A Fine-Grained Debugger for Aspect-Oriented Programming

Haihan Yin, Christoph Bockisch, Mehmet Akşit
Software Engineering group, University of Twente, 7500 AE Enschede, the Netherlands
{h.yin, c.m.bockisch, m.aksit}@cs.utwente.nl

Abstract

To increase modularity, aspect-oriented programming provides a mechanism based on implicit invocation: An aspect can influence runtime behavior of other modules without the need that these modules refer to the aspect. Recent studies show that a significant part of reported bugs in aspect-oriented programs are caused exactly by this implicitness. These bugs are difficult to detect because aspect-oriented source code elements and their locations are transformed or even lost after compilation. We investigate four dedicated fault models and identify ten tasks that a debugger should be able to perform for detecting aspect-orientation-specific faults. We show that existing debuggers are not powerful enough to support all identified tasks because the aspect-oriented abstractions are lost after compilation.

This paper describes the design and implementation of a debugger for aspect-oriented languages using a dedicated intermediate representation preserving the abstraction level of aspect-oriented source code. We define a debugging model which is aware of aspect-oriented concepts. Based on the model, we implement a user interface with functionalities supporting the identified tasks, like visualizing pointcut evaluation and program composition.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids; D.3.2 [*Language Classifications*]: Very high-level languages

General Terms Language, Design

Keywords Debugger, AOP, visualization, advanced-dispatching, fine-grained intermediate representation

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. [12] focuses on the fault-proneness in evolving aspect-oriented programs. They investigated the aspect-oriented (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 aspect-oriented 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 [17]. 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.

Aspect-oriented languages are nowadays compiled to the intermediate representation (IR) of an established non-AO language; this usually entails transforming code already provided in that IR [3], 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 a debugger existing 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 aspect-oriented 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.

Several research works discuss AOP debuggers to provide information closer to the source code, such as the composite source code in Wicca [11], the aspect-aware break-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'12, March 25-30, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1092-5/12/03. . . $10.00

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

point model in AODA [10], or the identified AOP activities in TOD [18]. 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 aspect-oriented 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 paper, 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 like the one by Ferrari, mentioned above. Our debugger is aware of aspect-oriented 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 aspect-oriented 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.

2. Problem Analysis and Requirements

Recently fault models for AOP languages have been researched with the target to systematically generate tests which execute all potentially faulting program elements. We can use the results of these studies to derive the capabilities required from 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.

2.1 AOP Fault Models

We have investigated four fault models proposed in the literature and summarize them in table 1. In our study, we exclude faults related to static features like inter-type declarations because the static code inspection tools offered by modern development tools like the AJDT are already sufficient for localizing these faults. The first column shows the fault model by Alexander et al. [1] which contains examples of AOP-specific faults, like incorrect pointcut strength. Ceccato et al. [16] extend this model with three types concerning exceptional control flow and inter-type declarations (ITD). Ferrari et al. [13] 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 [4] which follows a similar approach; he focuses on AspectJ [15] 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.

2.2 Detecting faults

When a programmer encounters an error during the execution of an AspectJ program, this can be caused by a faults 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 should be applied. 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[2], rerun the program, analyze program states, and eventually (**T2**) locate faulty constructs.

2.2.1 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 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.

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

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	Mixed 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/ cflow/ cflowbelow/ if/ within/ withincode	**Incorrect context**
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 requirements**

Table 1. A systematic fault model for aspect-oriented programs

Incorrect designator The programmer may also encircle 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.

2.2.2 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 three types of composition control in AspectJ influencing the execution order of advices at shared join points: advice-type specification, precedence declaration and lexical order. Advice-type specification, e.g., the keywords `before` or `after`, define the order between advices relative to the join point. Precedence declaration (`declare precedence`) defines the partial order between different aspects. The precedence of advice defined in the same aspect is determined by their lexical order.

To detect this type of fault, 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 Advice 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, ideally at runtime.[3]

2.3 State-of-the-art

Table 2 summarizes the required debugging tasks identified in the previous sub-sections and gives them short names. In the following we discuss how these tasks are supported by the traditional Java Debugger and by AOP debuggers proposed in the literature.

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	Excluding and adding AO definitions

Table 2. Tasks that an ideal AOP debugger should perform

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). 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. [10] is built based on a debugging interface which restores some source-level abstractions from the bytecode. Entities comprising the debugging interface model many AspectJ concepts, such as join points, advices, etc. The debugging interface allows to query advices applied on 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, 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).

Wicca [11] 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., for checking if programs are correctly composed. Wicca also allows to enable/disable aspects at runtime. Though Wicca fully supports T8, T10, it poorly supports our other identified tasks because it debugs the woven code. 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.

Pothier and Tanter [18] 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, only T1, T2, T6, T7, T8 are partially supported in TOD.

2.4 Requirements for an AOP Debugger

Based on the above observations and discussions, we formulate requirements for a dynamic debugger dedicated to aspect-oriented programs. In the following three 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 location.

[3] Dynamic (de-)activation of aspects or advice bears the risk to leave the aspect in a wrong state, e.g., when join points at which the aspect performs an initialization have already passed. But often aspects are less complex and being able to (de-)activate them at runtime is an efficient debugging technique—it must be used with caution, though.

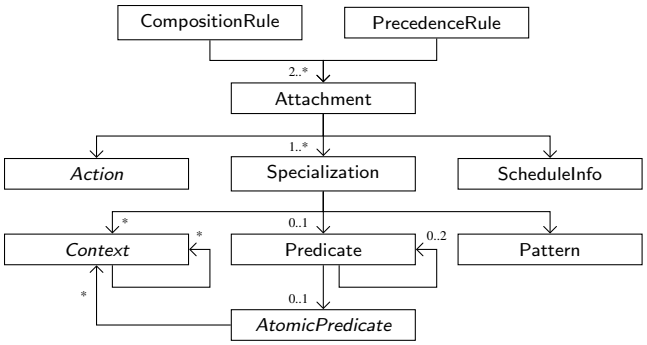

Figure 1. The LIAM meta-model of advanced dispatching

- A fine-grained debugging interface must be provided to allow observation of and interaction with the execution at the granularity of AO abstractions.
- 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 can be performed.

3. Debugging Information

We choose to base the implementation of the debugger on our previous work, a generic implementation architecture of so-called advanced-dispatching (AD) languages which includes AOP languages. One of the main components of this *ALIA4J* architecture[4] [7] is a meta-model of AD declarations, called *LIAM*[5]. 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 AO 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 AO-specific way. This fact as well as the declarative and fine-grained nature of LIAM facilitate the support for all identified debugging tasks.

3.1 Aspect-Oriented Intermediate Representation

The meta-model, LIAM, itself defines *categories* of concepts 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.

Figure 1 shows the meta-entities of LIAM, discussed in detail by Bockisch et al. [6, 7], which capture the core concepts underlying the various dispatching mechanisms. The meta-entities *Action*, *AtomicPredicate* and *Context* can be refined to concrete concepts; we provide refinements for several languages, including AspectJ [7].

An *Attachment* corresponds to a unit of dispatch declaration, roughly corresponding to a pointcut-advice pair in AspectJ. *Action* specifies functionality that may be executed as the result of dispatch (e.g., the body of an advice). *Specialization* defines static and dynamic properties of state on which dispatch depends. *Pattern* specifies syntactic and lexical properties of the dispatch site. *Predicate* and *Atomic Predicate* entities model conditions on the dynamic state a dispatch depends on. *Context* entities model access to values like the called object or argument values. The *Schedule Information* models the time relative to a join point when the action should be executed, i.e., before, after or around. Finally, *Precedence Rule* models partial ordering of actions and *Composition Rule* models the applicability of actions at a shared join point; for example, overriding can be expressed by this.

3.2 XML-based LIAM model

Figure 2 shows the life cycle of the debugging information related to AO features in our approach. Following the bold directed lines, AO-specific information is first written in the source code of aspects and then compiled into a separate XML file containing serialized LIAM-based declarations. At runtime, the XML file is deserialized and the program is executed taking the aspect definitions into account.

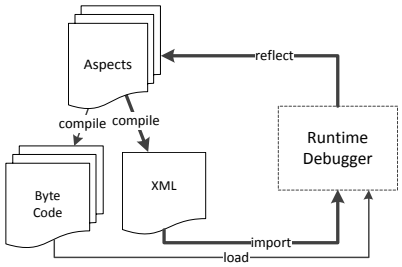

Figure 2. Debugging information life cycle

This approach requires a specific compiler to generate the IR. In the context of this paper, we just elaborate on our implementation of an AspectJ compiler based on the *abc* compiler [3]. As an example of the compilation, consider the AspectJ code in listing 1. After compilation, it is transformed into an *Attachment* XML element presented in listing 2.

There is a many-to-many relationship between source language constructs and LIAM entities. For example in listing 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 4–13, listing 2), as well as a value that is exposed to associated advice (represented by the *Context* in lines 14–17). The pointcut

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

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

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 locations in the source code which both are stored in our IR, as shown on lines 5–6 and 10–11 in listing 2.

With our intermediate representation presented above, we support the task *locating constructs* (**T2**) presented in section 2.3.

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

Listing 1. An aspect example in AspectJ

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

Listing 2. XML-based AO intermediate representation

4. Infrastructure of the AOP debugger

Extending figure 2, the overall structure of our debugger is presented in the figure 3. It consists of a debuggee side and a debugger side; both sides communicate via the *Java Platform Debugger Architecture* (JPDA)[6] 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. Nevertheless, our goal is to support aspect-oriented programs and our case study is based on AspectJ.

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

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.

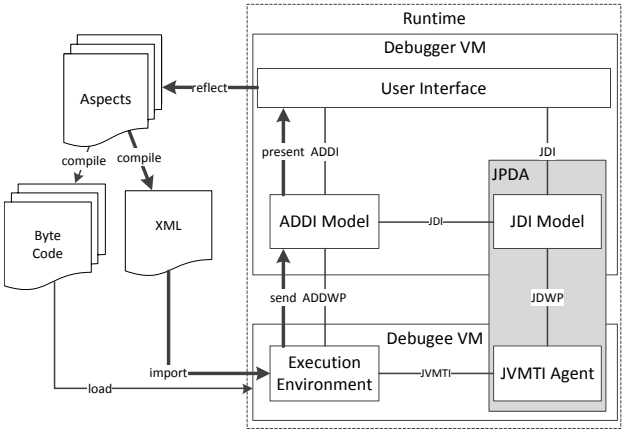

Figure 3. The architecture of our AOP debugger

4.1 Debuggee Side

In the ALIA4J approach, the execution environment is an extension to the Java Virtual Machine (JVM). The extension allows deploying and undeploying LIAM 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. [19]) 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) [8], 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 [7], 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.

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 2.3, which it implements in collaboration with the debuggee virtual machine. A simplified UML class diagram of ADDI is presented in figure 4.

The Java Debug Interface (JDI) provides mirrors for every runtime entity in a Java program, like objects, classes,

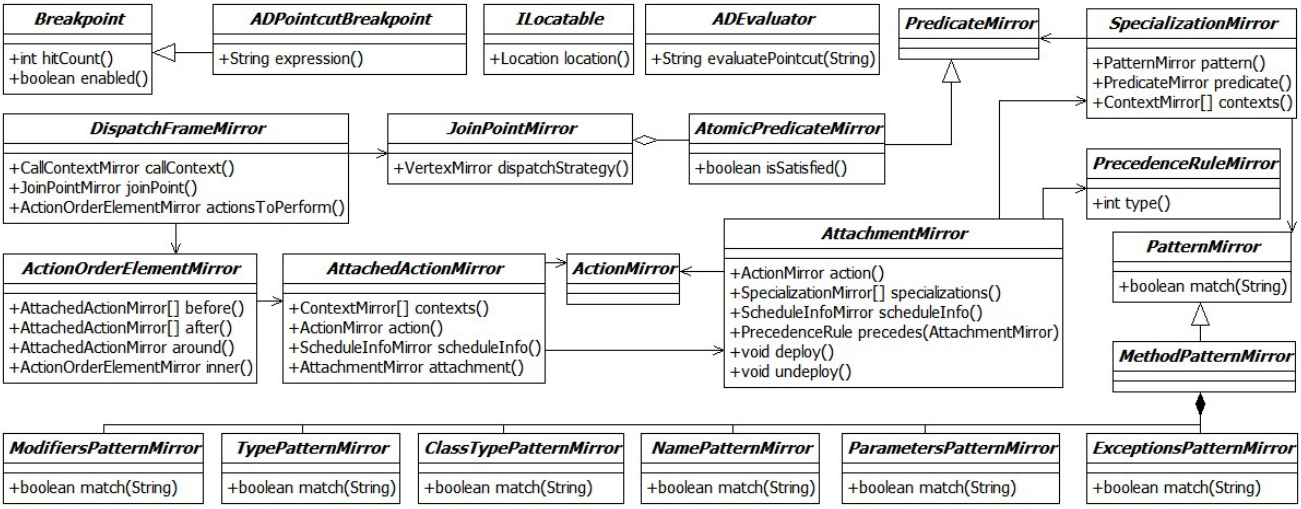

Figure 4. A simplified UML class diagram of the Advanced-Dispatching Debug Interface

or threads. The ADDI extends the JDI by additionally providing mirrors for the LIAM entities which exist in the debuggee virtual machine and which 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 existing Java breakpoints. 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:

ADPointcutBreakpoint provides the interface for setting breakpoint by utilizing pointcut expressions (**T1**).

ILocatable is an interface for locating entities. In our implementation of ADDI, subclasses of ActionMirror, AtomicPredicateMirror, PatternMirror, AttachmentMirror, DispatchFrameMirror, and PrecedenceRuleMirror implement this interface. Therefore, corresponding constructs can be located in the source code at runtime (**T2**).

ADEvaluator can perform evaluation on given pointcut expressions or sub-expressions (**T3**). It takes strings as input, sends them to the back-end. 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 performs evaluations to pointcut sub-expressions (**T3**).

ActionOrderElementMirror reifies the program composition (**T8**). It consists of four parts, namely *before*, *after*, *around*, and *inner*. The before, around, and after parts point to advice (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**).

PrecedenceRuleMirror represents ordering relations between attachments (**T9**). This includes precedence defined in AspectJ through the **declare precedence** statement, through the **before**, **after** or **around** keywords, and through the lexical order of advice definitions.

SpecializationMirror reifies static and dynamic sub-expressions of pointcuts which are decomposed into a pattern, a predicate, and contexts.[7] 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 4, patterns consist of smaller sub-patterns which are separate entities in ADDI and can be evaluated respectively (**T5**).

[7] See Bockisch et al. [5] for a detailed discussion of how to transform any AspectJ pointcut to our data structure.

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 AOP 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-advice in order to support the tasks discussed in section 2. The developed debugger provides three new views, namely the *Join Point* view, the *Attachments* view, and the *Pattern Evaluation* view.

Throughout this section, we illustrate the functionality of our debugger by means of the example AspectJ program shown in listing 3 and listing 4. There are four advices (on line 5, 8, 12, and 15, listing 4) declared in Azpect. Suppose the program is currently suspended at line 16 of listing 4. We introduce each view in this scenario in the following subsections.

```
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 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 4. An example aspect

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 5.

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 executed when the debuggee is intercepted. For each such join point, the signature and the source location of the corresponding join-point shadow are presented (**T7**).

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. In figure 5, it displays one composition with two sequential actions which are a field assignment and an advice execution. In AspectJ, advices do not have names. Therefore, we choose 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 6 shows the join point visualization for this case.

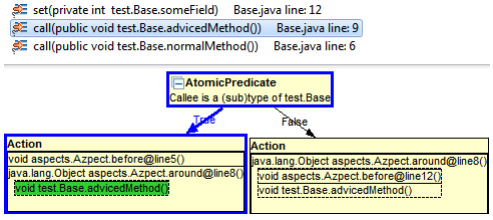

Figure 6. A graphical representation of dispatch

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 surrounding action may not execute *proceed* and thus may leave out the execution of this action. Double-clicking on a label representing an atomic predicate or an action reveals the corresponding source location (**T2**).

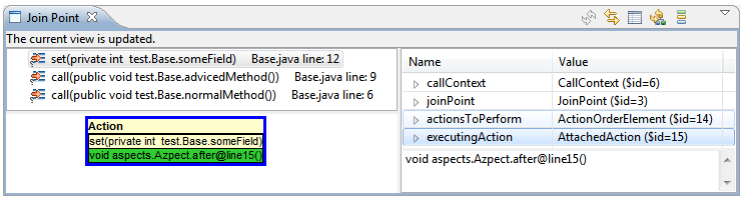

Figure 5. A snapshot of the Join Point view

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 in the following.

Textual Representation of Dispatch By clicking the "Table" button on the toolbar, the bottom left part is switched to a table, as shown in figure 7. 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 advice whose pointcut statically matches the join point.

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 7. A textual representation of dispatch

Second, for all actions whose pointcut dynamically matches the join point, the execution sequence as well as 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 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 three additional aspects which are shown in listing 5. Both aspects, PrecedingAzpect and PrecededAzpect, declare a **before** advice. The aspect IrrelevantAzpect defines the precedence between PrecedingAzpect and PrecededAzpect.

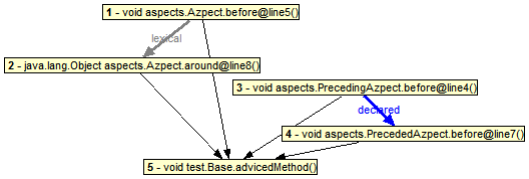

Figure 8. The graphical representation of precedence dependencies

```
1  package aspects;
2  import test.Base;
3  aspect PrecedingAzpect {
4    before() : call(* Base.advicedMethod()) { ... }
5  }
6  aspect PrecededAzpect {
7    before() : call(* Base.advicedMethod()) { ... }
8  }
9  aspect IrrelevantAzpect {
10   declare precedence : PrecedingAzpect, PrecededAzpect;
11 }
```

Listing 5. Aspect illustrating precedence dependencies

Consider that the execution is suspended at the call to advicedMethod() at line 9, listing 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 8.

Labels representing actions are numbered and connected by directed lines. The direction of a connection indicates the precedence between two actions. We use the numbers as substitute for action names in the following paragraph; for example, "action 1" represents Azpect.before@line5().

There are three types of connection representing precedence declared in different ways: 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"; or it may be determined by the kind of action (i.e., **before**, **after**, **around** advice or the join point action), visualized by a thin black connection without label.

Explicitly declared precedence has a source location, like line 10 in listing 5. The corresponding source location can be highlighted when the connection is double-clicked (**T2**). An example of precedence declaration by means of lexical order is shown in listing 4: Action 1 is declared on line 5 and,

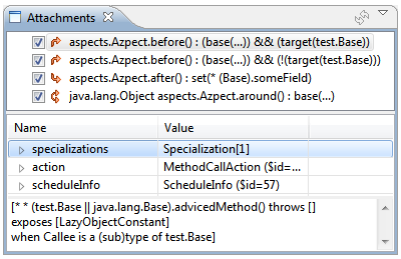

Figure 9. A snapshot of the Attachments view

thus, precedes action 2 defined on line 8. The precedence between any two actions without a connection is random, such as action 1 and action 3.

5.2 Attachments View

In order to dynamically deploy and undeploy attachments during runtime, the *Attachments* view is provided. A snapshot of the *Attachments* view is given in figure 9. The top panel shows a textual representation of all attachments that are defined in the executing program together with a checkbox indicating whether the attachment is currently deployed or not. Unchecking or checking one of the items will lead to undeployment or deployment of the corresponding attachment in the debugged program (**T10**). The bottom parts presents details of the selected attachment.

In figure 9, the first attachment, representing the *before* advice declared on line 5 in listing 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**).

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 5. We find that the **before** advice declared on line 5 in listing 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 10. It gives the evaluation results for each sub-pattern (**T5**).

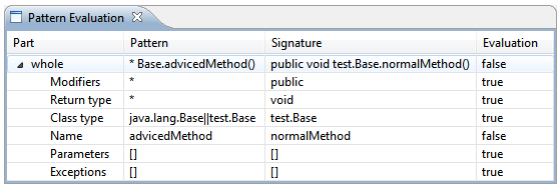

Figure 10. A snapshot of the Pattern Evaluation view

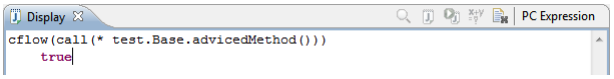

Figure 11. The extended Display view for evaluating pointcut expressions

5.4 Setting Pointcut Breakpoint and Evaluating Pointcut Expression

Except three newly added views, we also extend two existing views in Eclipse. We extend the *Breakpoints* view to allow specifying pointcut expressions in order to set pointcut breakpoints (**T1**).

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 extend the *Display* view. For example, suppose the second frame shown in figure 5 is selected, the programmer evaluates the expression **cflow**(**call**(∗ test.Base.advicedMethod())). The result is shown in figure 11.

6. Related Work

The related work basically falls into two 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 paper.

6.1 Debuggers for Aspect-Oriented Languages

We discussed the state-of-the-art AOP debuggers in section 2.3. The evaluation is summarized in table 3 showing that 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.

6.2 Development tools for aspect-oriented languages

AO-specific information provided by tools or systems are not only provided in online debuggers. Static tools can be

Tag	Task Name	Our debugger	JDB	AODA	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	Excluding and adding AO definitions	√			√	

Table 3. Comparison between different AOP debuggers from the perspective of supporting the identified tasks

used as auxiliary approaches to understand program behavior or structure during debugging.

Common IDE tools for AOP languages, like the AspectJ Development Tools (AJDT)[8], CaesarJ [2] Development Tools (CJDT)[9], JAsCo [20] Development Tools (JAsCoDT)[10] 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 [14]. 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 advice 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 [17] 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. Coelho and Murphy [9] implemented *ActiveAspect* which can present a subset of the crosscutting structure at the right time, thus decreasing the complexity of information to be analyzed. These systems aid language users to comprehend programs by simplifying the presentation of the crosscutting structure. Our debugger more concentrates on dynamic information, especially for pointcut and pattern evaluation, and program composition.

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

[9] See `http://caesarj.org`.

[10] See `http://ssel.vub.ac.be/jasco/index.html`.

7. Conclusions and Future Work

In this paper 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 ten 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. Therefore, we have based our prototype on our previous work which provides such an intermediate representation for languages with advanced-dispatching (AD) which is a generalization of aspect-oriented programming (AOP). We transform aspect-oriented (AO) information 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 AO abstractions. Based on the ADDI, we implemented a user interface in terms of three new and two extended Eclipse views.

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 de-

fined in the AOP program that lead to a specific composition.

2. It performs evaluations on pointcut and pattern subexpressions.
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 advice 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 deploying and undelopying them at runtime.

While our requirements for the debugger are based on a taxonomy of faults in aspect-oriented programs, we believe that our approach can be generalized to the concept of advanced dispatching. In our work we did not consciously applied any constraints specific to aspect-oriented programming languages. Therefore we expect that this work can be easily extended to support other advanced-dispatching programming languages supported by the ALIA4J architecture, like predicate dispatching or domain-specific languages.

Besides studying the applicability of our approach to other programming language paradigms, we will extend our user interface with advanced support for simplifying recurring tasks. Furthermore, we will investigate supporting debugging in ALIA4J's optimizing execution environments.

Acknowledgments

This work is partly funded by the Chinese Scholarship Council (CSC Scholarship No.2008613009).

References

[1] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical report, 2004.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. 3880:135–173, 2006.

[3] 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.

[4] 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.

[5] 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.

[6] C. Bockisch, S. Malakuti, M. Akşit, and S. Katz. Making aspects natural: events and composition. In *Proceedings of the 10th AOSD*, pages 285–300, New York, NY, USA, 2011. ACM.

[7] C. Bockisch, A. Sewe, M. Mezini, and M. Akşit. An overview of alia4j: an execution model for advanced-dispatching languages. In *Proceedings of the 49th TOOLS*, pages 131–146, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] S. Herrmann, C. Hundt, M. Mosconi, C. Pfeiffer, and J. Wloka. Das object teams development tooling. *Softwaretechnik-Trends*, 26(4):42–43, 2006.

[15] 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.

[16] F. M. Ceccato, P. Tonella. Is aop code easier to test than oop code? In *In Workshop on Testing Aspect-Oriented Programs*, 2005.

[17] 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.

[18] 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.

[19] 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.

[20] 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.