A Fine-Grained, Customizable Debugger for Aspect-Oriented Programming

Haihan Yin, Christoph Bockisch, and 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, many aspect-oriented programming languages provide 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 11 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. This is based on a model of the more general formalism of advanced dispatching. Based on this model, we implement a user interface with functionalities supporting the identified tasks, such as visualizing pointcut evaluation and program composition. Due to the generality of our intermediate representation, our debugger can be used for a wide range of programming languages. To account for the syntactic differences among these languages, we allow language designers to customize the textual representations on the user interface.

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

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. [18] 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 [25]. 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 (IR) of an established non-AO language; this usually entails transforming code already provided in that IR [5], 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 [16], the aspect-aware breakpoint model in AODA [15], or the identified AOP activities in TOD [26]. 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 paper, we introduce our concept and implementation of a dedicated debugger for AO programs which is able to support locating all types of

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

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.

This paper extends our AOSD'12 publication [31]; the sections 5.5—discussing a new breakpoint view—and 6—showing how to customize the visualization in our debugger to specific languages—are completely new. Besides, we now discuss in detail the risks involved with "excluding and adding AO definitions" and we have identified an additional task to cope with the risks (section 2.2); this additional task is also considered in our infrastructure (section 4.2) and our user interface (sections 5.1 and 5.2). We have extended the discussion of related work in sections 2.3 and 7 and have updated the introduction and conclusion according to our changes.

Section 2 describes how we generate requirements from existing AOP fault models. Section 3 introduces a dedicated advanced-dispatching meta model and how we improve the compilation process to preserve advanced-dispatching information. Sections 4 and 5 present the underlying debugging model and the user interface of our debugger. Section 6 shows how to extend our debugger to customize the visualization in favor of a specific language. Sections 7 and 8 list related works and conclude the paper respectively.

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.

2.1 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 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 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. [23] extend this model with three types concerning exceptional control flow and inter-type declarations (ITD). Ferrari et al. [19] 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 [6] which follows a similar approach; he focuses on AspectJ [22] 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 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[2], rerun the program, analyze program states, and eventually (**T2**) locate faulty constructs.

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.*

[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.

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

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**
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**

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.

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[3]. (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.

2.3 State-of-the-Art in Debugging AO Programs

Table 2 summarizes the required debugging tasks identified in the previous subsections 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.

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

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

Table 2. Tasks that an ideal AOP debugger should perform

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

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. [15] 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* [3] 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 [16] 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.

Pothier and Tanter [26] 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.

2.4 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 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 can be performed.
- The information presented to the developer in the user interface should have a representation specific to the concretely used AO language.

3 Debugging Information

We chose 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. This makes our debugger applicable to a wider range of programming languages than AOP. One of the main components of this *ALIA4J* architecture[4] [9] 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 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.

3.1 Advanced-Dispatching Intermediate Representation

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.

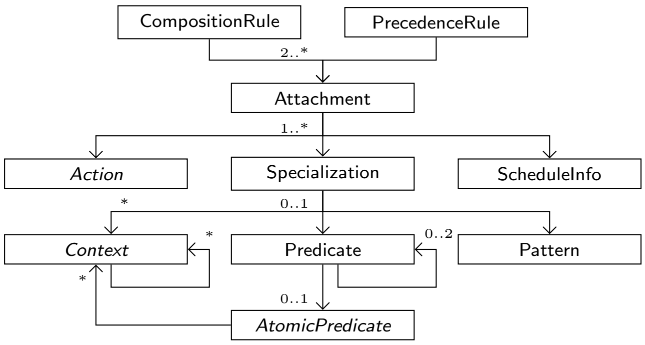


Fig. 1. The LIAM meta-model of advanced dispatching

Figure 1 shows the meta-entities of LIAM, discussed in detail by Bockisch et al. [8,10], which capture the core concepts underlying the various dispatch-

[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/`

ing mechanisms. The meta-entities *Action*, *AtomicPredicate*, and *Context* can be refined to concrete concepts; we provide refinements for several languages, including AspectJ [9].

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 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 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 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 paper, we just elaborate on our implementation of an AspectJ compiler based on the *abc* compiler [5]. 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 4–12, listing 2), as well as a value that is exposed to associated advices (represented by the *Context* in lines 13–15). The pointcut

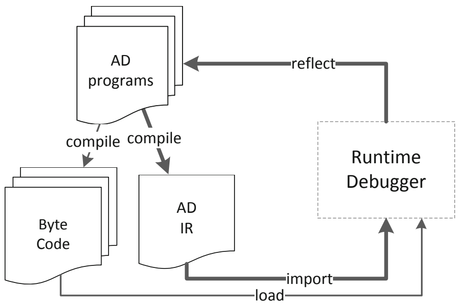


Fig. 2. Debugging information life cycle

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.

```
aspect Azpect {
    before(Base b) : call(∗ Base.foo()) && target(b) { ... }
}
```

Listing 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. 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 2.3. 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 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 6.

4 Infrastructure of Our Debugger

Extending figure 2, the overall structure of our debugger is presented in 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, 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.

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 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. [28]) 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) [12], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with

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

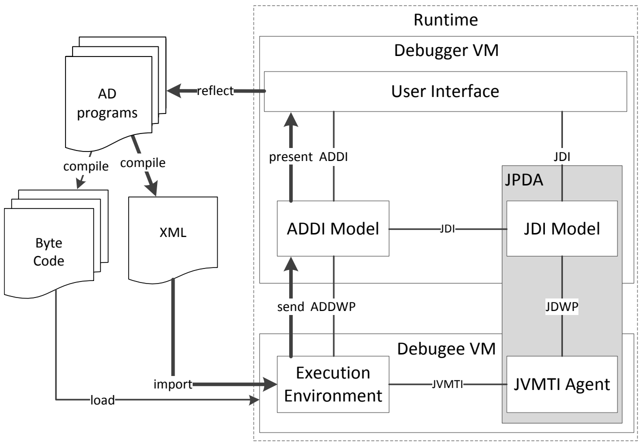


Fig. 3. The architecture of our AD debugger

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 [9], 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, and implements them 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, 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

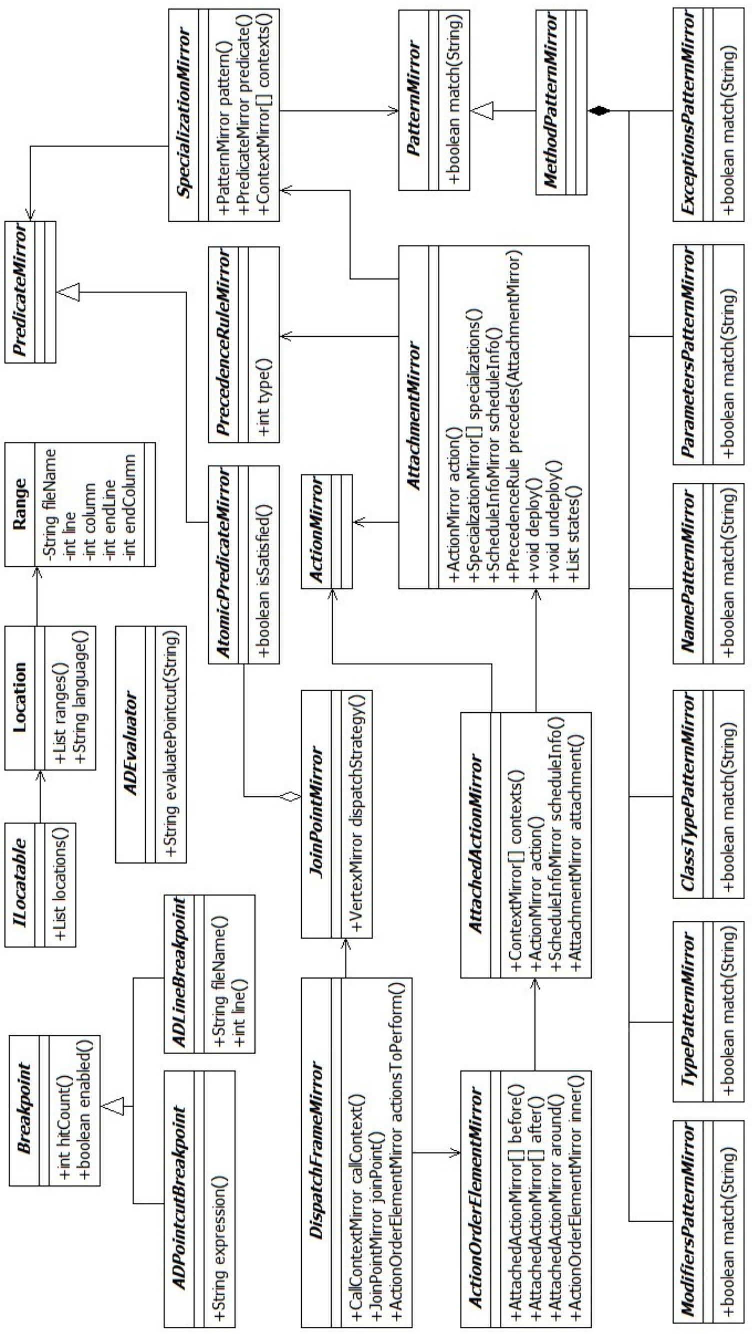


Fig. 4. A simplified UML class diagram of the Advanced-Dispatching Debug Interface

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 3.2). 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 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 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.[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**).

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. 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 3 shows the base program whose actions are advised by the aspect in 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 sub-sections.

```
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

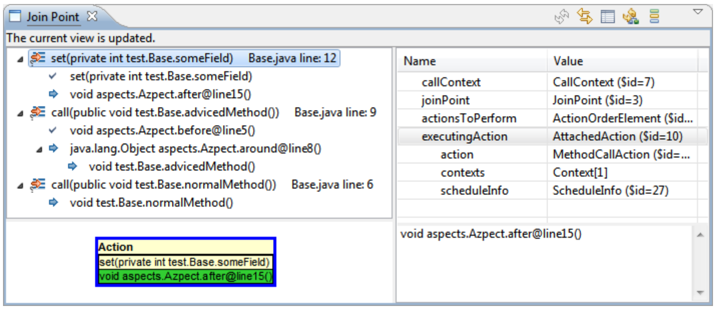
[7] See Bockisch et al. [7] for a detailed discussion of how to transform any AspectJ pointcut to our data structure.

```
 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.



The current view is updated.

```
◢ set(private int test.Base.someField)    Base.java line: 12
      ✓ set(private int test.Base.someField)
      ➡ void aspects.Azpect.after@line15()
◢ call(public void test.Base.advicedMethod())    Base.java line: 9
      ✓ void aspects.Azpect.before@line5()
    ◢ ➡ java.lang.Object aspects.Azpect.around@line8()
          ➡ void test.Base.advicedMethod()
◢ call(public void test.Base.normalMethod())    Base.java line: 6
      ➡ void test.Base.normalMethod()
```

Name	Value
callContext	CallContext ($id=7)
joinPoint	JoinPoint ($id=3)
actionsToPerform	ActionOrderElement ($id...
executingAction	AttachedAction ($id=10)
action	MethodCallAction ($id=...
contexts	Context[1]
scheduleInfo	ScheduleInfo ($id=27)

void aspects.Azpect.after@line15()

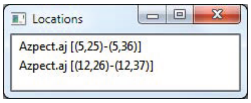
Action
set(private int test.Base.someField)
void aspects.Azpect.after@line15()

Fig. 5. 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 6. 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.



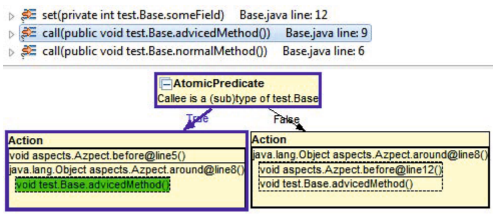
Azpect.aj [(5,25)-(5,36)]
Azpect.aj [(12,26)-(12,37)]

Fig. 6. A snapshot of the location window

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 5 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 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



```
set(private int test.Base.someField)   Base.java line: 12
call(public void test.Base.advicedMethod())   Base.java line: 9
call(public void test.Base.normalMethod())   Base.java line: 6
```

AtomicPredicate
Callee is a (sub)type of test.Base
True
False
Action
void aspects.Azpect.before@line5()
java.lang.Object aspects.Azpect.around@line8()
void test.Base.advicedMethod()
Action
java.lang.Object aspects.Azpect.around@line8()
void aspects.Azpect.before@line12()
void test.Base.advicedMethod()

Fig. 7. A graphical representation of dispatch

code. For illustration, consider that the second frame is selected in the example. Figure 7 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 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 8. Compared to figure 7, 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 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 9. 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

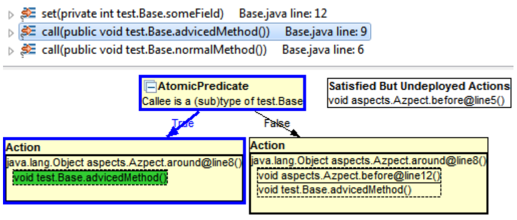
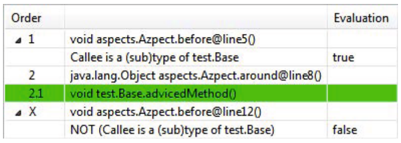

set(private int test.Base.someField) Base.java line: 12
call(public void test.Base.advicedMethod()) Base.java line: 9
call(public void test.Base.normalMethod()) Base.java line: 6
AtomicPredicate
Callee is a (sub)type of test.Base
Satisfied But Undeployed Actions
void aspects.Azpect.before@line5()
True
False
Action
java.lang.Object aspects.Azpect.around@line8()
void test.Base.advicedMethod()
Action
java.lang.Object aspects.Azpect.around@line8()
void aspects.Azpect.before@line12()
void test.Base.advicedMethod()

Fig. 8. A graphical representation of dispatch with a node showing "Satisfied But Undeployed 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

Fig. 9. A textual representation of dispatch

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 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 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 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 10.

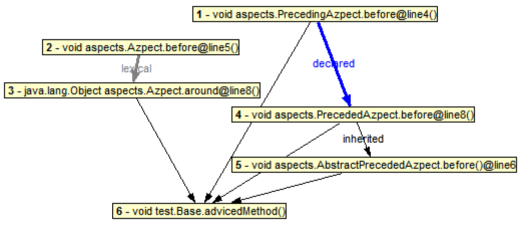

1 - void aspects.PrecedingAzpect.before@line4()
2 - void aspects.Azpect.before@line5()
lexical
declared
3 - java.lang.Object aspects.Azpect.around@line8()
4 - void aspects.PrecededAzpect.before@line8()
inherited
5 - void aspects.AbstractPrecededAzpect.before()@line6
6 - void test.Base.advicedMethod()

Fig. 10. The graphical representation of precedence dependencies

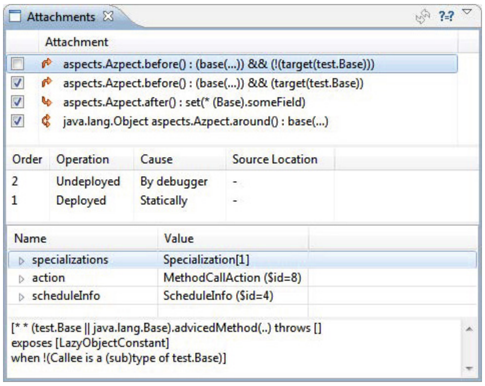
Labels representing actions are numbered and connected by directed lines. The direction of a connection indicates the precedence between two actions. We use the numbers as substitute for action names in the following paragraph; for example, "action 2" represents Azpect.before@line5().

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 5. For the "inherited" precedence, the **extends** clause is the source location, like line 7 in listing 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 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.

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 11. 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 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.



	Attachment
☐	aspects.Azpect.before() : (base(...)) && (!(target(test.Base)))
✓	aspects.Azpect.before() : (base(...)) && (target(test.Base))
✓	aspects.Azpect.after() : set(* (Base).someField)
✓	java.lang.Object aspects.Azpect.around() : base(...)

Order	Operation	Cause	Source Location
2	Undeployed	By debugger	-
1	Deployed	Statically	-

Name	Value
specializations	Specialization[1]
action	MethodCallAction ($id=8)
scheduleInfo	ScheduleInfo ($id=4)

```
[* * (test.Base || java.lang.Base).advicedMethod(..) throws []
exposes [LazyObjectConstant]
when !(Callee is a (sub)type of test.Base)]
```

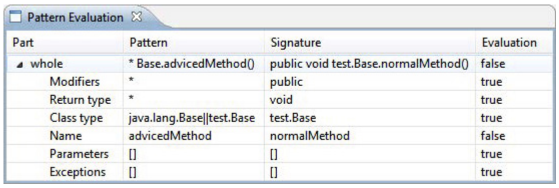
Fig. 11. A snapshot of the Attachments view

In figure 11, 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 12. It gives the evaluation results for each sub-pattern (**T5**).

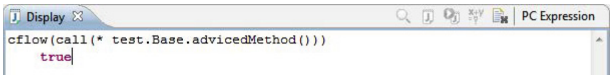


Part	Pattern	Signature	Evaluation
whole	* Base.advicedMethod()	public void test.Base.normalMethod()	false
Modifiers	*	public	true
Return type	*	void	true
Class type	java.lang.Base||test.Base	test.Base	true
Name	advicedMethod	normalMethod	false
Parameters	[]	[]	true
Exceptions	[]	[]	true

Fig. 12. A snapshot of the Pattern Evaluation view

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 5 is selected, the programmer evaluates the expression **cflow**(**call**($*$ test.Base.advicedMethod())). The result is shown in figure 13.



```
cflow(call(* test.Base.advicedMethod()))
    true
```

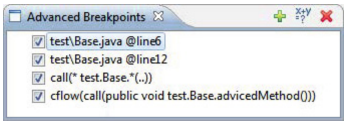
Fig. 13. The extended Display view for evaluating pointcut expressions

5.5 Advanced Breakpoints View

We added the *Advanced Breakpoints* view, as figure 14 shows, to allow setting breakpoint at pointcuts (**T1**). We currently provide two types of breakpoints which are line-based and expression-based. In figure 14, 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 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.



```
☑ test\Base.java @line6
☑ test\Base.java @line12
☑ call(* test.Base.*(..))
☑ cflow(call(public void test.Base.advicedMethod()))
```

Fig. 14. A snapshot of the Advanced Breakpoints view

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

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 3 lists four definitions with the same meaning, but written in different languages. This includes two

Table 3. Same dispatching restrictions expressed in different languages

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

AOP languages (AspectJ and JBoss AOP [1]), and two predicate-dispatching languages (MultiJava [13] and JPred [24]). 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 6.1), how to choose a customization at runtime (section 6.2), and how to construct customized descriptions (section 6.3).

6.1 Customizing the Presentation of an Entity in a Modular Way

Figure 15 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.

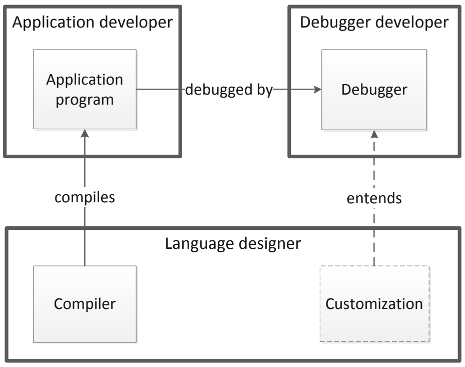


Fig. 15. Components which are used in debugging are developed by different parties

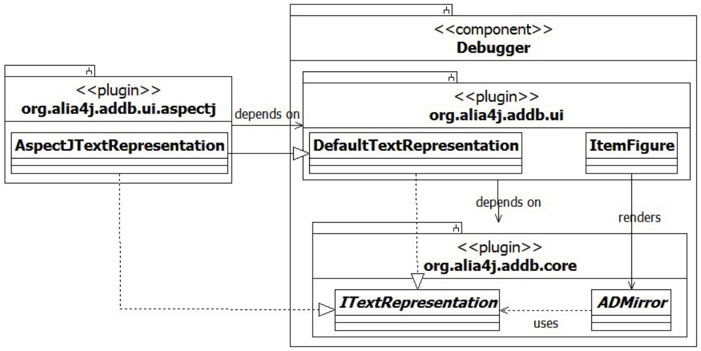


Fig. 16. The plug-in structure of our debugger. An extension point defines that a customization extension needs to realize the interface *ITextRepresentation*

.

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

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* [3]—which also identified debugging multi-language programs as a relevant problem—to illustrate this. We show this example program in listing 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 7. A multi-aspect-language example.

When the program in listing 7 is suspended at the execution of Stack.push(), AD actions declared at lines 5, 10, and 13 may be performed. To illustrate possible program compositions at this join point shadow, figure 17 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. 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.

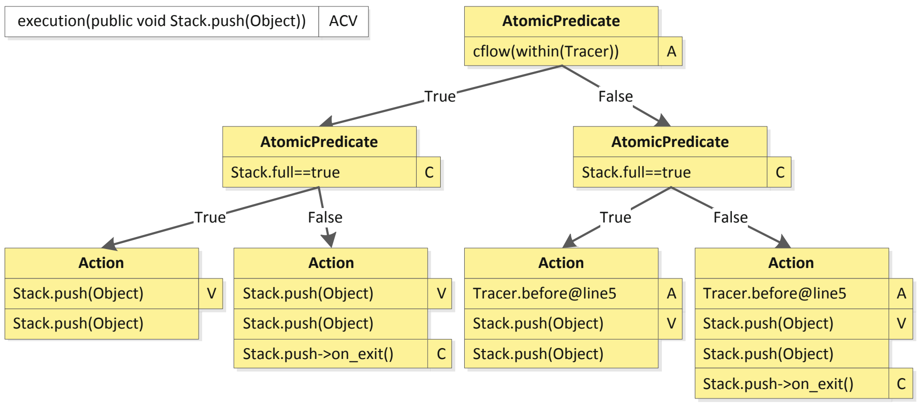


Fig. 17. Graphical representations tagged with language information

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 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 14: 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 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 [3]. 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.

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

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 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

Table 4. A comparison between different approaches applying customizations in a multi-AD-language example

	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

are available, the maximum number of configurations is the same as for the "local customization".

Table 4 summarizes the above discussion. Columns 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.

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 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 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(", ∗"); }
```

```
            }
            sb.append(")");
        } else if(context instanceof CalleeContextMirror) { ... }
        ...
        return sb.toString();
    }
}
```

Listing 8. The interface *ITextRepresentation* and its two implementations

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 paper.

7.1 Debuggers for Aspect-Oriented Languages

We discussed the state-of-the-art AOP debuggers in section 2.3 and the evaluation is summarized in table 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.

Table 5. Comparison between different AOP debuggers from the perspective of supporting the identified tasks. *Aws* stands for the AwesomeDebugger.

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	√				

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 (AJDT)[8], CaesarJ [4] Development Tools (CJDT)[9], JAsCo [29] 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 [21]. 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 [25] 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. [17] 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 [14] 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[11] 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

[8] See `http://www.eclipse.org/ajdt/`
[9] See `http://caesarj.org`
[10] See `http://ssel.vub.ac.be/jasco/index.html`
[11] See `http://sourceforge.net/projects/eclipse-plug130/`

our dispatch function decomposes each predicate into a set of atomic predicates. 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.

7.3 Tool Customizations

To provide highly tailorable user interfaces, many frameworks are created. For example, FlexiBeans [30] uses a component technology to provide a tool with run-time flexibility. Heer et. al. [20] found that existing information visualizations like treemaps [11] 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 [27] also argued that modern tools lack flexibility of customization and presented an approach implementing a flexible framework for visualizing code *structure*.

8 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 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 have based our prototype on our previous work which provides an intermediate representation for languages with advanced-dispatching which is a generalization of aspect-oriented programming mechanisms such as pointcut-advice or inter-type declarations. 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.

This work can already support other advanced-dispatching programming languages supported by the ALIA4J architecture, like predicate dispatching or domain-specific languages. However, we need to first identify dedicated fault-models for these paradigms to make out paradigm-specific debugging tasks as we did for aspect-oriented programming. Furthermore, we will investigate supporting debugging in ALIA4J's optimizing execution environments.

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

References

1. JBoss AOP (2008), `http://www.jboss.org/jbossaop`
2. Alexander, R.T., Bieman, J.M., Andrews, A.A.: Towards the Systematic Testing of Aspect-Oriented Programs. Technical report (2004)

3. Apter, Y., Lorenz, D.H., Mishali, O.: A Debug Interface for Debugging Multiple Domain Specific Aspect Languages. In: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development, AOSD 2012, pp. 47–58. ACM, New York (2012)
4. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
5. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An extensible AspectJ compiler. In: Proceedings of the 4th AOSD, pp. 87–98. ACM, New York (2005)
6. Baekken, J.S.: A Fault Model for Pointcuts and Advice in AspectJ Programs. Master's thesis, School of Electronical Engineering and Computer Science, Washington State University (2006)
7. Bockisch, C., Haupt, M., Mezini, M., Mitschke, R.: Envelope-Based Weaving for Faster Aspect Compilers. In: NODe/GSEM. LNI, vol. 69, pp. 3–18. GI (2005)
8. Bockisch, C., Malakuti, S., Akşit, M., Katz, S.: Making Aspects Natural: Events and Composition. In: Proceedings of the 10th AOSD, pp. 285–300. ACM, New York (2011)
9. Bockisch, C., Sewe, A., Mezini, M., Akşit, M.: An Overview of ALIA4J: An Execution Model for Advanced-Dispatching Languages. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 131–146. Springer, Heidelberg (2011)
10. Bockisch, C., Sewe, A., Yin, H., Mezini, M., Aksit, M.: An In-Depth Look at ALIA4J. Journal of Object Technology 11(1), 1–28 (2012)
11. Bruls, M., Huizing, K., van Wijk, J.: Squarified Treemaps. In: Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization, pp. 33–42. Press (1999)
12. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Comput. 35, 677–691 (1986)
13. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design Rationale, Compiler Implementation, and Applications. ACM Transactions on Programming Languages and Systems 28(3) (2006)
14. Coelho, W., Murphy, G.C.: Presenting Crosscutting Structure with Active Models. In: Proceedings of the 5th AOSD, pp. 158–168. ACM, New York (2006)
15. De Borger, W., Lagaisse, B., Joosen, W.: A Generic and Reflective Debugging Architecture to Support Runtime Visibility and Traceability of Aspects. In: Proceedings of the 8th AOSD, pp. 173–184. ACM, New York (2009)
16. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging Aspect-Enabled Programs. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 200–215. Springer, Heidelberg (2007)
17. Fabry, J., Kellens, A., Ducasse, S.: AspectMaps: A Scalable Visualization of Join Point Shadows. In: ICPC, pp. 121–130 (2011)
18. Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Figueiredo, E., Cacho, N., Lopes, F., Temudo, N., Silva, L., Soares, S., Rashid, A., Masiero, P., Batista, T., Maldonado, J.: An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs. In: Proceedings of the 32nd ICSE, vol. 1, pp. 65–74. ACM, New York (2010)
19. Ferrari, F.C., Maldonado, J.C., Rashid, A.: Mutation Testing for Aspect-Oriented Programs. In: Proceedings of the 2008 ICST, pp. 52–61. IEEE Computer Society, Washington, DC (2008)
20. Heer, J., Card, S.K., Landay, J.A.: prefuse: a toolkit for interactive information visualization. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2005, pp. 421–430. ACM, New York (2005)

21. Herrmann, S., Hundt, C., Mosconi, M., Pfeiffer, C., Wloka, J.: Das Object Teams Development Tooling. Softwaretechnik-Trends 26(4), 42–43 (2006)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. Ceccato, F.M., Tonella, P.: Is AOP code easier to test than OOP code? In: Workshop on Testing Aspect-Oriented Programs (2005)
24. Millstein, T., Frost, C., Ryder, J., Warth, A.: Expressive and Modular Predicate Dispatch for Java. ACM Trans. Program. Lang. Syst. 31(2), 7:1–7:54 (2009)
25. Pfeiffer, J., Gurd, J.R.: Visualisation-Based Tool Support for the Development of Aspect-Oriented Programs. In: Proceedings of the 5th AOSD, pp. 146–157. ACM, New York (2006)
26. Pothier, G., Tanter, E.: Extending Omniscient Debugging to Support Aspect-Oriented Programming. In: Proceedings of SAC, pp. 266–270. ACM, New York (2008)
27. Schafer, T., Mezini, M.: Towards More Flexibility in Software Visualization Tools. In: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, p. 20. IEEE Computer Society, Washington, DC (2005)
28. Sewe, A., Bockisch, C., Mezini, M.: Redundancy-Free Residual Dispatch: Using Ordered Binary Decision Diagrams for Efficient Dispatch. In: Proceedings of the 7th Workshop on FOAL, pp. 1–7. ACM, New York (2008)
29. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In: Proceedings of the 2nd AOSD, pp. 21–29. ACM, New York (2003)
30. Wulf, V., Pipek, V., Won, M.: Component-based tailorability: Enabling highly flexible software applications. Int. J. Hum.-Comput. Stud. 66(1), 1–22 (2008)
31. Yin, H., Bockisch, C., Aksit, M.: A Fine-Grained Debugger for Aspect-Oriented Programming. In: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development, AOSD 2012, pp. 59–70. ACM, New York (2012)