Extending Omniscient Debugging to Support Aspect-Oriented Programming

Guillaume Pothier[*]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
gpothier@dcc.uchile.cl

Éric Tanter [†]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

Debugging is a tedious and costly process that demands a profound understanding of the dynamic behavior of programs. Debugging *aspect-oriented* software is even more difficult: to implement the semantics of aspects, a number of implicit activities are performed, whose relation to source code is less direct to grasp. We show how *omniscient debugging*, a technique that consists in recording the activity of a program to later navigate in its history, can be extended to suit the particularities of aspect-oriented software. By enhancing program understandability, improvements to the tooling associated with aspect orientation will encourage the widespread acceptance of this emerging paradigm.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; D.2.6 [**Programming Environments**]: Integrated environments; D.3.4 [**Processors**]: Debuggers

General Terms

Design, Languages, Reliability

Keywords

Omniscient debugging, aspect-oriented programming

1. INTRODUCTION

Aspect-Oriented Programming (AOP) provides means for proper modularization of crosscutting concerns, whose implementation would otherwise be scattered across several modules [6]. In most AOP approaches, modularization is achieved by defining *aspects* that affect the structure or the behavior of a *base program* that is mostly unaware, at least syntactically, of these aspects. One drawback of this approach is that it makes the comprehension of AOP-based systems more difficult: understanding a piece of code might require the understanding of the whole system, or at least of its aspects [13].

The time-consuming task of *debugging* has a significant impact on the cost of software [14]. Most of the time is usually spent locating the *cause* of the bug, often using a tedious trial-and-error approach, while actually *fixing* the bug can be trivial [5]. A strategy frequently used by programmers is to *mentally simulate* the execution of the program [5]. Thus the complexity of debugging increases with the level of abstraction of the programming paradigm because the correspondence between source code and runtime behavior becomes less direct. For instance, in object-oriented programming, one cannot always know by looking at the source code which method will be evaluated as a result of a method call, because of the dynamic dispatch mechanism. This is even more true for AOP, where the behavior of a given piece of code can be altered to an arbitrary degree by an aspect in another source code file. Section 2 details these difficulties.

A promising direction for alleviating the debugging burden is the use of *omniscient debuggers*, also known as *back-in-time* or *post-mortem* debuggers [7, 9, 10, 11]. They record the events that occur during the execution of the debugged program, and then let the user conveniently navigate through the obtained execution trace. This approach combines the advantages of log-based debugging –past activity is never lost– and those of breakpoint-based debugging –easy navigation, step-by-step execution, complete stack inspection. An omniscient debugger can simulate step-by-step execution forward *and backward*, and can immediately answer questions that would otherwise require a significant effort, like *"At what point was variable* `x` *assigned value* `y`*?"* or *"What was the state of object* `o` *when it was passed as an argument to the method* `foo`*?"*.

This paper shows how omniscient debugging can be extended so as to embrace aspect-oriented programming. We first present an analysis of the current situation for debugging aspect-oriented programs, focusing on the case of AspectJ, since it is the best supported language to date (Sect. 3). We then describe a set of extensions to the TOD open source debugger [11] that greatly enhance the understanding of the dynamics, and therefore the task of debugging, of aspect-oriented programs (Sect. 4).

[*]G. Pothier is financed by a PhD grant from NIC Chile.
[†]É. Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and by FONDECYT Project 11060493.

```
public aspect Foo {
  pointcut cond(int x):  call(* A.foo(int))
    && target(B) && args(x) && if(x<3);

  before(int x):  cond(x) {
    System.out.println("Bingo:  "+x);
  } }
```

Figure 1: Example aspect in AspectJ.

2. AOP AND DEBUGGING

This section briefly introduces the AspectJ [8] language, and then explains the issues brought by AOP to software understanding and debugging.

2.1 AspectJ

AspectJ [8] extends the Java language with a new unit called *aspect* that permits to implement crosscutting concerns modularly. AspectJ supports two kinds of crosscutting: *dynamic crosscutting* makes is possible to define additional behavior to be executed when certain conditions occur in the base program; *static crosscutting* makes it possible to modify the static structure of a program, *e.g.*adding new methods or modifying the class hierarchy.

In AspectJ a *join point* represents a well-defined point in the execution of a program, such as method call, field write, exception handler execution, etc. The (static) location of a join point in the source code is called a join point *shadow*. A join point may also specify a dynamically-evaluated condition, called the *residue*, to determine at runtime whether a join point shadow actually is the expected join point.

Join points of interest are grouped into a *pointcut* in order to specify the places where an aspect actually affects a base application. Pointcuts are specified using several primitive *pointcut designators* (PCDs) which can be combined using the standard logical operators. For example, the aspect in Figure 1 defines a pointcut named `cond` that combines several primitive PCDs in order to select calls to method `foo` of class `A` when the actual type of the receiver is `B` (a subclass of `A`) and the parameter is less than 3; additionally it exposes the parameter as context information.

Finally, the crosscutting behavior that should be applied upon occurrences of join points matched by a given pointcut definition is called an *advice*, which is a method-like construction that defines additional behavior to execute at certain join points. Advices must be explicitly bound to pointcuts. In the example of Figure 1, an advice is called *before* the pointcut `cond` matches and prints an informative message using exposed context information.

2.2 Debug model for AOP

The debug model for AOP developed in [4] is helpful in understanding the difficulties of debugging AO programs. Of particular interest are a classification of AOP *activities*, a comprehensive *fault model* and a definition of *debugging obliviousness* and *debugging intimacy*.

Classification of AOP activities. The execution of an AO program consists of different activities. Beyond base code execution, there is one *explicit* activity, namely advice execution, and several *implicit* activities used to coordinate advices with base code [3, 4]: dynamic aspect instantiation and selection, *i.e.* determining which aspect instance should apply; residue evaluation; aspect activation, *i.e.* gathering context information and transferring control to the advice; and bookkeeping for specific features, such as maintaining a thread-local stack for `cflow` pointcuts. A much more detailed decomposition is given in [3].

Fault model. The existence of implicit AOP activities increases the difficulty of debugging AO programs because many instructions are executed at runtime that are not explicit in the source code. The following are examples of AOP-specific fault types [4]: (1) incorrect pointcut descriptor, when a pointcut declaration does not have the intended effect, (2) incorrect aspect composition, when several aspects match the same join point and are not executed in the expected order, (3) adverse changes on base program, when an aspect alters the functionality of the base program in such a way that it ceases to work properly, and (4) incorrect context exposure, when the context is not exposed as intended to an advice.

Debugging obliviousness and intimacy. In the context of debugging AO programs, debugging obliviousness is the capacity to ignore all AOP-related activities. Conversely, debugging intimacy is the capacity to observe all activities in their full details [4]. Instead of considering obliviousness and intimacy as exclusive alternatives, our proposal describes how to support a *range* of options between these two extremes (Section 4). This makes it possible for the programmer to choose the appropriate level of intimacy depending on the debugging task at hand.

3. STATE OF THE PRACTICE

This section analyzes tools that can be currently used to debug AspectJ programs, and emphasizes on their limitations with respect to the debug model of Section 2.2.

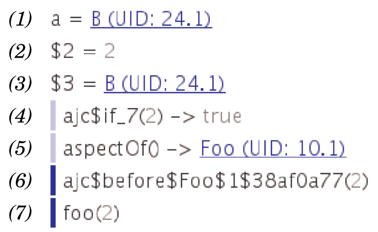
3.1 The AJDT debugger

A major asset for the use of AspectJ is the AspectJ Development Tools (AJDT), a set of plugins for the Eclipse IDE [2]. It provides various features that help understanding AO programs, in particular: *(a)* markers that show join point shadows in the base source code and permit to jump from the shadow to the corresponding pointcut definition (and vice versa), and *(b)* high-level visualizations that show the scattering of join point shadows in whole packages.

Debugging in AJDT is rather ad-hoc, neither fully oblivious nor fully intimate. When the execution of the debugged program is halted at a breakpoint, the programmer can use the traditional *step-over* and *step-into* operations.

Step-into provides a certain level of intimacy. Invoking step-into when an advice is about to be executed actually steps into the code of the advice, but with an extra step where the debugger shows the first line of the file that defines the aspect: this actually corresponds to the *dynamic aspect selection* AOP activity, but this is not explicit in the debugger (and rather misleading).

The level of intimacy in the evaluation of residues is however not consistent. In the pointcut definition of Figure 1 the residue comprises two conditions: (1) the actual type of the target must be `B` (a subclass of `A`), and (2) the argument `x` must be less than 3. When invoking step-into in `a.foo(2)`, only the `if` condition is stepped into: the condition on the



```
(1)  a = B (UID: 24.1)
(2)  $2 = 2
(3)  $3 = B (UID: 24.1)
(4)  | ajc$if_7(2) -> true
(5)  | aspectOf() -> Foo (UID: 10.1)
(6)  | ajc$before$Foo$1$38af0a77(2)
(7)  | foo(2)
```

These events correspond to the execution of `a.foo(2)` with the aspect of Fig. 1. The \$2 and \$3 represent synthetic local variables added by the AspectJ compiler.

Figure 2: List of events corresponding to a conditional pointcut.

target is silently stepped over. This is due to the semantics of the step-into operation of Java: the execution halts when it reaches another source code line, but the `instanceof` operation implementing the type test is considered to be on the same line than the rest of the AOP activities.[1]

Step-over, on the other hand, provides full obliviousness: all AOP activities on the current line are ignored, including advice execution. But there is no way to step into a method call on the current line while ignoring AOP activities.

3.2 AOP debugging with TOD

TOD [11] is an open source omniscient debugger for Java integrated in the Eclipse IDE. It is able to debug AspectJ programs but has no special provisions for handling the obliviousness/intimacy trade-off.

Figure 2 shows the events that are registered when executing a piece of code affected by an aspect. Lines 2 to 6 correspond to AOP-specific activities: lines 2, 3 and 6 correspond to aspect activation, line 4 corresponds to residue evaluation and line 5 to aspect selection.

Compared to AJDT, TOD provides a bit more debugging intimacy: lines 2 and 3, that are a part of the context exposure mechanism, are not shown in AJDT; line 5 explicitly shows a call to `aspectOf`, while AJDT showed the first line of the aspect source file. For the residue evaluation, both TOD and AJDT behave in the same inconsistent way: the `if()` condition is shown but the test on the actual type of the target is not. In the case of TOD the reason is that only method calls and actions that change the state of the program are registered; the `instanceof` operation implementing the type test is not registered.

4. OMNISCIENT DEBUGGING FOR AOP

In this section we describe several extensions to omniscient debugging that facilitate the debugging of AO programs. These extensions have been integrated in the TOD open source omniscient debugger.

4.1 Improving intimacy of residue evaluation

Section 3 showed that the level of intimacy for the evaluation of join point residues is not consistent in existing debugging solutions. In the case of an omniscient debugger such as TOD, only method calls and actions that modify the state of the program are registered; tests such as `instanceof` are not. While this permits to reduce the number of events to register, this also hides useful information. In order to support full debugging intimacy, an omniscient debugger must register the outcome of all the tests that occur during that activity. For an omniscient debugger that captures events through a customized runtime, the runtime has to be able to emit events indicating the outcome of tests. For an omniscient debugger that relies on code instrumentation to generate events, such as TOD, this requires the insertion of additional instrumentation to the program: event generation code is now inserted at locations where an edge of the *control dependence graph* of the program is traversed [15]. Figures 3c and 3d show how full intimacy in residue evaluation adds details to the event list of Figure 2.

4.2 Identification of AOP activities

In aspect language implementations, instructions that represent AOP activities are *woven* with base code instructions so as to provide the semantics of aspects. This weaving process can be *invasive* (the base program is modified to include those instructions, either at the source level or at the binary level), or *noninvasive* (AOP-specific instructions are performed by an AOP-aware runtime) [4]. In both cases it is possible to determine at runtime the AOP activity corresponding to each executed instruction, as well as the pointcut corresponding to each AOP-specific instruction. The case of noninvasive weaving is the easiest: the runtime can be extended to provide the required information, as it is fully responsible for implementing AOP semantics.

Because the current implementations of AspectJ use invasive weaving, we use a *tagging* scheme [3] to identify AOP activities. Each instruction is given, at weave time, a tag that indicates its activity, as well as the identifier of the corresponding pointcut (for AOP-specific instructions). Additionally, runtime propagation rules are defined to ensure, for example, that code from the base program is considered as base code when called directly from the base program, but as advice code when called from an advice. In this way it is always possible to determine the activity to which the currently-executing instruction belongs.

Once the activity of each instruction is identified, it is possible to tag all the events registered by the omniscient debugger with the corresponding activity and pointcut identifier where applicable. This enables debugging obliviousness, as events tagged as AOP-specific can be concealed (Fig. 3a). Moreover this also improves intimacy, as the role of each event can be made explicit (Fig. 3d).

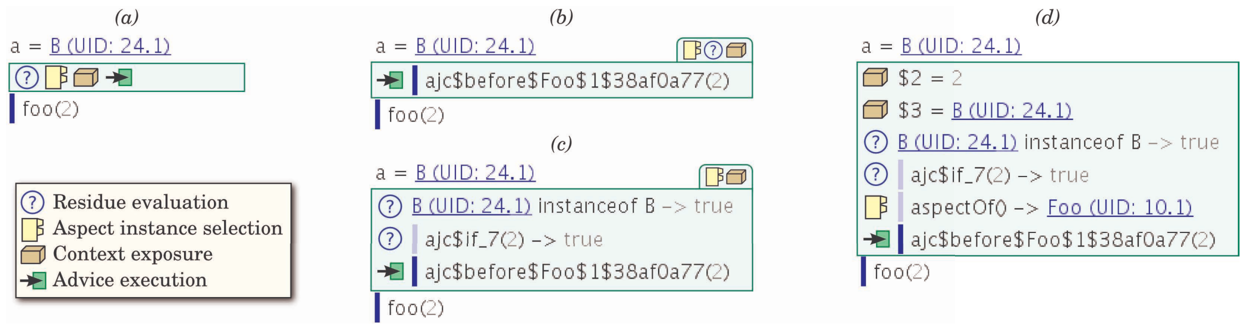
4.3 Obliviousness/intimacy trade-off

In Sections 4.1 and 4.2 we showed how full intimacy and full obliviousness can be achieved. However an AOP debugger should permit to choose the appropriate level of detail for the task at hand. Figure 3 shows increasing level of detail for the same sequence of events, in our extension of TOD.

If obliviousness is required, sequences of AOP-related events between two base code events are collapsed into one line[2] (Fig. 3a). This line provides a visual summary of the activities of the collapsed sequence: the programmer therefore knows at a glance that, before the call to `foo` a conditional

[1]There is a step-into mode that halts at each bytecode (`STEP_MIN` instead of `STEP_LINE`), but it is not used by Eclipse, nor by any other debugger we know of. It would probably require a disassembled view of the bytecode of the debugged activity to be useful.

[2]A completely oblivious debugger would not even show that line, but it seems rather counterproductive.



(a)

a = B (UID: 24.1)

foo(2)

(b)

a = B (UID: 24.1)

ajc$before$Foo$1$38af0a77(2)

foo(2)

(c)

a = B (UID: 24.1)

B (UID: 24.1) instanceof B -> true

ajc$if_7(2) -> true

ajc$before$Foo$1$38af0a77(2)

foo(2)

(d)

a = B (UID: 24.1)

$2 = 2

$3 = B (UID: 24.1)

B (UID: 24.1) instanceof B -> true

ajc$if_7(2) -> true

aspectOf() -> Foo (UID: 10.1)

ajc$before$Foo$1$38af0a77(2)

foo(2)

Residue evaluation

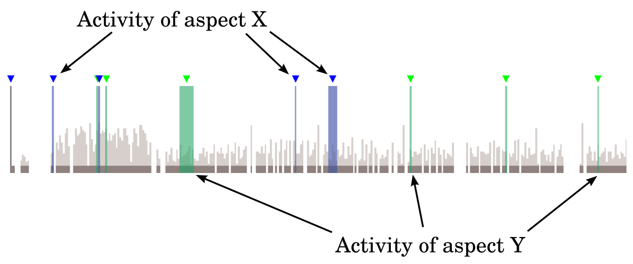
Aspect instance selection

Context exposure

Advice execution

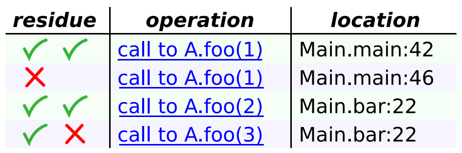
In *(a)* debugging obliviousness is obtained by collapsing the events that correspond to AOP-related activities into a single line. Full intimacy is shown in *(d)*, and two intermediate levels of intimacy are shown in *(b)* and *(c)*. Each icon represents a kind of AOP activity.

Figure 3: The obliviousness/intimacy trade-off in an event list.



Grey bars show the density of events; the superimposed colored boxes indicate times at which AOP activities occurred.

Figure 4: Aspect mural.



residue	operation	location
✓ ✓	call to A.foo(1)	Main.main:42
✗	call to A.foo(1)	Main.main:46
✓ ✓	call to A.foo(2)	Main.bar:22
✓ ✗	call to A.foo(3)	Main.bar:22

History of the pointcut of Fig. 1. The *residue* column shows the tests that passed and those that did not. Clicking on the operation shows the corresponding event in its context.

Figure 5: Pointcut history view (mock-up).

pointcut was evaluated, the advice was indeed called and some context was exposed.

If intimacy is required, the sequence can be gradually expanded to show the details of the AOP activities (Fig. 3b to 3d). First only the event that corresponds to the advice call is shown; this permits to easily step into the execution trace of the advice in the case where no fault in AOP-specific activities is suspected. Otherwise the details of the implicit AOP activities can be revealed: for instance if the advice was erroneously executed, residue evaluation events can be shown (Fig. 3c). If maximum detail is needed, it is possible to show the complete sequence of events (Fig. 3d). The icons in front of each event help understanding the AOP activities.

4.4 Bird's eye views

As an omniscient debugger registers the whole history of the debugged program, it can provide certain summarized views on its execution. These views are useful to abstract away from the details of execution events. For example TOD provides *murals* that show the evolution of the density over time of events that meet certain criteria [11]. With the event tagging scheme mentioned in Sect. 4.2, our extension of TOD provides *aspect murals* that show the activity of an aspect during the execution of the program. Such *temporal* views of AOP activity are the dynamic counterpart of the static crosscutting views of AJDT (Sect. 3.1). They are particularly helpful in understanding the interplay between aspects and base code, as well as between different aspects. Figure 4 shows an aspect mural where two aspects are selected and their activity is represented by two distinct colors. It is also possible to show the activity of aspects on methods of a single class, or of a single instance.

Another interesting view (currently under development) shows the execution history of the join point shadows of a particular pointcut. Pointcut history views are particularly useful for pointcuts with a residue, as they show which occurrences of join points matched and which ones did not. It is even possible to examine in details the individual residue conditions, facilitating the resolution of the *incorrect pointcut descriptor* AOP fault (Sect. 2.2). On Figure 5, on the second line the condition on the type of the target was not verified, shortcutting further evaluations; on the last line the condition on the argument was not verified. Note that while it is easy to determine the number of tests that passed during the residue evaluation, determining the precise pointcut condition corresponding to each test requires a non-trivial static analysis (not tackled yet).

5. RELATED WORK

The comprehensive debug model for AOP presented in [4] has been lengthy discussed in this paper. The model also presents topics that are orthogonal to omniscient debugging such as the ability to use *edit-and-continue* debugging, or to introduce new aspects at runtime. It introduces Wicca, a dynamic AOP system for the C# language, which excels at debugging intimacy but lacks debugging obliviousness.

Static analysis is an important tool for program understanding. The *whole execution traces* presented in [15] offer a compact yet comprehensive representation of program activity and permit to perform advanced semantic queries, such as dynamic *slices*, that identify the parts of a program that directly or indirectly affect the value of a given variable at a given program point. A solution for computing slices of AO programs is presented in [16].

Using omniscient debugging for AOP was first explored in [12]. Their approach is based on slicing and therefore focuses on how to delimit regions of interest *in the source code* for a particular debugging task. Our approach in contrast focuses more on letting the programmer explore the details of the activities related to aspects, at the level of *runtime events*; abstraction on the execution trace is provided by summary views (aspect murals and pointcut history). Combining both approaches seems very promising.

We used TOD [11] for our analysis of the state of the practice. TOD is a *scalable* omniscient debugger for Java that can cope with huge amounts of events. It also offers a flexible trace model that can be easily extended for our purposes. The first omniscient debugger for Java, presented in [9], does not provide such scalability but has interesting features such as the ability to resume execution from an arbitrary point in time. ZStep95 [10] is a *reversible stepper* for Lisp that provide animated views of data structures in addition to the standard features of an omniscient debugger. None of these debuggers have special support for AOP.

6. CONCLUSION

In order to ease the debugging of aspect-oriented programs, we propose a number of extensions to omniscient debuggers: collapsable list of events to hide unwanted details of AOP activities, activity icons to help understand AOP activities, more thorough event model for observing the details of residue evaluation, and high-level views such as aspect murals to help understand the interplay between aspects and base code. These extensions, implemented on TOD, are designed to address issues observed in both traditional breakpoint-based debuggers and current omniscient debuggers when debugging programs with aspects. Improving the debugging experience is crucial for the widespread acceptance of AOP in industry.

We are currently performing a systematic evaluation of the attainable intimacy for various AspectJ features not discussed here such as around advice and control-flow pointcuts. Addressing the obliviousness/intimacy trade-off for intertype declarations would also be of great value, considering the importance of structural aspect mechanisms in practice [1].

7. REFERENCES

[1] S. Apel, C. Kstner, and S. Trujillo. On the necessity of empirical studies in the assessment of modularization mechanisms for crosscutting concerns. In *1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM.07)*, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society Press. In conjunction with ICSE 2007.

[2] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with AJDT. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.

[3] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of aspectj programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–169, New York, NY, USA, 2004. ACM Press.

[4] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *SC2007: Proceedings of Software Composition 2007*, 2007.

[5] M. Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997.

[6] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001.

[7] C. Hofer, M. Denker, and S. Ducasse. Implementing a backward-in-time debugger. In *Proceedings of NODe'06*, volume P-88, pages 17–32. Lecture Notes in Informatics, 2006.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[9] B. Lewis. Debugging backwards in time. In M. Ronsse and K. D. Bosschere, editors, *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Ghent, Belgium, 2003.

[10] H. Lieberman and C. Fry. ZStep 95: A reversible, animated source code stepper. In J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization — Programming as a Multimedia Experience*, pages 277–292, Cambridge, MA-London, 1998. The MIT Press.

[11] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *OOPSLA'07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications (to appear)*, 2007.

[12] S. Ritzkowski. Omniscient slicing for aspect-debugging. Diploma thesis, Software Technology Group, Darmstadt University of Technology, Germany, 2006. `http://www.st.informatik.tu-darmstadt.de/public/Thesis.jsp?id=81`.

[13] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM Press.

[14] M. Xie and B. Yang. A study of the effect of imperfect debugging on software development cost. *IEEE Trans. Software Eng*, 29(5):471–473, 2003.

[15] X. Zhang and R. Gupta. Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.*, 2(3):301–334, 2005.

[16] J. Zhao. Slicing aspect-oriented software. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 251, Washington, DC, USA, 2002. IEEE Computer Society.