the observed failure to the defect. However, interactive debugging facilities do not provide the full execution history. Besides, existing trace-based debuggers are AD-agnostic. Therefore, we proposed a trace-based debugger for AD languages in Chapter 4 and the instantiated DPM is shown in Figure 6.2.

The runtime information is collected and stored, and debugging is performed on the recorded information, which is also called *trace*. To generate such a trace, we designed a trace model and a storage model. The trace model defines which information needs to be collected at runtime. The storage model defines the format of the collected information on the storage media. To inspect interesting information, programmers can not only navigate the trace in a step-wise manner but also search information by performing queries.

The DPM of the interactive debugging and the trace-based debugging have

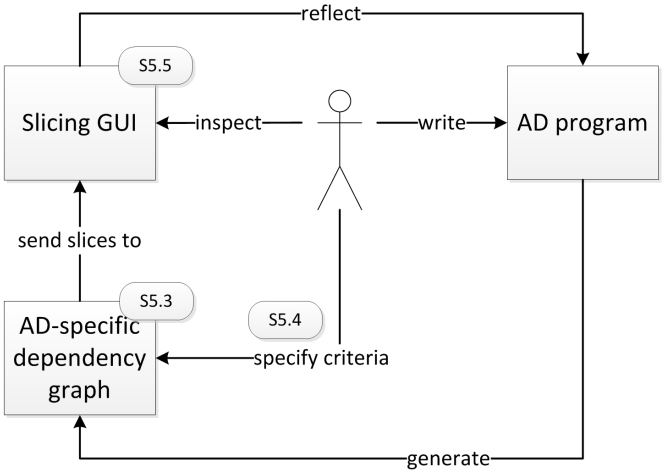


Figure 6.3: A debugging process model of the slicing

the following important commonalities:

- The identified eleven debugging tasks for the interactive debugging are also supported by the trace-based debugging.
- To rebuild the *reflect*-relation, two essential steps in both approaches are preserving AD-specific information after compilation and restoring it at runtime.
- In both approaches, programmers need to interact with the runtime information either directly or indirectly. In addition to runtime information, the trace also includes history-specific information, which links information generated at different moments during runtime.
- Breakpoints can be generalized to queries. For example, a line breakpoint can be formulated as a query that searches information generated from the source code at a specific line.

6.1.3 Slicing

In addition to the previous two debugging approaches that require manual inspections, we developed an AO-specific slicing algorithm in Chapter 5. Slicing can automatically select source code that influences the execution of statements, advice application or program states that lead to a failure, and the defect is included the selected code. The instantiated DPM is shown in Figure 6.3.

Slicing is performed on dependency graphs (DG), which contains dependencies between source elements. Slicing uses the dependencies to trace relevant

source element with respect to the starting element, which represents where a failure was found. Therefore, dependencies in DG can be deemed as a kind of debugging information. We discussed how three AO-specific constructs, which are join point shadows, program compositions, and non-argument context values, can influence the execution of AO programs. Considering the three constructs, we developed AODG, which is an AO-specific DG, as well as a slicing algorithm that performs on AODG. We conducted evaluations on the efficiency of building AODGs and the effectiveness of the slicing algorithm. The efficiency evaluation shows that the overhead is acceptable when compiling small projects. The effectiveness evaluation shows that our slicing algorithm can include all relevant nodes.

6.1.4 A Combined Approach

Each approach can on its own assist programmers to find the defect, and each one has strengths and weaknesses. When putting all three approaches together, they give a more powerful and thorough way of debugging AD programs. Figure 6.4 shows a combined DPM of the three approaches.

Slicing, interactive debugging, and trace-based debugging are performed before, during, and after the program execution respectively. Slicing gives programs a list of source locations that may cause a failure. The suggested locations significantly narrow down the inspection scope for further analysis. They can be used in setting breakpoints or formulating queries in interactive debugging or trace-based debugging, respectively. Interactive debugging provides detailed information at a current suspension. To explore the past execution or to have an overview over the entire execution, trace-based debugging can be applied.

All three approaches contain dedicated debugging information, corresponding GUIs, and ways of searching debugging information. The first feature is the basis of the other two. Therefore, we draw the conclusion: *a key to defuse the "debugging scandal" is to provide debugging information that is specific to the concepts used in the source code.*

6.2 Future Work

Developing fault models of other AD paradigms. The interactive debugger is developed based on AO-specific fault models, but it is already applicable to other AD paradigms supported by ALIA4J. We lack a systematic research on the fault models of other AD paradigms, which includes faults related to concepts and features different from those in AOP. This leads to functional differences at

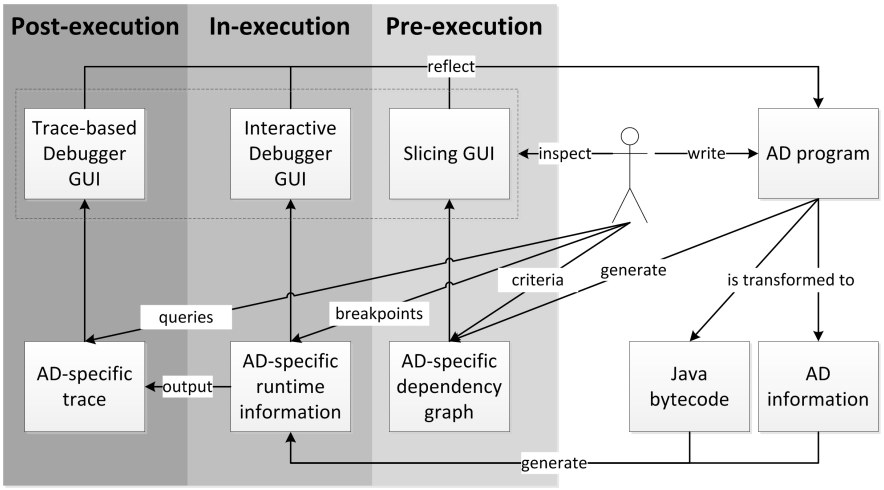


Figure 6.4: A debugging process model that combines the three approaches

the front end of the debugger. Take JPred [1], which is a predicate dispatching language, for example, it uses specialized parameters to override methods and only one method is eventually executed at a method call. Therefore, the program compositions at dispatch sites have a fixed form that consists of only one method.

Improving the breakpoint language. The breakpoint language is motivated by limited and ad hoc scenarios based on our past experiences and observations. Some supported expressions can be refined. For example, **composition**() expressions can only describe existences and exclusions of advices in a composition. Except these two types of relationships, advices can be executed in a specific order, such as sequential order at the same level, sequential order at different levels, nesting order, etc. Another example, **checkNPE**() expressions are used only for detecting *null* pointers at dereference operations. The concept can be generalized and used for other issues associated with a consecutive lists of operations. We need to investigate more scenarios and enhance generality of the language. The language should ideally provide only the atomic units and allow programmers to customize any breakpoint specification by composing the atomic units.

Providing context-aware queries. To increase debugging efficiency, we provide a library that contains some frequently used queries in the trace-based debugger. However, learning and writing a query still need effort. Some queries are always performed in a particular context. For example, when a programmer inspects a variable value, she is more likely to search how a variable value was up-

[1]See `http://www.cs.ucla.edu/~todd/research/jpred.html`

dated than to search other information. If we can provide the likely queries as UI widgets according to the place where a programmer is inspecting, programmers can select a provided query.

Supporting multi-threaded AD programs AD modules can drastically change the behavior and control flow of the base program, leading to unexpected behavior and resulting in the same complexity that multi-threaded programs are notorious for [24]. Multi-threaded programs are difficult to be analysed and debugged because of the non-determinism in thread scheduling. Failures caused by unexpected thread scheduling may disappear in following runs. Multi-threaded AD programs make debugging tasks even more challenging. We can extend the trace-based debugger to record a trace for each thread with the source-level abstractions. However, issues, such as minimizing impacts on the original program execution and recording multi-threaded-conforming data, need substantial studies.

Enhancing AODG. The evaluation of the effectiveness of the slicing algorithm performed on AODG shows that the precision of the algorithm is low. This means that the algorithm selects much more source code than necessary to be inspected. To increase the precision, we can perform several ways of restricting the resulting slice. For example, specifying a scope of the interesting source code and nodes outside the scope are excluded from the resulting slice. As another example, we can run the program in multiple test cases. For code that is absent in slices of passed cases but present in slices of failed cases, it has a high probability to be the defect.

Conducting usability evaluations. The debugging techniques introduced in this thesis are implemented and provided with graphical user interfaces (GUI). We lack of a systematic evaluation about the usability of the GUIs. The evaluation can help us to enhance the GUIs and also validate our theoretical research.

Appendix A

XText grammar of the Breakpoint Language

```
grammar org.xtext.xaspectj.XAspectJ with org.eclipse.xtext.common.Terminals
generate xAspectJ "http://www.xtext.org/xaspectj/XAspectJ"
XAspectJ :
  (pointcuts += PointcutDeclaration)*
;
PointcutDeclaration :
  name=SimpleNamePattern '(' parameterList=FormalParameterList? ')' ':'
  pointcut=Pointcut ';'
;
BindingParameterList :
  bindings+=BindingParameter (',' bindings+=BindingParameter)*
;
BindingParameter :
  pointcutRef=[PointcutDeclaration | SimpleNamePattern ]
  '(' parameterName=SimpleNamePattern')'
;
Pointcut:
  UnaryPointcut (({AndPointcut.left=current} '&&' | {OrPointcut.left=current} '||')
  right=UnaryPointcut)*
;
UnaryPointcut returns Pointcut :
  BasicPointcut |
  {NegatedPointcut} '!' expression=UnaryPointcut
;
BasicPointcut returns Pointcut :
  '(' Pointcut ')' |
  {ReferencePC} pointcutRef=[PointcutDeclaration | SimpleNamePattern ] '('')' |
  {CallOnPC} 'call''(' callPart=MethodPattern ')''on''(' onPart=ClasstypeDotId ')' |
  {CallPC} 'call' '(' expression=MethodConstructorPattern ')' |
  {ExecutionPC} 'execution' '(' expression=MethodConstructorPattern ')' |
  {InitializationPC} 'initialization' '(' expression=ConstructorPattern ')' |
  {PreinitializationPC} 'preinitialization' '(' expression=ConstructorPattern ')' |
  {StaticintializationPC} 'staticinitialization' '(' expression=ClassNamePatternExpr ')' |
  {GetPC} 'get' '(' expression=FieldPattern ')' |
  {SetPC} 'set' '(' expression=FieldPattern ')' |
  {HandlerPC} 'handler' '(' expression=ClassNamePatternExpr ')' |
  {AdviceexecutionPC} 'adviceexecution' '(' expression=WildcardQualifiedName ')' |
  {WithinPC} 'within' '(' expression=ClassNamePatternExpr ')' |
  {WithincodePC} 'withincode' '(' expression=MethodConstructorPattern ')' |
  {CflowPC} 'cflow' '(' expresssion=Pointcut ')' |
  {CflowbelowPC} 'cflowbelow' '(' expression=Pointcut ')' |
  {ThisPC} 'this' '(' expression=TypeIdStar ')' |
  {TargetPC}'target' '(' expression=TypeIdStar ')' |
  {ArgsPC}'args' '(' expression=TypeIdStarList ')' |
```

```
45    {CompositionPC} 'composition' '(' expression=CompositionExpr ')' |
46    {CheckNpePC} 'checkNPE' '(' ')' |
47    {LocationPC} 'location' '(' expression=LocationExpr ')' |
48    {PathPC} 'path' '(' expression=PathExpr ')' |
49    {BindPC} 'bind' '(' expression=BindList ')' |
50    {IfPC} 'if' '(' expression=IfConditionExpr ')'
51  ;
52  ////////////////////// Method, Constructor and Field Patterns ////////////////////
53  MethodConstructorPattern :
54    ConstructorPattern |
55    MethodPattern
56  ;
57  ConstructorPattern :
58    (modifiersPattern=ModifierPatternExpr)?
59    constructorId=ClasstypeDotNew
60    '(' parametersPattern=FormalPatternList? ')'
61    ('throws' exceptionsPattern=ThrowsPatternList)?
62  ;
63  MethodPattern :
64    (modifiersPattern=ModifierPatternExpr)?
65    returnTypePattern=TypePatternExpr
66    methodId=ClasstypeDotId
67    '(' parametersPattern=FormalPatternList? ')'
68    ('throws' exceptionsPattern=ThrowsPatternList)?
69  ;
70  FieldPattern :
71    (modifiersPattern=ModifierPatternExpr)?
72    typePattern=TypePatternExpr
73    filedId=ClasstypeDotId
74  ;
75  OnExpr :
76    'on''(' expression=ClasstypeDotId ')'
77  ;
78  ////////////////////////////// Modifier Patterns ////////////////////////////////
79  ModifierPatternExpr :
80    (modifiers+=ModifierExpr)+
81  ;
82  ModifierExpr :
83    modifier=Modifier |
84    {NegatedModifiersPattern} '!' modifier=Modifier
85  ;
86  Modifier :
87    'public' | 'private' | 'protected' | 'static' | 'final' | 'abstract' | 'synchronized' |
88    'volatile' | 'transient' | 'native' | 'interface'
89  ;
90  ////////////////////////// Type Pattern Expressions //////////////////////////////
91  TypePatternExpr :
92    UnaryTypePatternExpr
93    (( {AndTypePattern.left=current} '&&' | {OrTypePattern.left=current} '||')
```

```
94    right=UnaryTypePatternExpr)*
95  ;
96  UnaryTypePatternExpr returns TypePatternExpr :
97    BasicTypePattern |
98    {NegatedTypePattern} '!' pattern=UnaryTypePatternExpr
99  ;
100 BasicTypePattern returns TypePatternExpr :
101   {VoidTypePattern} 'void' |
102   {BaseTypePattern} pattern=BaseTypePattern |
103   {ArrayTypePattern} pattern=BaseTypePattern ('['']')+ |
104   '(' TypePatternExpr ')'
105 ;
106 BaseTypePattern returns TypePatternExpr:
107   {PrimitiveTypePattern} pattern=PrimitiveType |
108   NamePattern |
109   {SubTypePattern} pattern=NamePattern '+'
110 ;
111 PrimitiveType :
112   'boolean' | 'byte' | 'char' | 'double' | 'float' | 'int' | 'long' | 'short'
113 ;
114 ////////////////////////////////// Name Patterns ////////////////////////////////////
115 NamePattern :
116   {DotNamePattern} className=WildcardQualifiedName |
117   {DotDotNamePattern} className=WildcardQualifiedName '..' rest=SimpleNamePattern
118 ;
119 WildcardQualifiedName :
120   segments+=SimpleNamePattern ('.' segments+=SimpleNamePattern)*
121 ;
122 terminal SimpleNamePattern:
123   ('a'..'z'|'A'..'Z'|'_'|'*') ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'*')*
124 ;
125 ClasstypeDotId :
126    {DotNameId} name=WildcardQualifiedName |
127    {DotSubNameId} className=WildcardQualifiedName '+' '.' name=SimpleNamePattern |
128    {DotDotNameId} className=WildcardQualifiedName '..' name=SimpleNamePattern |
129    {PatternNameId} '(' classNamePattern=TypePatternExpr ')' '.' name=SimpleNamePattern
130 ;
131 ClasstypeDotNew :
132   {NameNew} 'new' |
133   {DotNameNew} className=WildcardQualifiedName '.' 'new' |
134   {DotSubNameNew} className=WildcardQualifiedName '+' '.' 'new' |
135   {DotDotNameNew} className=WildcardQualifiedName '..' 'new' |
136   {PatternNameNew} '(' classNamePattern=TypePatternExpr')' '.' 'new'
137 ;
138 ////////////////////////////// Parameter List Patterns ////////////////////////////
139 FormalPatternList :
140   formalPatterns+=FormalPattern (',' formalPatterns+=FormalPattern)*
141 ;
142 FormalPattern :
```

```
143   {DotDotFormalPattern} '..' |
144   TypePatternExpr
145 ;
146 FormalParameterList :
147   parameters+=FormalParameter (',' parameters+=FormalParameter)*
148 ;
149 FormalParameter :
150   parameterType=WildcardQualifiedName parameterName=SimpleNamePattern
151 ;
152 TypeIdStarList :
153   parameters+=TypeIdStar (',' parameters+=TypeIdStar)*
154 ;
155 TypeIdStar :
156   FormalPattern
157 ;
158 ////////////////////////////// Throws Patterns ////////////////////////////////
159 ThrowsPatternList :
160   exceptionPatterns+=ClassNamePatternExpr (','
161   exceptionPatterns+=ClassNamePatternExpr )*
162 ;
163 ////////////////////// Class Name Pattern Expressions ////////////////////////
164 ClassNamePatternExpr :
165   UnaryClassNamePatternExpr
166   (({AndClassNamePattern.left=current} '&&' | {OrClassNamePattern.left=current} '||')
167   right=UnaryClassNamePatternExpr)*
168 ;
169 UnaryClassNamePatternExpr returns ClassNamePatternExpr :
170   BasicClassNamePattern |
171   {NegatedClassNamePattern} '!' pattern=UnaryClassNamePatternExpr
172 ;
173 BasicClassNamePattern returns ClassNamePatternExpr :
174   '(' ClassNamePatternExpr ')' |
175   pattern=NamePattern |
176   {SubTypePattern} pattern=NamePattern '+'
177 ;
178 ////////////////////////// Composition Expression ////////////////////////////
179 CompositionExpr :
180   SingleCompositionExpr
181     (({SelectionComposition.left=current} ','
182     ) right = SingleCompositionExpr)*
183 ;
184 UnaryCompositionExpr returns CompositionExpr :
185   BasicCompositionExpr |
186   {NegatedCompositionExpr} '!' expr=UnaryCompositionExpr
187 ;
188 BasicCompositionExpr returns CompositionExpr :
189   '(' CompositionExpr ')' |
190   {ReferenceExpr} bpRef=[PointcutDeclaration | SimpleNamePattern ]
191 ;
```

```
192 ////////////////////////////// Stateful breakpoint ////////////////////////////
193 BindList :
194   bindings+=BindSingle (',' bindings+=BindSingle)*
195 ;
196 BindSingle :
197   bpRef=[PointcutDeclaration | SimpleNamePattern ] '(' bindNameList=BindNameList ')'
198 ;
199 BindNameList :
200   bindNames+=SimpleNamePattern (',' bindNames+=SimpleNamePattern)*
201 ;
202 IfConditionExpr:
203   UnaryConditionExpr
204   (({AndCondition.left=current} '&&' | {OrCondition.left=current} '||')
205   right=UnaryConditionExpr)*
206 ;
207 UnaryConditionExpr returns IfConditionExpr :
208   BasicConditionExpr |
209   {NegatedConditionExpr} '!' condition=UnaryConditionExpr
210 ;
211 BasicConditionExpr returns IfConditionExpr :
212   '(' IfConditionExpr ')' |
213   ComparisonCondition
214 ;
215 ComparisonCondition returns IfCondition :
216   {EqualCondition} left=SimpleNamePattern '==' right=SimpleNamePattern
217 ;
218 ////////////////////////////// Location && Path ////////////////////////////////
219 LocationExpr:
220   filePath=STRING ',' fileName=STRING ',' '[' rangeList=RangeListExpr ']'
221 ;
222 RangeListExpr:
223   ranges+=RangeExpr (',' ranges+=RangeExpr)*
224 ;
225 RangeExpr:
226   {LineRangeExpr} line=INT |
227   {RegionRangeExpr} start=INT '..' end=INT
228 ;
229 PathExpr:
230   bpRefs+=[PointcutDeclaration | SimpleNamePattern ] (' '
231   bpRefs+=[PointcutDeclaration | SimpleNamePattern ])*
232 ;
```

Listing 6.1: XText grammar of BPL introduced in Chapter 3

Appendix B

Simplified join point descriptions on the user interface of the trace-based debugger

Chapter 4 introduces a trace-based debugger with a user interface visualising execution trace. Due to the compact space on the user interface, we need to simplify the textual descriptions of join point in the trace. Following table shows the acronym, the full name, the format of the textual representations, and an example for each join point type. The formats use the following symbols.

Symbol ':' concatenates a class name and a identifier of an instance of that class, e.g., `Obj:1`.

Symbol '-' is used for referring an action. It concatenates the schedule information and the identifier of the action, e.g., `BEFORE-1234`

Symbol '&' refers to the value of a variable, e.g. *&Class1*.

Symbol '/' concatenates an expression with the returned value of that expression, e.g.,
`Obj:1.aFunction()/10`.

Symbol '=' refers to a variable write, e.g., `var=10`.

ACJP	ActionCallJoinPoint
`Action:<actionOID>(<scheduleInfo>-<advicedJoinPointId>)`	
Example: `Action:1(BEFORE-1234)`	

ADJP	AttachmentDeploymentJoinPoint
`AttDepl:<attachmentOID>`	
Example: `AttDepl:1`	

AEJP	ActionExitJoinPoint
`Action:<actionOID>(<scheduleInfo>-<advicedJoinPointId>)/ <actionReturnValue>(<returnType>)`	
Example: `Action:1(AROUND-1234)/10 (int)`	

AUJP	AttachmentUndeploymentJoinPoint
`AttUnDepl:<attachmentOID>`	
Example: `AttUnDepl:1`	

AWJP	ArrayWriteJoinPoint
`<array>[<index>]=<writeValue>`	
Example: `Obj:1[0]=10`	

CRJP	CompositionRuleEvaluationJoinPoint
`Comp:<compositionRuleOID>(<advicedJoinPointId>)`	
Example: `Comp:1(1234)`	

FCJP	FunctionCallJoinPoint
`<context>.<functionName>()`	
Example: `Class1.aStaticFunction()`	

FEJP	FunctionExitJoinPoint
`<context>.<functionName>()/<returnValue> (<returnType>)`	
Example: `Obj:1.aFunction()/10 (int)`	

FRJP	FieldReadJoinPoint
`&<fieldOwner.<varName>/<readValue> (<varType>)`	
Example: `&Class1.aStaticField/10 (int)`	

FWJP	FieldWriteJoinPoint
`<fieldOwner>.<varName>=<writeValue> (<varType>)`	
Example:Obj:1.aField=10 (int)	

LVWJP	LocaVariableWriteJoinPoint
`<varName>=<writeValue> (varType)`	
Example: `var=10(int)`	

PRJP	PrecedenceRuleEvaluationJoinPoint
`Prec:<precedenceRuleOID>(<advicedJoinPointId>)`	
Example: `Prec:1(1234)`	

PCJP	PredicateEvaluationCallJoinPoint
`Pred:<predicateOID>(<interceptedJoinPointId>)`	
Example: `Pred:1(1234)`	

PEJP	PredicateEvaluationExitJoinPoint
`Pred:<predicateOID>(<interceptedJoinPointId>)/<evalResult>`	
Example: `Pred:1(1234)/true`	

THJP	ThrowableJoinPoint
`Throw <throwableOID>()`	
Example: `Throw Obj:1`	

Table 6.1: The formats of textual representation for each join point type.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990. 6

[2] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Technical report, 2004. 23

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, Oct. 2005. 17, 67, 96

[4] Y. Apter, D. H. Lorenz, and O. Mishali. A Debug Interface for Debugging Multiple Domain Specific Aspect Languages. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 47–58, New York, NY, USA, 2012. ACM. 48, 50, 54

[5] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. 3880:135–173, 2006. 57

[6] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the 4th AOSD*, pages 87–98, New York, NY, USA, 2005. ACM. 15, 22, 30

[7] J. S. Baekken. *A Fault Model for Pointcuts and Advice in AspectJ Programs.* Master's thesis, School of Electronical Engineering and Computer Science, Washington State University, 2006. 23

[8] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 384–396, New York, NY, USA, 1993. ACM. 127

[9] C. Bockisch, M. Haupt, M. Mezini, and R. Mitschke. Envelope-Based Weaving for Faster Aspect Compilers. In *NODe/GSEM*, volume 69 of *LNI*, pages 3–18. GI, 2005. 36

[10] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at alia4j. *Journal of Object Technology*, 11(1):7:1–28, Apr. 2012. 29, 34, 105, 108, 122

[11] E. Bodden. Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 492–495, New York, NY, USA, 2011. ACM. 95

[12] E. Bodden, F. Chen, and G. Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, New York, NY, USA, 2009. ACM. 68, 75

[13] J. Bohnet, M. Koeleman, and J. Doellner. Visualizing massively pruned execution traces to facilitate trace exploration. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 57–64, 2009. 124

[14] M. Bruls, K. Huizing, and J. van Wijk. Squarified Treemaps. In *In Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. Press, 1999. 58

[15] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986. 34

[16] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 96–106, New York, NY, USA, 2007. ACM. 95

[17] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design Rationale, Compiler Implementation, and Applications. *ACM Transactions on Programming Languages and Systems*, 28(3), 2006. 45

[18] W. Coelho and G. C. Murphy. Presenting Crosscutting Structure with Active Models. In *Proceedings of the 5th AOSD*, pages 158–168, New York, NY, USA, 2006. ACM. 57

[19] T. Cottenier. The motorola weavr: Model weaving in a large industrial context. In *in Proceedings of the International Conference on AspectOriented Software Development, Industry Track*, 2006. 1

[20] J. K. Czyz and B. Jayaraman. Declarative and visual debugging in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 31–35, New York, NY, USA, 2007. ACM. 124

[21] W. De Borger, B. Lagaisse, and W. Joosen. A Generic and Reflective Debugging Architecture to Support Runtime Visibility and Traceability of Aspects. In *Proceedings of the 8th AOSD*, pages 173–184, New York, NY, USA, 2009. ACM. 22, 54

[22] M. Ducassé. Coca: an automated debugger for C. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 504–513, New York, NY, USA, 1999. ACM. 5, 61, 96

[23] P. E. A. Dürr. *Resource-based verification for robust composition of aspects*. PhD thesis, Enschede, June 2008. 68

[24] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging Aspect-Enabled Programs. In *Proceedings of the 6th international conference on SC*, pages 200–215, Berlin, Heidelberg, 2007. Springer-Verlag. 22, 55, 155

[25] M. Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, Apr. 1997. 2

[26] M. Eisenstadt, J. Domingue, T. Rajan, and E. Motta. Visual knowledge engineering. *IEEE Trans. Softw. Eng.*, 16(10):1164–1177, Oct. 1990. 99

[27] M. D. Ernst. Practical fine-grained static slicing of optimized code. Technical report, Technical Report MSR-TR-94-14, Microsoft Research, 1994. 6

[28] J. Fabry, A. Kellens, and S. Ducasse. AspectMaps: A Scalable Visualization of Join Point Shadows. In *ICPC*, pages 121–130, 2011. 57

[29] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado. An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs. In *Proceedings of the 32nd ICSE - Volume 1*, pages 65–74, New York, NY, USA, 2010. ACM. 21

[30] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation Testing for Aspect-Oriented Programs. In *Proceedings of the 2008 ICST*, pages 52–61, Washington, DC, USA, 2008. IEEE Computer Society. 23

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 108

[32] H. Z. Girgis and B. Jayaraman. JavaDD: a declarative debugger for java. Technical report, 2006. 95, 123, 124

[33] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. *SIGPLAN Not.*, 40(10):385–402, Oct. 2005. 123

[34] J. A. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 6(2):151–170, 1975. 127

[35] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, Jan. 2002. 99

[36] A. Hannousse, R. Douence, and G. Ardourel. Static analysis of aspect interaction and composition in component models. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 43–52, New York, NY, USA, 2011. ACM. 68, 101

[37] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '05, pages 421–430, New York, NY, USA, 2005. ACM. 58

[38] C. Hermanns and H. Kuchen. Hybrid debugging of java programs. In *Software and Data Technologies*, volume 303 of *Communications in Computer and Information Science*, pages 91–107. Springer Berlin Heidelberg, 2013. 123

[39] S. Herrmann, C. Hundt, M. Mosconi, C. Pfeiffer, and J. Wloka. Das Object Teams Development Tooling. *Softwaretechnik-Trends*, 26(4):42–43, 2006. 57

[40] C. Hofer, M. Denker, and S. Ducasse. Design and Implementation of a Backward-In-Time Debugger. In *Proceedings of NODe 2006*, volume P-88, pages 17–32, 2006. 5, 123

[41] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *SIGPLAN Not.*, 25(6):234–245, June 1990. 127

[42] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, July 1989. 127

[43] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, June 1988. 6, 127

[44] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 178–187, Washington, DC, USA, 2004. IEEE Computer Society. 146

[45] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, FOAL '08, New York, NY, USA, 2008. ACM. 68, 101

[46] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[47] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th ECOOP*, pages 327–353, London, UK, UK, 2001. Springer-Verlag. 23

[48] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM. 123

[49] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, Dec. 1990. 6

[50] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society. 6, 127

[51] R. Lencevicius. On-the-fly query-based debugging with examples. In *AADEBUG*, 2000. 123

[52] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA*, pages 304–317, 1997. 123

[53] B. Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003. 5, 105, 123

[54] H. Lieberman. Introduction. *Commun. ACM*, 40(4):26–29, Apr. 1997. 2

[55] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. 1

[56] F. M. Ceccato, P. Tonella. Is AOP code easier to test than OOP code? In *In Workshop on Testing Aspect-Oriented Programs*, 2005. 23

[57] A. Marot and R. Wuyts. Detecting unanticipated aspect interferences at runtime with compositional intentions. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, New York, NY, USA, 2009. ACM. 68, 101

[58] A. Marot and R. Wuyts. Composing aspects with aspects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, New York, NY, USA, 2010. ACM. 68, 75, 96

[59] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, Oct. 2005. 123

[60] P. M. Merrick and A. F. H. R. Laboratory. *Debugging techniques used by experienced programmers to debug their own code [microform] / Pamela M. Merrick*. Air Force Human Resources Laboratory, Air Force Systems Command Brooks Air Force Base, Tex, 1990. 127

[61] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and Modular Predicate Dispatch for Java. *ACM Trans. Program. Lang. Syst.*, 31(2):7:1–7:54, Feb. 2009. 45

[62] S. Mirghasemi. Query-point debugging. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 763–764, New York, NY, USA, 2009. ACM. 123

[63] I. Nagy. *On the design of aspect-oriented composition models for software evolution*. PhD thesis, Enschede, June 2006. 68, 101

[64] I. Nagy, R. van Engelen, and D. van der Ploeg. An overview of mirjam and weavec. In R. van Engelen and J. Voeten, editors, *Ideals: evolvability of software-intensive high-tech systems*, pages 69–86. Embedded Systems Institute, Eindhoven, 2007. 1

[65] R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Softw. Pract. Exper.*, 21(2):209–229, Feb. 1991. 5

[66] R. M. Parizi and A. A. A. Ghani. AJcFgraph - AspectJ control flow graph builder for aspect-oriented software. 3:170–181, 2008. 146

[67] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, UK, 2002. Springer-Verlag. 124

[68] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS 98*, pages 219–234, 1998. 106

[69] J. Pfeiffer and J. R. Gurd. Visualisation-Based Tool Support for the Development of Aspect-Oriented Programs. In *Proceedings of the 5th AOSD*, pages 146–157, New York, NY, USA, 2006. ACM. 21, 57

[70] G. Pothier and E. Tanter. Extending Omniscient Debugging to Support Aspect-Oriented Programming. In *Proceedings of SAC*, pages 266–270, New York, NY, USA, 2008. ACM. 55, 99, 121

[71] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 535–552, New York, NY, USA, 2007. ACM. 5, 105, 121

[72] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 147–158, New York, NY, USA, 2004. ACM. 101

[73] E. Satterthwaite. Debugging tools for high level languages. *Software: Practice and Experience*, 2(3):197–217, 1972. 2

[74] T. Schafer and M. Mezini. Towards More Flexibility in Software Visualization Tools. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '05, pages 20–, Washington, DC, USA, 2005. IEEE Computer Society. 58

[75] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 30–39, New York, NY, USA, 2003. ACM. 145

[76] A. Sewe, C. Bockisch, and M. Mezini. Redundancy-Free Residual Dispatch: Using Ordered Binary Decision Diagrams for Efficient Dispatch. In *Proceedings of the 7th workshop on FOAL*, pages 1–7, New York, NY, USA, 2008. ACM. 34

[77] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proceedings of the 2nd AOSD*, pages 21–29, New York, NY, USA, 2003. ACM. 57

[78] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994. 127

[79] Y. Usui and S. Chiba. Bugdel: An aspect-oriented debugging system. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 790–795, Washington, DC, USA, 2005. IEEE Computer Society. 95

[80] M. van't Riet, H. Yin, and C. Bockisch. The potential of omniscient debugging for aspect-oriented programming languages. In *Proceedings of the 1st workshop on Comprehension of complex systems*, CoCoS '13, pages 13–16, New York, NY, USA, 2013. ACM. 11

[81] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984. 6

[82] D. Willis, S. J. Noble, and D. J. Pearce. Squirrel: A query based debugger for java, 2005. 123

[83] G. V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, Dec. 2004. 2

[84] V. Wulf, V. Pipek, and M. Won. Component-based tailorability: Enabling highly flexible software applications. *Int. J. Hum.-Comput. Stud.*, 66(1):1–22, Jan. 2008. 58

[85] G. Xu and A. Rountev. AJANA: a general framework for source-code-level interprocedural dataflow analysis of aspectj software. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 36–47, New York, NY, USA, 2008. ACM. 127, 128, 146

[86] H. Yin and C. Bockisch. Developing a generic debugger for advanced-dispatching languages. In *Proceedings of the Third International Workshop on Academic Software Development Tools and Techniques*, WASDeTT-3. 11

[87] H. Yin, C. Bockisch, and M. Aksit. A fine-grained debugger for aspect-oriented programming. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, New York, NY, USA, 2012. ACM. 11, 75

[88] H. Yin, C. Bockisch, and M. Aksit. A pointcut language for setting advanced breakpoints. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 145–156, New York, NY, USA, 2013. ACM. 11

[89] H. Yin, C. M. Bockisch, and M. Akşit. A fine-grained, customizable debugger for aspect-oriented programming. *Transactions on Aspect-Oriented Software Development X*, 7800:1–38, 2013. 11

[90] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. 2, 127

[91] S. Zhang and J. Zhao. On identifying bug patterns in aspect-oriented programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 431–438, Washington, DC, USA, 2007. IEEE Computer Society. 116

[92] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 251–, Washington, DC, USA, 2002. IEEE Computer Society. 127, 128, 145