Developing A Generic Debugger for Advanced-Dispatching Languages

Thesis by

Haihan Yin

Supervisor: Dr.Ing. Christoph Bockisch

Software Engineering Group

University of Twente

August 2010

Abstract

Programming-language research has introduced a considerable number of advanced-dispatching mechanisms in order to improve modularity. Advanced-dispatching mechanisms allow changing the behavior of a function without modifying their call site and thus make the local behavior of code less comprehensible. Debuggers are tools, thus needed, which can help a developer to comprehend program behavior but current debuggers do not provide inspection of advanced-dispatching-related language constructs. This thesis presents a debugger which extends a traditional Java debugger with the ability of debugging on an extended debugger architecture that preserves advanced-dispatching language constructs and a user interface for inspecting this.

Acknowledgements

First and foremost, my deepest gratitude goes to my thesis supervisors, Dr. Ing. Christoph Bockisch. Without his constant encouragement, guidance and support, the completion of this thesis would never be possible.

I must thank my parents in China since they always encourage me, especially when I feel frustrated and lonely. I am also indebted to my uncle, Dr. HUANG Zhisheng. His tuition has ever been the fount of inspiration to me. I also would like to thank my aunt and my cousin for their help and support since I came to the Netherlands. Moreover, I would like to thank Mr. QIN Ning for his advises on writing.

Last but not least, I deeply thank everyone who helps me for my thesis.

At last, any errors within this thesis I accept as my own.

Contents

Chapter 1

Introduction

To improve the modularity of source code, a considerable number of new programming-language mechanisms has been developed that are based on manipulating dispatch, e.g., of method calls. Examples are multiple [9] and predicate dispatching [14] or pointcut-advice [18], a particular flavor of aspect-oriented programming (AOP). Because they are beyond the traditional *receiver-type polymorphism* dispatch mechanism, these new mechanisms are called *advanced-dispatching* (AD) [7].

The majority of newly developed languages adds advanced-dispatching concepts to an already existing, mainstream language like Java or the .NET languages, which is generally called the *base language*. The new mechanisms have the potential to increase the modularity of program code, compared to code written in the base language. However, using the new language mechanisms is not well-supported by tools. Thus their usage may hamper software development even in spite of their potential of improving the code quality.

The term *dispatching* refers to binding functionality to the execution of certain instructions, so-called *dispatch sites*, at runtime, thereby choosing from different alternatives that are applicable in different states of the program execution. An example of conventional dispatch is the invocation of a virtual method in Object-Oriented (OO) languages: The invocation is the dispatch site and the alternative functionalities are the different implementations of the method in the type hierarchy; the runtime state on which the dispatch depends is the dynamic type of the receiver object. A detailed discussion of the approach can be found in [7][1]

[1] Some details presented in [7] are outdated, but it may nevertheless act as an introduction to the basic concepts.

Advanced-dispatching refers to language mechanisms that go beyond this traditional receiver-type polymorphism. What makes the dispatching advanced in these cases is that a dispatch can consider additional and more complex runtime states, and that functionality can be composed in various ways.

Advanced-dispatching languages share concepts from several broad categories: selection of call site based on syntactic properties, access to the runtime state in which they are executed, evaluation of functions over the runtime state to select from alternative meanings, declaration of meaning in terms of actions on the runtime state, and description of relationships between applicable actions. In order to let advanced-dispatching languages share their implementations, Bockisch et al. have provided Advanced-dispatching Language-Implementation Architecture (ALIA). ALIA consists of a language-independent meta-model of advanced-dispatching concepts and any number of execution environment that process models conforming to this single meta-model. For those advanced-dispatching languages whose base language is Java, ALIA is implemented as ALIA4J.

While high-quality tools, such as the AspectJ Development Tools (AJDT) [3], exist to visualize the static structure of programs written in these languages, little to no support is provided related to dynamic language features. As lack of supporting tools significantly hampers the popularity of otherwise valuable new programming languages, this thesis aims at providing a generic debugger for advanced-dispatching languages.

The term "bug" and "debug" was first used buy Admiral Grace Hopper in 1940 because she found a moth stuck in a circuit. Now the "bug" is widely accepted as "an unexpected behavior of a system". The difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming languages used and the available tools, such as debuggers. Debuggers are software tools that help determine why the program does not behave correctly. They assist developers to understand the program and find the cause and location of the bug. They also enable programmers to control the program execution flow, stop the program at any desired point and inspect and analyse the runtime states. As the software architecture and programming languages become more complex and implementation is inherently an error-prone process, debugging has increasingly significant impact on the cost of software [20].

Eaddy et al. [13] proposed a debug model for Aspect-Oriented Programming (AOP). As AO languages is one category of AD language, their debug model can be extended and adapted to the domain of AD programming. The adapted debug model consist of three components: a classification of AD-specific activities, faults introduced in these activities and a set of ideal properties that an AD debugger should support. These properties are

idempotence, debug obliviousness, debug intimacy, dynamism, dispatching-declaration introduction, locating and affection exploration.

My development is carried out on the execution environment called NOIRIn. NOIRIn provides all desired AD-related information during runtime. In order to present this information, an AD debugger architecture is designed based on the Java Platform Debug Architecture (JPDA) [2]. The AD debugger architecture associates with two Java Virtual Machine (JVM): the debuggee JVM and the debugger JVM. NOIRIn runs on the debuggee JVM and provides required information in terms of Java constructs. This information is read by the debugger side through a communication channel and then adapted to dedicated AD constructs. AD constructs form an underlying model represented by the Advanced-Dispatching Debug Interface (ADDI). Through ADDI, user interfaces for inspecting and manipulating AD debug information are implemented.

For evaluating properties desired from AD debuggers, this thesis gives detailed discussions like to what extent it is supported and how, where are the limitations and why, etc. The result is that four properties are fully supported, two are partially supported and one is not supported.

The remainder of this thesis is structured as follows.

- Chapter 2 presents the required technical background.
- Chapter 3 describes problems of debugging AD programs with conventional debuggers and proposes a dedicated AD debug model.
- Chapter 4 gives a deeper analysis of problems when debugging AD programs as well as properties required from a good debugger for such programs.
- Chapter 5 describes improvements to the NOIRIn and implementation details of the AD debugger.
- Chapter 6 shows an example using the AD debugger to find a bug and discusses how well each required property is met in the implementation.
- Chapter 7 designs a solution for supporting locating AD-specific constructs.
- Chapter 8 investigates some AD tools and systems which include Wicca [13], AspectJ Development Tools (AJDT) [3], Aspect Oriented Debugging Architecture (AODA) [11], TOD [22], CaesarJ Development Tools (CJDT) [4], JAsCo Development Tools (JAsCoDT) [1].

- Chapter 9 summarizes this thesis and gives directions for future work.

Chapter 2

Background

2.1 Characteristics of Advanced-dispatching Languages

Advanced-dispatching (AD) languages can be classified into two categories: predicate dispatching languages, like MultiJava [10], JPred [19], and pointcut-advice languages, like AspectJ [17], Compose* [12]. In these languages, function calls are late-bound to meaning and which meaning is ultimately executed upon a call is determined at runtime. AD mechanisms allow changing the behavior of a function without modifying the call site and thus the code's modularity increases.

The program in listing 2.1 is written in JPred. A method uses the keyword *when* with an appended predicate specifying what conditions should be satisfied to execute that method. Multiple methods may share the same signature and which method is eventually executed depends on the boolean result of the predicate at runtime. At each call site, only one method with the most specific predicate is selected. In this example, the method on line 2 is only applicable to an invocation of FileEditor.handle(Event) if the class of actual parameter passed to the method is a subclass of *Open*. In any other circumstances, the method on line 3 is executed.

```
1 class FileEditor {
2   void handle(Event e) when e@Open {... /* open a file */}
3   void handle(Event e) {... /* handle unexpected event */}
4 }
```

Listing 2.1: A code example in JPred syntax

The program in listing 2.2 is written in AspectJ which is the most popular AO language. In AspectJ, a pointcut can specify source locations where crosscutting actions take place.

These locations are called *join point shadows*. A pointcut may also specify a dynamic-evaluated condition, called *residue*. An advice is executed when its pointcut is satisfied. In this example, line 3 specifies a join point shadow **void** Point.set∗(..) which matches all public methods in the Point class with a name starting with **set** and taking any number and type of arguments. Line 2 and 4 specify that the callee object should be an instance of *Point*. This advice is invoked *before* the execution of method represented by **void** Point.set∗(..).

```
1 public aspect PointMonitor {
2   before(Point e) :
3     call(public void Point.set*(..)) &&
4     target(p)
5       { ... /* notification */ }
6 }
```

Listing 2.2: A code example in AspectJ syntax

2.2 ALIA Overview

With the *Advanced-dispatching Language-Implementation Architecture for Java* (ALIA4J), Bockisch et al. have provided an architecture for implementing programming languages with advanced dispatching concepts in a way that they can share the implementation of overlapping concepts [6]. They have found that advanced dispatching is the basis of many different programming paradigms (like predicate dispatching, pointcut-advice, inter-type member declarations, or security policies) which are supported by ALIA4J.

ALIA4J has two main components: The first is the *Language-Independent Advanced-dispatching Meta-model* (LIAM) for expressing advanced-dispatching declarations, the second one is the *Framework for Implementing Advanced-dispatching Languages* (FIAL), a framework for execution environments that can process these dispatch declarations. A brief overview, of the approach can be found in [7].

LIAM acts as the *format* of the intermediate representation for advanced dispatching in programs. The meta-model itself defines *categories* of concepts and how these concepts can interact, e.g., a dispatch may be ruled by *atomic predicates* which depends values in the dynamic *context* of the dispatch. It has to be *refined* with the concrete advanced-dispatching concepts of actual programming languages that are supposed to be mapped to LIAM. For example consider the AspectJ pointcut designator **target**(T), which represents an atomic predicate, namely that some context value must satisfy the expression **instanceof** T, and this

value is the receiver object. Thus, amongst others, the LIAM concept *atomic predicate* must be refined to an *instance of predicate* and the concept *context* must be refined to *callee object* when realizing AspectJ with ALIA4J.

The actual intermediate representation of a concrete (e.g., AspectJ) program, in turn, is a model conforming the meta-model refinement for that language—these models are called *LIAM models* simply. Code of the program *not* using advanced dispatching mechanisms is represented in its conventional Java bytecode form. This is also true for the body of advice written in the base language as is the case in, e.g., AspectJ. In this case, the LIAM model will contain a *action* entity that stores a symbolic reference to the method into which the advice body is compiled; execution the functionality defined by this action means to execute the referred method. The situation is different in languages with a domain-specific "advice" model like Compose*. Advice, actually called "filter actions" in Compose*, can be declarative like "raise an error"; such functionality can also be realized by refining the LIAM meta-entity *action*.

FIAL implements common components and work flows required to implement execution environments based on a JVM for executing LIAM models, most importantly it defines how to derive an dispatch function *per dispatch site* that considers all dispatch declarations present in the program.

Figure 2.1 shows an overview of the ALIA4J approach. Concretely, the flow of compiling and executing applications in this approach is shown. The compiler ❶ starts processing the source code; a dedicated importer component ❷ adapts the compiler's output to a model for the advanced dispatch declarations in the program ❸ based on the refined subclasses ❹ of the LIAM meta-entities ❺. Furthermore, the compiler produces an intermediate representation of those parts of the program that are expressible in the base language ❻ alone.

The eight meta-entities of LIAM capture the core concepts underlying the various dispatching mechanisms, but at a finer granularity than the concrete concepts found in high-level languages; one concrete concept often maps to a combination of LIAM's core concepts. Figure 2.2 shows the meta-entities in LIAM, which are implemented as abstract classes. Attachment, specialization, and predicate are an exception to this rule, i.e., they are concrete classes, as they provide logical groupings of entities of the meta-model and cannot be refined. The meta-entities are discussed in detail in [6, Chapter 3.2][1].

In short, an attachment corresponds to a unit of dispatch description. In terms of aspect-orientation (AO), this roughly corresponds to a pointcut-advice pair, in terms of predicate

[1]There, some meta-entities are named differently, but the structure of the meta-entities is the same. Therefore, the interested reader will be able to map the discussion to the new names.

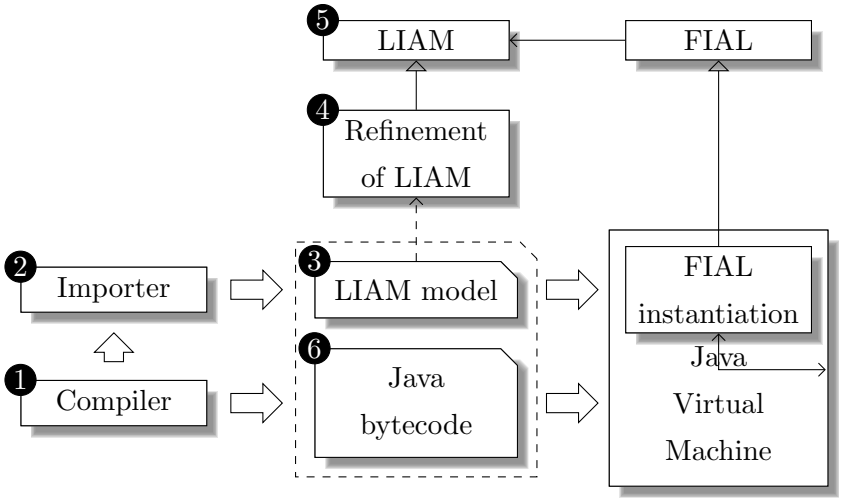

Figure 2.1: Overview of the application life cycle in ALIA4J-based language implementations.

dispatching to a predicate method. It should be noted that in some advanced-dispatching languages the dispatch declaration is not fully localized. For instance the pointcut-advice definition in AspectJ only is complete together with the instantiation strategy declared for the containing aspect: In the AspectJ language, aspects are types, similar to classes, and instances of these types can exist; an advice is comparable to an instance method that executes in the context of such an aspect instance. In the header of an aspect, a strategy can be declared for retrieving the aspect instance necessary to execute an advice at a join point, examples are **issingleton**() or **pertarget**. This strategy is defined once per aspect in AspectJ. But in LIAM models, this definition is part of each attachment.

Action specifies an action to which the dispatch may lead (e.g., an advice or the predicate-method body). Specialization defines static and dynamic properties of a dispatch result to which an action should be attached: patterns specify syntactic properties of call sites which are affected by the declared dispatch; predicate and atomic predicate entities model dynamic properties a dispatch depends on (dynamic pointcut designators in AO terminology). Context entities model access to values in the context of a dispatch, like the calling object or argument values. Finally, the schedule information models constraints between multiple actions applicable at the same generic-function call. This includes the order of their execution, as well as relations like mutual exclusion.

At runtime, FIAL derives a dispatch model for each dispatch site in the program from all attachments that have been defined. Thereby, FIAL solves the constraints specified as

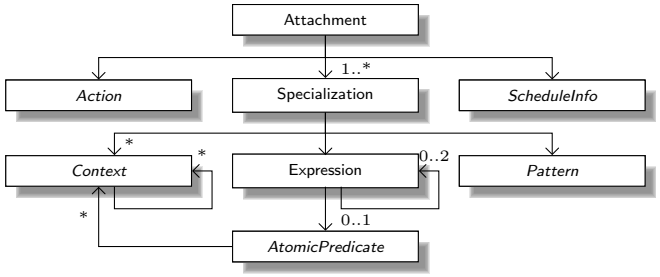

Figure 2.2: Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

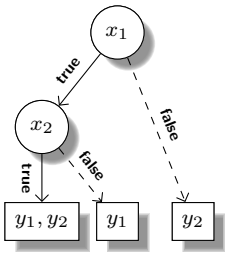

Figure 2.3: A dispatch function's evaluation strategy.

schedule information and derives a single dispatch function per call site from the predicates of all specializations. This function is represented as a binary decision diagram (BDD) [8], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with the actions to be executed. For each possible result of dispatch, the BDD has one leaf node, representing an alternative result of the dispatch. Figure 2.3 shows an example of such a dispatch model with the atomic predicates x_1 and x_2 and the actions y_1 and y_2. The reference [24] provides a detailed explanation of this model.

2.2.1 NOIRIn

FIAL, as the framework of execution environments, has multiple instantiations with different code generation strategies. For example, the Envelope-based Reference Implementation (ERIn) weaves code at the Java bytecode level, $STEAMLOOM^{ALIA}$ manipulates machine code and NOIRIn, which is used in the development of the AD debugger, does not perform code weaving.

NOIRIn performs dispatching by collecting runtime values and invokes actions reflectively according to information stored in LIAM models. Though NOIRIn is less complex than the weaving execution environments, it can provide all desired information about

dispatching. Figure 2.4 presents the workflow of deploying an advanced-dispatching declaration and of executing advanced dispatch in NOIRIn. The following list describes each step in detail.

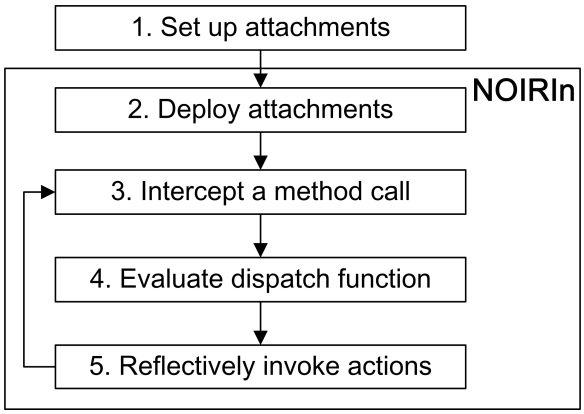

Figure 2.4: The workflow of performing dispatching in NOIRIn

1. A dedicated importer component adapts the complied source code of an AD declaration program, like an AspectJ program, to a set of attachment models conforming LIAM and sends these attachments into NOIRIn.

2. An attachment does not take effect until it is deployed. During the process of deployment, NOIRIn extracts the *Pattern* from the attachment in order to find out all matched dispatch sites. Then the new attachment is combined with existing dispatch function of each dispatch site.

3. After the preparation of the previous two steps, the actual application program starts. When a dispatch site is encountered, e.g., a method is invoked in the scope of user written code, NOIRIn intercepts this call and collects the call context, like the line number in which this call happens, the declaring class of the callee, etc. According to the call context, NOIRIn finds out the corresponding dispatch function.

4. As stated previously, the dispatch function is represented as a BDD. This dispatch function is evaluated from the root to a leaf according to the call context. When the evaluation reaches a leaf, it is determined which actions are actually to be performed.

5. For the applicable actions, an order is determined by resolving the constraints of the associated *Schedule Information*. Finally, NOIRIn executes all applicable actions in

the determined order. After all actions have been executed, NOIRIn reads the next dispatch, starting over at step 3.

2.3 The Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) [2] is presented in the figure 2.5. It defines a system consisting of two relatively independent layers and a communication channel between them. From the bottom up, the three components are the Java Virtual Machine Tool Interface (JVMTI), the Java Debug Wire Protocol (JDWP) and the Java Debug Interface (JDI). The JVMTI is a native interface of the JVM. It allows to check runtime states of a program, set call-back functions and control some environment variables. The JDWP is the protocol used for the communication between the debugger JVM and the debuggee JVM.

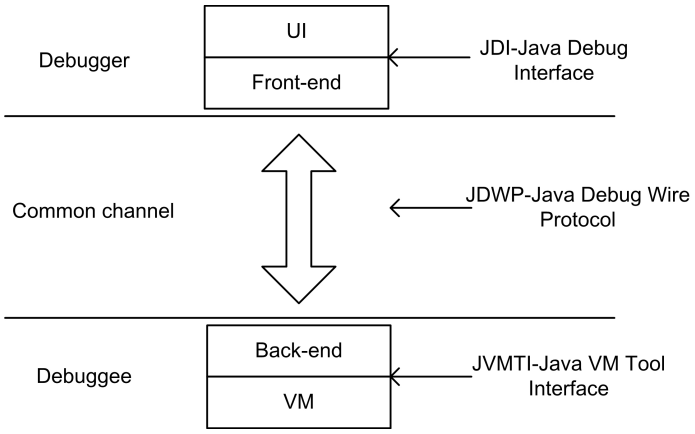

Figure 2.5: The architecture of Java Platform Debugger Architecture

The JDI is the highest layer in JPDA and it defines debugging interfaces needed by the debugger and converts requests to JDWP. Based on these interfaces, the debugger can know runtime states of the debuggee JVM, e.g., which classes and objects exist in JVM. Besides, the debugger can control the execution of the debuggee JVM, e.g., suspend and resume threads, set breakpoints, etc. According to different responsibilities, the JDI can be further divided into three modules: link module, event and handling module, data module. The link module sets up the communication channel between the debugger side and debuggee side. The event and handling module provides the interacting ways between the two sides. The data module is responsible for modeling data on both sides. In this thesis, an extended

debugging model is proposed based on the data module. The following paragraphs introduce the data module in more detail.

The JDI is also called a mirror-base, reflective interface. Almost all other interfaces inherit the interface *Mirror*. The mirror mechanism maps all entities including value, type, field, method, event, state and resources on the debuggee JVM into mirror objects. For instance, loaded classes are mapped to *ReferenceType*, objects are mapped to *ObjectReference*, primitive values are mapped to *PrimitiveValue*, all debugging related events are mapped to *Event*, requests sent by the debugger are mapped to *EventRequest*, the debuggee JVM is mapped to *VirtualMachine*, etc. Accessing fields or invoking methods of an object existing in the debuggee JVM can be performed by the debugger in a reflective way, using the mirrors. The following figure gives a simplified UML class diagram of the JDI model.

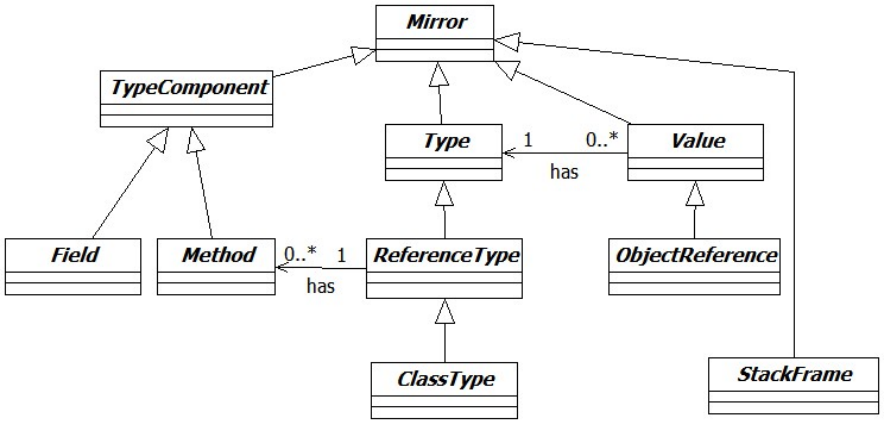

Figure 2.6: UML class diagram of a simplified Java Debug Interface model

Mirror reifies a proxy used by a debugger to examine or manipulate some entity in debuggee JVM. All most all resources on debuggee JVM can be mapped to mirrors. Mirrors can access the virtual machine (virtualMachine()) where their counterpart is defined.

StackFrame mirrors a frame from a debuggee JVM at some point in its thread's execution. The lifetime of a StackFrame is very limited. It is available only for suspended threads and becomes invalid once its thread is resumed. StackFrames provide access to a method's local variables (visibleVariables()) and their current values (getValue(LocalVariable)).

Location reifies a point within the executing code of the debuggee JVM. Locations are used to identify the current position of a suspended thread. They are also used to identify the position at which to set a breakpoint. Locations have a declaring class (declaringType()) and a line number (lineNumber()).

TypeComponent reifies an entity declared within a user defined type which can be a class or an interface. TypeComponent is the super class of Field and Method. Each TypeComponent has a declaring type (declaringType()) and a name (name()).

Field reifies a field declared in a type. Each Field has its own type.

Method reifies a static or instance method in the debuggee JVM. Each method has a return type (returnType()), a list of argument types (argumentTypes()) and a location where it is declared (location()).

Type reifies a type. A type can be primitive, void or a ReferenceType. Types have a name (name()) and a signature (signature()).

ReferenceType can be ClassType, InterfaceType or ArrayType. ReferenceTypes can be obtained by querying a particular ObjectReference for its type. ReferenceType provides access to static type information such as methods (allMethods()) and fields (allFields()).

ClassType reifies a class. ClassType extends Type. A ClassType has Fields, Methods, Constructors, Interfaces, a superclass, subclasses, etc.

Value reifies a typed entity. A value can be primitive, void or an ObjectReference. Each value has a type (type()).

ObjectReference reifies an object that currently exists in the debuggee JVM. An ObjectReference can invoke any method (invokeMethod()) declared in its type.

2.4 The Eclipse Debugger

Eclipse provides a language-independent debug model (called the platform debug model) which defines generic debugging interfaces that are intended to be implemented and extended by language-specific implementations. The Eclipse debugger has a mirror of each relevant runtime value in the execution of the debugged program. The hierarchy of debugged artifacts is shown in figure 2.7.

The *DebugTarget* is a debuggable execution context, in the case of Java a virtual machine, and it contains several *Thread*s which again contain *StackFrame*s. The *StackFrame* is an execution context in a suspended thread and contains *Variable*s. *Variable* has a *Value*; values can be objects with fields, which are also modeled as *Variable*. The model also defines interfaces for sending requests, like "Resume" to the debug model elements, and interfaces for handling events, like reaching a breakpoint.

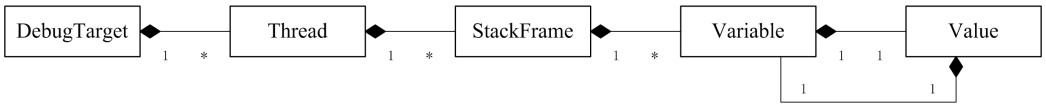

Figure 2.7: Eclipse Platform Debug Model

In a typical debugging workflow the developer first locates a line in the source code where he/she assumes an error, e.g., because an exception is thrown at this line, and sets a breakpoint on that source code line. When the program is started the next time, the debugger connects to the *DebugTarget* representing its execution and sends a request to activate the breakpoint. When the execution reaches the code corresponding to the breakpoint's line, the execution thread is suspended, and an event is sent to the debugger virtual machine, notifying it that the *Thread* is intercepted at a certain *StackFrame*. The debugger can now present this information to the developer, e.g., by highlighting the source code line that corresponds to the *StackFrame* and by presenting which *Variables* are present at the stack frame. The programmer can inspect the runtime values of these variables which require some more interaction between the debugger and the debuggee. To control the further execution of the suspended program, the developer also has different options which usually include:

Step Into means execution proceeds into any function in the current source statement and stops at the first executable source line in that function.

Step Over means execution proceeds past any function calls to the textually succeeding source line in the source scope. "Step Into" and "Step Over" require *internal breakpoints* which are auxiliary breakpoints set by debugger in order to mark some special places, like method entrance and method exit.

Resume continues execution.

Terminate ceases execution.

Chapter 3

Motivation

3.1 Problems of debugging Advanced-dispatching programs

From the two examples introduced in section 2.1, we observe that the actual behavior of an AD program is determined at runtime. Especially in AO languages, the behavior of a given piece of code can be altered to an arbitrary degree by an aspect in another source code file. This feature makes AD programs notoriously complex and thus comprehensibility is decreased.

A debugger is a tool helping a developer to understand program behavior. For those AD languages whose base language is Java, their development tools use the conventional Java debugger because programs written by them can be compiled to pure Java bytecode and can run on a standard JVM. Take debugging an AspectJ program in Eclipse for example, the inconsistence between language and debugger brings many problems which include the following ones.

First, the Java debugger can show which action is executed, however, it is unable to present the reasoning behind choosing this action. Besides, the debugger does not provide aspect-related information like runtime states accessed during dispatch, e.g., in terms of dynamic pointcut designators, when the program is suspended at a certain join-point shadow.

Second, the debugger uses code generated by an infrastructure so that developers may easily get confused because what is shown is absent in user-written code. For example, a stack frame corresponding to an advice uses the name generated by the compiler. Listing 3.1 gives an example of a base program and an aspect is presented in listing 3.2. When `p.move()` on line 4 of listing 3.1 is called, the *before* advice in listing 3.2 will be executed.

The stack frame corresponding to the advice shows logger.ajc$before$aspt_logger1d9fa4f9c() but this never appears in the source code.

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

Listing 3.1: An example of base program

```
1 import app.Point;
2 public aspect logger {
3   before() : call(* Point.*(..)) {
4     /* do something */
5   }
6 }
```

Listing 3.2: An example of aspect program

Third, problems like the inability of locating source code are easily encountered. Eaddy et al. [13] classified this problem as code-location problem. One important purpose of debugging is locating language constructs in the source code. The compiled code may undergo a series of transformations during which the source location information is not maintained. This causes the debugger to show no or the wrong source code, or to show compiled and woven *intermediate code* instead of the original *source code*.

The following example shows how a base program is transformed after weaving. Listing 3.3 shows the bytecode of the line 4 in listing 3.1. After adding an aspect which is presented in listing 3.2, the bytecode for the same line becomes listing 3.4. Compared with the listing 3.3, the bytecode in listing 3.4 inserts two instructions of method invocation. However, these two invocations are not written in the source code.

```
1 aload 1: p
2 invokevirtual Point.move() : void
```

Listing 3.3: Bytecode of the line 4 in listing 3.1

```
1 aload_1: p
2 invokestatic logger.aspectOf() : logger
3 invokevirtual logger.ajc$before
4     $aspt_logger$1$d9fa4f9c() : void
5 invokevirtual Point.move() : void
```

Listing 3.4: Bytecode of the line 4 in listing 3.1 after weaving with aspect in listing 3.2

3.2 A Debug Model for Advanced-dispatching Languages

As described in the previous subsection, the Java debugger does not increase program comprehensibility and it causes many problems when it is used to debug AD programs.

Therefore, it is necessary to design and implement a dedicated AD debugger that is what this thesis aims for.

Eaddy et al. [13] proposed an AOP debug model that classifies AOP-specific program composition techniques and activities, and relates them to the AOP-specific faults they induce, and specifies criterions that all AOP debugging system should support. Their work gives me a good guidance about developing an AO-enabled debugger. Besides, it fits well to the context of my debugger because an AO program is one kind of AD program. In this section, I adapt and extend this AOP debug model to the domain of advanced-dispatching and propose a debug model for AD programming.

The AOP debug model has five components: weaving strategies, AOP activities, a fault model, a definition for debug obliviousness, and debugging criterions. The weaving strategies component classifies weaving into invasive weaving and non-invasive weaving. Invasive weaving is further classified into source weaving and binary weaving. Because how code is woven is not in the scope of this thesis, this component will not be discussed. Besides the debugging criterions component uses the definition for debug obliviousness, these two components are merged. The following subsections discuss activities, faults and criterions respectively.

3.2.1 A Classification of Advanced-dispatching Activities

An AD activity is any program behavior supporting to perform on advanced-dispatching mechanism. In [13], six AOP activities are classified. Some can be adapted to AD activities and some are not applicable. To illustrate how concepts of AO activities are migrated to the AD domain, the following table lists all AOP activities and their purposes. For each AOP activity, a corresponding AD activity is given in the third column. Following paragraphs introduce each AD activity in detail.

Dispatch function evaluation is an activity determining at runtime which actions are going to be performed and in what order. The output of *dispatch function evaluation* is a list of actions which are going to be performed. While the output of the *dynamic aspect selection* is a list of aspects whose advices are going to be executed. Each action has only one declaring class or aspect. Therefore, the program composition indicated by outputs of two activities are the same. The fourth step "Evaluate dispatch function" in figure 2.4 shows when this activity is carried out.

Attachment deployment and undelpoyment is an activity enabling or disabling an attachment in an execution environment at runtime. Actually, the purpose of the *aspect*

AOP activity	**Purpose**	**AD activity**
Dynamic aspect selection	Determines at runtime which aspects apply and when	Dispatch function evaluation
Aspect instantiation	Instantiates or selects aspect instances	Attachment deployment and undeployment
Aspect activation	Alters control flow to execute advice and provides access to join point context	Call interception
Advice execution	Execution of the advice body	Action execution
Bookkeeping	Maintains additional AOP dynamic state	Bookkeeping
Static scaffolding	Static modifications to the program's code, type system, or metadata	-

Table 3.1: AOP activities and corresponding AD activities

instantiation is deploying aspects and thus it is extended to this activity. This activity can be mapped to the second step "Deploy attachments" in figure 2.4.

Call interception is an activity switching control flow from user written programs to the execution environment in order to access call context, evaluate dispatch function and eventually perform actions. This activity can be mapped to the third step "Intercept a method call" in figure 2.4.

Action execution is an activity executing an action. The *action* has a broader scope because it not only includes advice body but also other ordinary methods. This activity can be mapped to the fifth step "Reflectively invoke actions" in figure 2.4.

Bookkeeping is an activity maintaining additional AD dynamic state. Similar to the AO programs, additional states like call stack, calling depth, etc., are needed to be maintained. This activity can be scattered around the whole dispatching process and embedded in other activities.

The activity *static scaffolding* is not applicable for predicate dispatching languages, like JPred. Therefore, this activity is discarded in the AD domain. Figure 2.4 shows a mainstream workflow performing dispatching. All steps except step one which is not performed in execution environment are classified as activities introduced above.

3.2.2 Fault model

Each activity introduced in the previous subsubsection may introduce new types of faults which base programs do not have. This subsection does not aim at listing all possible faults. Some typical fault examples are given below.

Incorrect dispatch site. An advanced-dispatching is performed/not performed at an unexpected/expected function call site. This fault is caused by incorrect *Pattern*s and it is exhibited by activities *dispatch function evaluation* and *call interception*.

Incorrect program composition. The relationship between multiple applicable actions at one dispatch site is incorrect. For example, incorrect advice type (e.g., before, around), incorrect sequence (e.g., declare precedence). This fault is caused by incorrect *ScheduleInfo* and it is exhibited by the activity *action execution*.

Incorrect context exposure. A context is not exposed as intended to action. This fault is caused by an incorrect type of the context, like callee, argument, or incorrect value of the context and it is exhibited by activities *dispatch function evaluation* and *action execution*.

Incorrect functional change. A performed dispatching alters the functionality of the base program in such a way that it ceases to work properly. This fault is caused by incorrect action selection or logic errors in actions and it is exhibited by the activity *action execution*.

Incorrect attachment deployment and undeployment. An attachment is deployed or undeployed at an unexpected time that causes incorrect program behavior. This fault is exhibited by the activity *attachment deployment and undelopoyment*.

3.2.3 Properties of an Ideal Debugging Solution for Advanced-dispatching Languages

Based on the properties proposed in [13], discussions about properties required from a debugging solution for AD programs are given below.

Idempotence. Preservation of the base program's debug information no matter whether the advanced-dispatching programs are performed.

Debug obliviousness is defined as "the ability to hide AOP activities during debugging" in [13]. The base program generally performs the most logic and it can be executed without aspects. Therefore, it is necessary to inspect the execution flow of the base program alone. However, predicate-dispatching programs uses the advanced declarations as part of the main logic. Thus, the debug obliviousness in the AD domain requires to ignore all AD

activities except the *action execution*. Whether to ignore the *action execution* should be handled differently according to specific languages.

Debug intimacy. The ability to observer all activities in their full details. Full detail means all related debugging information including runtime states, stack trace, breakpoints, CPU utilization, especially the language-specific information. Considering existing AO debuggers, some desired language-specific information are missing.

Dynamism. The ability to deploy or undeploy attachments in order to enable or disable attachments at runtime.

Aspect introduction is defined as an ability introducing a new attachment without restarting in [13]. In ALIA4J, there is no corresponding entity which can mapped properly to the concept *Aspect*, the top element declaring advanced-dispatching is *Attachment* in ALIA4J. Therefore, this property is called ***Attachment introduction*** in the AD domain.

Runtime modification is defined as an ability to modify code at runtime and execute the modified part without restarting. This definition does not tell to what extent the code can be modified. Adding a new aspect can be a modification and deleting all code can also be a possible modification. From the evaluation of AOP systems in [13], none of existing systems but their *Wicca* provides this ability. This property is not considered in the development of this thesis.

Fault isolation is defined as an ability which automatically determines if a fault lies within the base code, advice code, or some other AOP activity code in [13]. Actually, locating a fault is a complicated task which needs strong artificial intelligence. Because, there is no guarantee that any program is fault-free and the cause of a fault needs analysis in specific contexts. However, providing all affected places of an entity to narrow down the analysing scope is feasible. ***Affection exploration*** can be deemed as a compensation of the *fault isolation*. Besides, it provides another way for developers to better understand program structure and behavior.

Locating is the ability to locate AD-specific constructs in source code correctly. It should be noted that locating for constructs present in base programs is classified to *idempotence*.

To summarize, an ideal debugging solution for AD languages should at least provide the following properties: idempotence, debug obliviousness, debug intimacy, dynamism, attachment introduction, affection exploration and locating.

Chapter 4

Approach Supporting Seven Properties

Based on the seven properties proposed in the previous section, this section discusses how they will be met in the implementation.

4.1 Idempotence

In ALIA4J, all investigated languages use Java as their base language. So the idempotence can be interpreted as the ability to debug Java programs. There are already some mature debugging tools for Java programs, like jdb, or the Eclipse Java debugger. They have provided many debugging functionalities and section 2.4 describes how to debug a Java program in Eclipse. The development of AD debuggers emphasizes providing AD-related debugging information. Therefore, the AD debugger can extend an existing Java debugger which shares responsibility for debugging Java programs.

In my development, the Eclipse Java debugger is chosen to be the extending base. It is built based on the Java Platform Debugger Architecture (JPDA) which has two layers, the debugger side and the debuggee side. Thus, the AD debugger needs to adapt these two layers into the domain of AD languages.

4.2 Debug obliviousness

In order to provide debug obliviousness, the debugger should be able to identify AD activities. Code related to all AD activities except the *action execution* lies in the NOIRIn

infrastructure. Therefore, those activities can be ignored by skipping NOIRIn infrastructure code during debugging.

As discussed in section 3.2.3, *action execution* is only ignorable for advices in AO programs. In ALIA4J, every action including advice or methods is composed into an *Attachment* which has *ScheduleInfo*. In order to distinguish the execution of advice and methods, there is a dedicated subtype of *ScheduleInfo* for normal methods.

The current stepping strategies which include "Step Into", "Step Over" and "Resume" are not capable of ignoring execution of specified action. The debugger needs a new stepping strategy and a possible algorithm is listed below.

1. The debugger obtains all performing actions from the evaluation result at this dispatch site.

2. The debugger finds the entrance location for each action and adds internal breakpoints there.

3. When the developer chooses an action to step to, the execution is resumed until it reaches a desired internal breakpoint.

4. All internal breakpoints added in this process are deleted.

4.3 Debug intimacy

NOIRIn performs all dispatching processes so that it is omniscient to dispatching details, such as call contexts, dispatch functions. In order to expose this information to the debugger, NOIRIn, as a part of the debuggee program, should provide them at an appropriate time and in an easy-to-access place. The time can be notified by a special event triggered when all required information is just updated. The event representing accomplishment of information updating at the debuggee side can be mapped to a creation of a specific frame at the debugger side. Thus, the debugger knows that all needed AD-related information is prepared when this frame is created. The place providing AD-related information should be visible at this specific frame and data should be organized in a dispatch-site-centric way.

Because NOIRIn infrastructure is also written in Java, AD-related information are mirrors of Java constructs after being sent to the debugger side, like *ObjectReference*. In order to let this information represent mirrors of AD constructs, a dedicated debug architecture for AD programs needs to be built and obtained mirrors should be adapted into constructs of the architecture. For example, an *ObjectReference* mirroring an *Attachment* is passed to

the debugger side through JPDA. The conventional debugger only turns it into a mirror of a normal object of a class called *Attachment*. To assign it with specific role of attachment in the AD debugger, the mirror can be wrapped into a dedicated object representing the mirror of an *Attachment* using object composition. Calls to dedicated functionalities of *Attachment* can be forwarded to the wrapped *ObjectReference*. Then the *ObjectRefence* uses interfaces defined in JDI to accomplish the required task.

The way presenting obtained information is important for developers because an inappropriate way may decreases efficiency significantly. As introduced in section 2.2, the dispatch function is represented as a binary decision diagram (BDD). Besides, the structure of LIAM models and relationship between LIAM entities are complicated. Textual description is limited to show this information but a graphical representation can provide more intuition and comprehensibility.

4.4 Dynamism

NOIRIn provides functions for deploying and undeploying attachments at runtime. So the debugger side should provide a user interface calling corresponding functions.

4.5 Attachment introduction

An attachment consists of many LIAM entities and each of them needs to be created before introducing this attachment. Developers who are not familiar with LIAM need to spent some time learning documentation before creating an entity correctly. To ease the burden of learning documentation, an attachment creation guide should be provided, such as a panel or a wizard.

4.6 Locating

The locating ability cannot be achieved by the debugger alone and it needs support from compiler. Because all debugged entities are outputs of the compiler, such as bytecode, or LIAM model actual. Some debuggers are able to map the compiled output to the source code in order to perform symbolic debugging [23]. The mapping uses locations stored in the compiled output to locate corresponding source code. However, the correspondence between source code and compiled code is broken because of code transformation. This problem is described in section 2.1.

Therefore, LIAM constructs should store locations of the original code before transformation. Fortunately, NOIRIn does not perform code weaving and thus code locations are not changed. When location of a certain construct is required, the debugger reads this information through the underlying architecture introduced in *Debug intimacy* and then highlights related source code.

4.7 Affection exploration

The developer should be able to specify which construct needs to be explored and view the exploring result at the debugger side. The debuggee side should be enhanced with functions finding out affected places according to a given construct. For example, if the developer finds that an unexpected result occurs after calling a method, the bug could lie in places affecting this method or affected by it. Table 4.1 lists some frequently used constructs with their affected places.

Construct	**Affected places**
Field	access sites (read and write), matched attachments, etc.
Method	accessed fields, called methods, overriding methods, call sites, matched attachments, etc.
Class	declared fields, declared methods, interfaces, super classes, subclasses, matched attachments, etc.
Attachment	schedule information, action, specializations, matched join point shadows, attachments sharing the same join point shadow, etc.

Table 4.1: Affected places for some frequently used constructs

Chapter 5

Three-layer Debugger Architecture

5.1 Overview of Advanced-Dispatching language Debugger Architecture

Section 2.2 has introduced the ALIA4J approach for implementing AD languages. ALIA4J provides a uniform representation of programs written in different AD languages in order to enable the reuse of implementations between these languages, and it embodies the execution semantics for AD in a language-independent way. Furthermore, its representation of AD stays first-class during the execution. Therefore, a debugger for AD languages is presented, which is based on the ALIA4J approach, in order to improve the tooling landscape for multiple existing and future AD languages at once. Basically, my work will allow the developer to debug the ALIA4J representation of AD instead of woven bytecode as in traditional approaches. The ALIA4J representation is much closer to the original source code than the woven bytecode, and it can preserve more AD-related debug information.

The *Advanced-Dispatching language Debugger Architecture* (ADDA) is developed as an extension to the *Java Platform Debugger Architecture* (JPDA) [2] based on ALIA4J's representation of advanced-dispatching. Figure 5.1 shows the overall structure of the ADDA. The *Execution Environment* component is grey because it is extended with functionalities supporting debugging. The *AD Information Helpers* (ADIH) extend an ALIA4J-based execution environment with functionality that allows to inspect the AD-related context of the running program and to interact with the execution of advanced dispatch. The communication is channeled through the standard Java Platform Debugger Architecture (JPDA) and the AD-related data is represented by the *Advanced-dispatching Debug Interface (ADDI)*

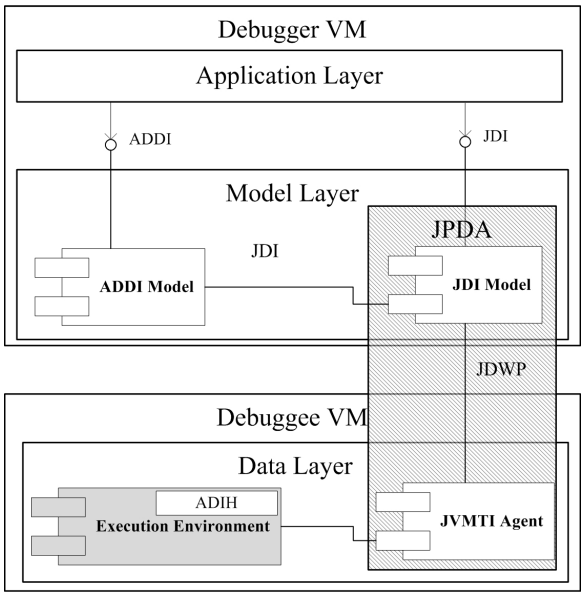

Figure 5.1: Structure of the Advanced-Dispatching language Debugger Architecture

Model. A debugger front-end for AD can connect to the ADDI and the JDI in order to debug the execution of AD and of standard Java in a program.

The front-end of the AD debugger is integrated into the Eclipse IDE, although any IDE with a comparable infrastructure would also be applicable. The AD debugger extends the Eclipse Java debugger, which is used for the program parts that do not use advanced-dispatching, with additional user interfaces. These are Eclipse views specific to visualizing and interacting with ALIA4J's representation of advanced dispatch in order to satisfy the properties motivated in section 3.2.3. In some cases, the behavior of existing debugger views has been changed to adapt to AD features.

The following subsections will discuss each component in detail. Section 5.2 presents the required extensions to an ALIA4J-based execution environment for communicating with the debugger. The JPDA has been introduced in 2.3 and section 5.3 shows an extended debug interface based on JDI that reflect the AD extensions to languages. The user interfaces are described in section 5.4.

5.2 Extensions to NOIRIn

5.2.1 Workflow Extension for Enabling Debugging

Section 2.2.1 describes the workflow of deploying an advanced-dispatching declaration and of executing advanced dispatch in NOIRIn. However, information, like when NOIRIn should be suspended and then communicate with debugger, what kind of data should be provided to debugger, where are these data stored, etc. are currently not available in NOIRIn. Based on the original workflow presented in figure 2.4, figure 5.2 adds some additional steps in the workflow for supporting debugging in NOIRIn. Steps of the original workflow are put in a grey box, while other steps are additional.

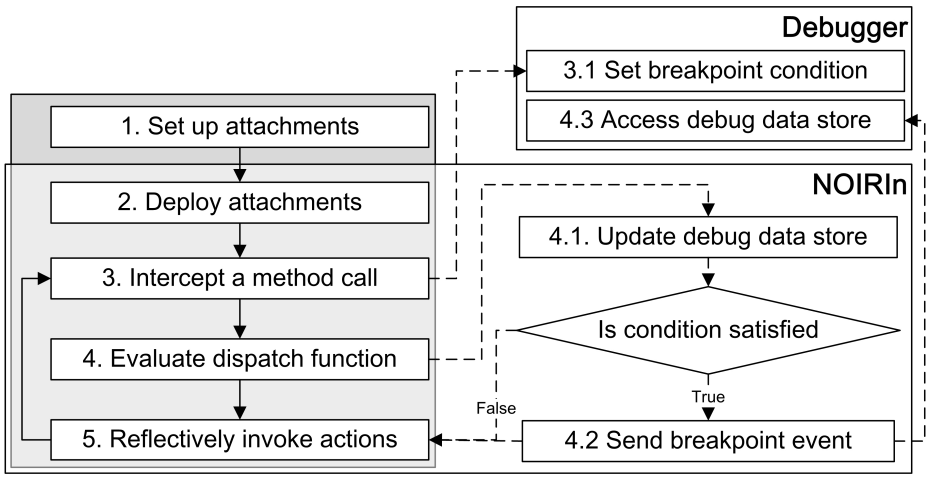

Figure 5.2: The work flow of intercepting a method call extended with ability supporting debugging in NOIRIn

After actions to perform have been determined in the step 4, almost all needed information including dispatch site, call context, etc., for invoking the next action are available. This information is wrapped in a *Frame* object and this *Frame* is pushed into a *Stack* which is stored in a singleton store. Thus, the store concentrates all required AD debug information and provides the only accessing entrance to the debugger.

An *If* statement invoking breakpointShadow() is added right after the code where the store is updated and it is presented on line 6-9 in listing 5.1. The breakpointShadow() is a method with one statement body which provides an execution point where the program can be suspended. The breakpoint point is set at the entrance of breakpointShadow() instead of line 8

because JDI provides an interface returning the entrance location of a specified method. Besides, the correctness of the returned location is not affected by modifications to the source file as long as the signature of the method is not changed. Therefore, there is a specific frame suspending at Interpreter.breakpointShadow() where the data in the store can be inspected.

```
1  public class Interpreter extends org.alia4j.fial.System {
2    private String breakpointFileName = '''';
3    private int breakpointLineNumber = −1;
4    private Object interpretCallSite(CallContext callContext) {
5      ... /* The store has been updated */
6      if(this.breakpointLineNumber == callContext.callLineNumber &&
7           this.breakpointFileName.equals(callContext.callerFileName)) {
8           breakpointShadow();
9      }
10     ...
11   }
12   //This function is only called by debugger
13   public void configureBreakpoint(String fileName, int lineNum) {
14     this.breakpointFileName = fileName;
15     this.breakpointLineNumber = lineNum;
16   }
17 }
```

Listing 5.1: Code about how a conditional breakpoint shadow is added and configured

The condition on line 6-7 specifies when the breakpointShadow() is executed and it consists of a line number and a file name. We assume that the current condition is strong enough to distinguish different execution points. The initialization shown on line 2-3 sets values that no point would match. The method configureBreakpoint is only called at the debugger side. When a call is intercepted at step 3, the debugger can require to suspend the program at breakpointShadow() for this call. Thus, the debugger reads the call context and passes specific values of the line number and file name to configureBreakpoint() (3.1). When the program runs to the breakpointShadow(), a breakpoint event is sent to the debugger (4.2). Then, the program is suspended and the debugger accesses the store until it asks the debuggee VM to resume.

5.2.2 Improvement to NORIn for simplifying debugger implementation

On the debugger side, the debugged entities are mirrors, code using mirrors needs to access fields and invoke methods reflectively and handle exceptions that may occur during reflection. To implement some desired functionalities which are not currently provided in the NOIRIn is troublesome. For instance, our debugger provides a functionality for undeloying a single attachment at runtime. NOIRIn offers a method undeploy(Attachment...) which is used for undeploying a list of attachments. To call this method at the debugger side, passing one argument which is the mirror of the attachment to undeploy to this method causes a type incompatible error. A correct argument should be the mirror of an array with one element which is the mirror of the attachment. Besides the code for creating this argument and invoking the method reflectively, some exceptions relating to *Reflection* need to be handled. It is a tedious and error-prone process to write code like this. Therefore some Advanced-Dispatching Information Helpers (ADIH) are implemented in NOIRIn such that each task can be performed just by one reflective method invocation. In this example, a method undeploySingle(Attachment) is implemented to accomplish this task in NOIRIn.

Atomic predicate are evaluated by passing context values to the method isSatisfied during the evaluation of the dispatch function. There is no variable saving the evaluation result because each atomic predicate is evaluated exactly once at a dispatch site. The next time when an atomic predicate is evaluated, context values are updated. There is no guarantee that the evaluation result would remains the same. When the evaluation result is required by the debugger, isSatisfied needs to be invoked again in the reflective way and different atomic predicates need different contexts. Therefore, a field evaluateResult is added to store the latest evaluation result. The evaluateResult is updated as soon as the corresponding atomic predicate is evaluated. The method isSatisfied returns a boolean value so that evaluateResult has three values which represent a true result, a false result and a unevaluated state respectively. The mirror of evaluateResult at the debugger side has one more value representing an unrequested state.

Besides, NOIRIn does not always provide interfaces for accessing AD declaration entities in a way suitable for debugging. Sometimes, further tasks need to be performed to adapt provided data to a required form. For instance, the debugger needs to find all matched methods in a specified class according to a *Pattern* of an *Attachment*. NOIRIn only provides an interface for reading the *Pattern*, the function matchedClassMethods(MethodPattern, Class) is added for this purpose. Another example is that the debuggee side puts *AttachedAction* objects in the dispatch strategy. However, the dispatch strategy on the debugger side is

designed to store *Attachment*. Therefore, a helper function attachedAction2Attachment() for finding the related *Attachment* according to an *AttachedAction* has been implemented.

All other information helpers and their usages are introduced below. But it should be noted that more ADIHs will be added with further development of the debugger.

getDeployedAttachments() returns an array of deployed attachments. The deployed attachments are stored in a *Set* in NOIRIn, this function performs the task transforming a *Set* to an *Array*.

getAttachmentsOnClassMembers(String) returns all attachments matching any member of a specific class. The class is specified by the fully qualified class name.

getAttachmentsOnMethod(String) returns all attachments matching a specific method. The method is specified by a string of the method signature.

attachedActionSet2Attachments(GenericFunction, Set) performs a task finding corresponding attachment for each attached action in a given set and returns an array of the retrieved attachments.

clazzForName(String clazzName) returns a loaded class in the debuggee JVM according to a given class name.

deploySingle(Attachment) deploys a given attachment.

5.3 Advanced-Dispatching Debug Interface

The Advanced-dispatching Debug Interface (ADDI) extends the Java Debugging Interface (JDI) by adding advanced-dispatching-related features to some existing entities and introducing new advanced-dispatching related entities. The structure of ADDI is presented in figure 2.6. Interfaces presented in dark grey diagrams are defined in JDI. Interfaces presented in light grey diagrams are entities defined in JDI but extended with advanced-dispatching related features. While the rest of interfaces presented in white diagrams are new interfaces introduced in ADDI. Some JDI entities like interfaces plus its methods are introduced in section 2.3, only extended parts are discussed in following paragraphs. In order to distinguish classes from debuggee side and debugger side, class in `type writer` style is from debuggee side and *italic* style is for debugger side.

A dispatch site, e.g. a field access, or a method invocation, may be matched by a pattern of an attachment. So the `TypeComponent` is extended with a method matchedAttachments() providing a list of matching attachments. A `Field` can be read or written so that it is extended with methods (readMatchedAttachments() and writeMatchedAttachments()) providing a list of attachments matching its reading and writing respectively. Similarly, `ClassType` is

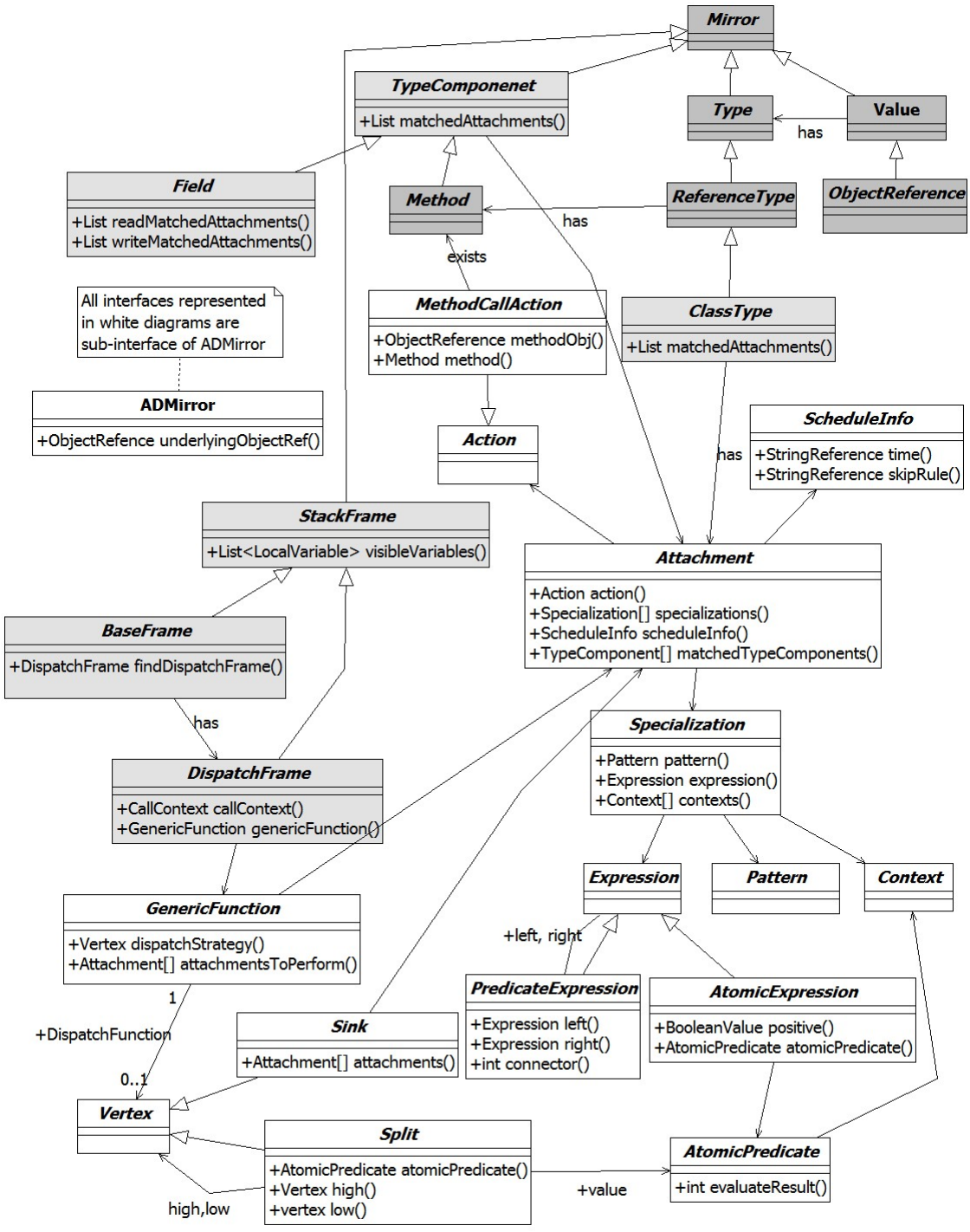

Figure 5.3: UML class diagram of Advanced-Dispatching Debug Interface (ADDI). Interfaces presented in dark grey diagrams are defined in JDI. Interfaces presented in light grey diagrams are entities defined in JDI but extended with advanced-dispatching related features. Interfaces presented in white diagrams are new interfaces introduced in ADDI.

extended with a method matchedAttachments() providing a list of matching attachments of its members. In order to leave JDI intact, these methods are introduced in an AspectJ program.

`BaseFrame` and `DispatchFrame` reify different stack frames according to where it resides. `BaseFrame` denotes a frame which is in the user-written code. `DispatchFrame` is a specific frame which resides at the breakpointShadow() introduced in section 5.2.1 where the dispatch strategy has been computed but not yet been performed. According to the execution flow in NOIRIn, a `BaseFrame` is always accompanied with a `DispatchFrame`. `BaseFrame` can find out the accompanying `DispatchFrame` by invoking method findDispatchFrame(). `DispatchFrame` can provide dispatching information like dispatch site (genericFunction()) and call context (callContext()).

`ADMirror` reifies an AD related entity and it is the root interface for all other interfaces in white diagrams. It wraps an `ObjectReference` which is the mirror of an actual object in the debuggee program. Mostly, the "mirror-wrapper" `ADMirror` provides AD related information and functionalities by accessing fields or invoking methods of the wrapped `ObjectReference`. If the required data cannot be provided by the wrapped `ObjectReference` alone, then the request needs to use ADIH.

`GenericFunction` reifies a dispatch. The generic function is a unique identification, e.g., signature, which may be shared by multiple methods. A `GenericFunction` has a dispatch strategy (dispatchStrategy()) and an array of `Attachment`s (attachmentsToPerform()) whose actions are going to be performed at the current call site. The dispatch strategy stores a dispatch function represented as a binary decision diagram (BDD) which is introduced in section 2.2. Nodes of the BDD are instances of `Vertex` which is either `Split` or `Sink`. `Split` represents an inner node which has two children nodes (high() and low()) and an `AtomicPredicate`. During the evaluation of the dispatch strategy, an evaluating `Split` chooses one of its child node as the next evaluating node according to the value of atomicPredicate.evaluateResult(). The evaluation ends when a `Sink` is reached. `Sink` represents a leaf node containing an array of `Attachment`s.

`Attachment` reifies an attachment. It provides interfaces for accessing components of `Attachment`. If all `Specialization`s are matched, the `Action` will be invoked at the time that the `ScheduleInfo` specifies. Besides, it can provide a list of `TypeComponent`s which match a pattern of the attachment and a mirror of the actual method that its `Action` implies. A `Specialization` has an `Expression` specifying a residue, a `Pattern` specifying a joinpoint shadow and several `Context`s which need to be exposed. An `Expression` is either a `PredicateExpression` which has two sub-expressions or a `AtomicExpression` containing

an `AtomicPredicate`. `ScheduleInfo` has a time (time()) describing the relationship between its attachment's action and the dispatch site, e.g., before, around and after. Currently, only one type of `Action` is implemented — `MethodCallAction`. Other types of actions includes `FieldReadAction`, `FieldWriteAction`, etc. `MethodCallAction` can find out the concrete method in term of the mirror of an instance of class `java.lang.reflect.Method`.

Each interface introduced above has an implementation type which is named with the interface name appended with "Impl". For example, the implementation type of `ADMirror` is `ADMirrorImpl`. Basically, `ADMirrorImpl` and its subtypes expose all declared fields (getDebugFields()) to the debugger. However, some types have auxiliary fields which are not necessary to be exposed. Therefore, `ADMirrorImpl` provides a function filteredDebugFields() which can be called by its subtypes to choose exposed fields. As shown in figure 5.4, the `MethodCallActionImpl` only exposes field method by using filteredDebugFields().

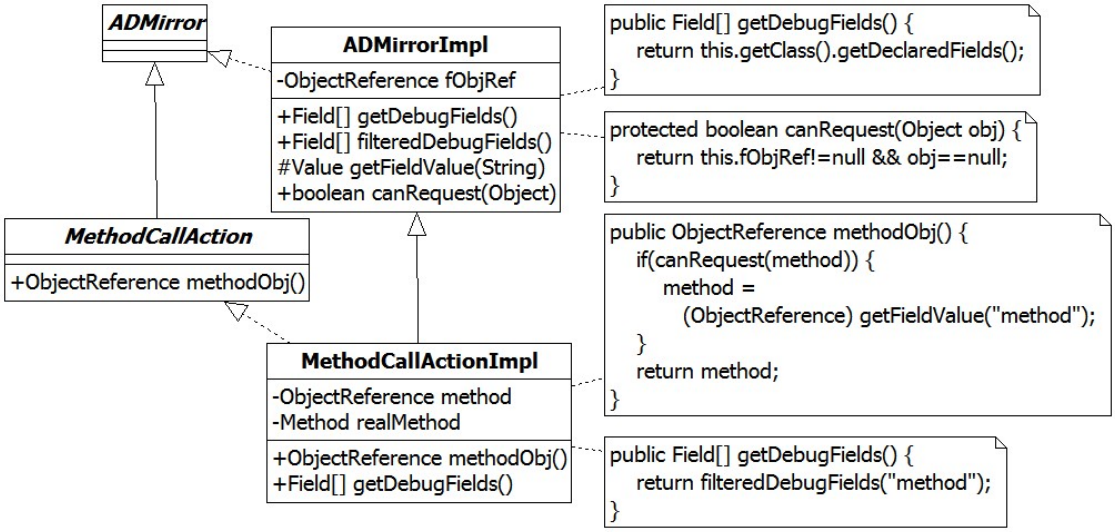

Figure 5.4: Implemtation details of *ADMirror* and its subtype *MethodCallAction*

To reduce the times sending requests to the debuggee JVM, almost all requested results are stored in fields. Therefore, repeated requests on the same field can be returned from local memory. Like in figure 5.4, field method stores the request result for the first time and subsequent requests can be replied directly from this field. A method canRequest(Object) is provided in `ADMirrorImpl` to check if a field has not been requested and it returns true when both requester object and requestee object are not null.

If one field of a class is requested for the first time, all fields of this class are requested. Because probably the next request is on another field of this class. For example, the user interface which will be introduced in the next subsection always shows all fields' values of an object at the same time.

There are much more classes defined in ALIA4J than those mirrored and wrapped to ADDI. Theoretically, every ALIA4J class can be mapped a mirror in the ADDI model. But it is tedious and unnecessary to mirror the whole structure. There are three ways used for simplifying the ADDI model.

Reducing classes. In the implementation of LIAM model, `Sink` stores `AttachedAction` which is derived from `Attachment`. *AttachedAction* is replaced with *Attachment* in ADDI in order to decrease the number of classes.

Masking details. `Pattern` has six subclasses and one of them is `MethodPattern`. A `MethodPattern` consists of seven sub-patterns including a `ModifiersPattern`, a `TypePattern`, etc. Because the current implementation has not used these information so that providing these details is redundant. In the implementation of ADDI, *Pattern* temporarily uses a string field patternString to mask all underlying details. But in the future work, further information are required and this field will be replaced with a concrete *ADMirror*. Then, this approach is used to mask more specific details.

Reusing JDI. Every *ADMirror* instance wraps an *ObjectReference*, but not every *ObjectReference* instance has to be wrapped as an *ADMirror*. *ADMirror* can use *ObjectReference*s as its fields and further details can be handled by well-defined JDI. Compared to the previous approach, this approach reveals all details.

5.4 User Interface

The application layer presented in figure 5.1 consists of several views, like the *Variables* and *Expressions* views which are provided by the default Java debugger in Eclipse. The AD debugger adds three views, namely the *Advanced Dispatch* view, the *Advanced-Dispatching Structure* view and the *Deployed Attachments* view. Since the local behavior in AD programs can depend on complex dynamic context, the Advanced Dispatch view shows which context is accessed during dispatch and in which way. This is necessary for the developer to understand which actions are/are not executed at a dispatch and why this is the case.

5.4.1 Advanced Dispatch view

The *Advanced Dispatch* view is the central view of the debugger showing *runtime* information about the dispatch at with the debuggee is currently suspended. It lets the developer inspect the runtime values of AD entities in the current frame, foresee the program composition flows of the next generic function invocation, directly step to any action which is going to executed, etc. All values are presented textually in a tree viewer. The dispatch function, which is represented as a BDD in ALIA4J, is additionally presented graphically. ALIA4J's dispatch strategy is a special form of a branching program, for which a graphical representation should be intuitive to the developer.

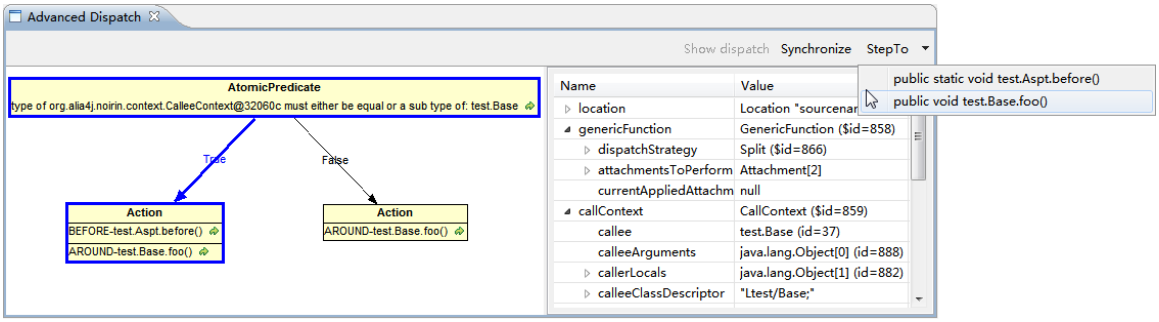

Figure 5.5: A snapshot of the Advanced Dispatch view showing the graphical representation for a dispatch strategy

A snapshot of the *Advanced Dispatch* view is given in the figure 5.5. The left window frame gives a graphical representation of the dispatch function for the next dispatch. The root node represents an *AtomicPredicate* and two leaf nodes show different program composition flows according to the evaluation result of the *AtomicPredicate*. The bold lines indicate the actual evaluation result and actions which are going to be performed. From this view, the developer can clearly see why and when the advice *Aspt.before()* is executed.

In order to show `ADMirror` instances on labels in the graphical representation, `ADMirror` need a function returning a string describing its content in an intelligible way. As shown in listing 5.2, a *LabelTrait* aspect adds a desired function toLabel() to `ADMirror` by using a statement **declare** parents. ADMirror.toLabel() gives a default implementation and Split.toLabel() overrides it with a more specific behavior. Function toLabel() is not directly added to `ADMirror` because it is totally application-specific. How an entity is presented has nothing to do with what it is and different AD debugger implementations may use different representing ways.

```
1  public aspect LableTrait {
2    public interface ILabelEntity {
3      public String toLabel();
4    }
5    declare parents : ADMirror extends ILabelEntity;
6    public String ADMirror.toLabel() {
7      return this.underlyingObjectRef().toString();
8    }
9    public String Split.toLabel() {
10     return this.atomicPredicate().toMirrorString();
11   }
12   //...
13 }
```

Listing 5.2: LabelTrait aspect is added in order to present ADMirror instances in graph

At the top right of the view, a "StepTo" button is provided for suspending the program at any performing action. Section 4.2 has described how to realize this new stepping strategy. If the execution steps to *Aspt.before()*, the dispatch function uses green color to indicate which one is the current executing action. This is shown in figure 5.6. Besides, all elements in the graph are appended with an arrow where related details can be expanded if developer clicks. In this way, tree-like details are flattened in this graph and efficiency for detail inspection is increased.

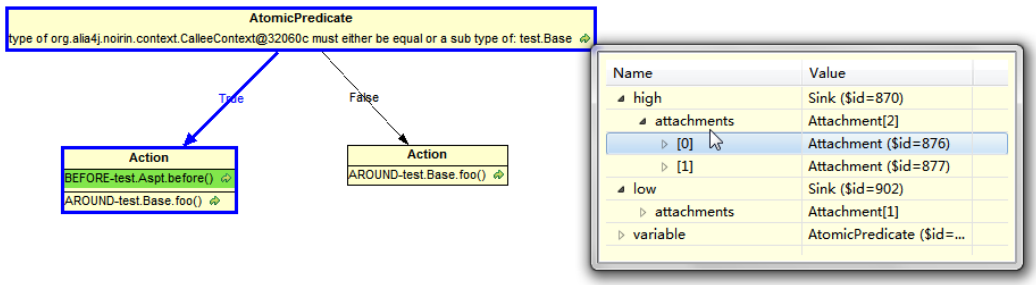

Figure 5.6: The dispatch function can show the current executing action and provide detailed inspection for each element.

Considering the complexity and importance of *Attachments*, the Advanced Dispatch view also provides a graphical representation for *Attachments*. As shown in figure 5.7, the developer can right-click an *Attachment* in the views's tree and select "Show graph".

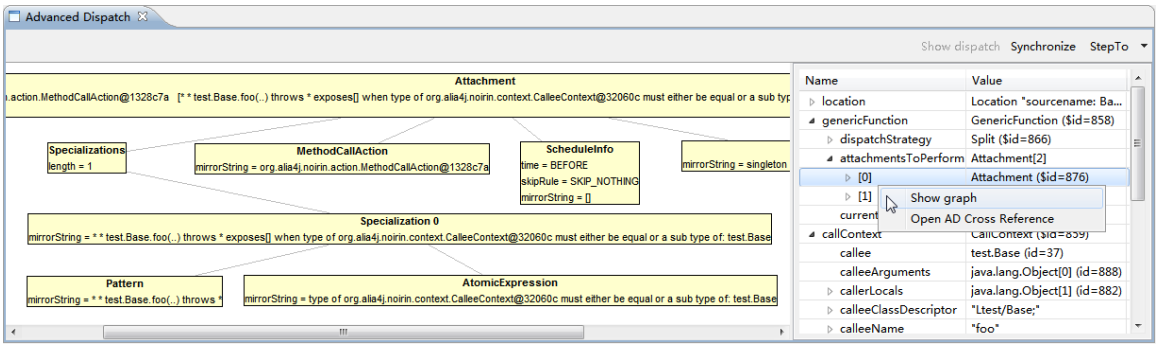

Figure 5.7: A snapshot of the Advanced Dispatch view showing the graphical representation for an attachment

5.4.2 Advanced-Dispatching Structure view

The *Advanced-Dispatching Structure* view assists the developer to explore *static* AD information within the project scope, i.e., all dispatching declarations in the program as well as all the generic functions in the program. This view also allows the developer to navigate from a dispatching declaration to the affected generic functions and vice versa. In some cases, a method may be matched by multiple *Attachments*. However, this information can not be shown in the *Advanced Dispatch* view in an explicit way. Figure 5.8 shows how the *Advanced-Dispatching Structure* view explores the affection of an AD entity. In this figure, the *Attachment* matches the method *Base.foo()* which is matched by only one *Attachment*.

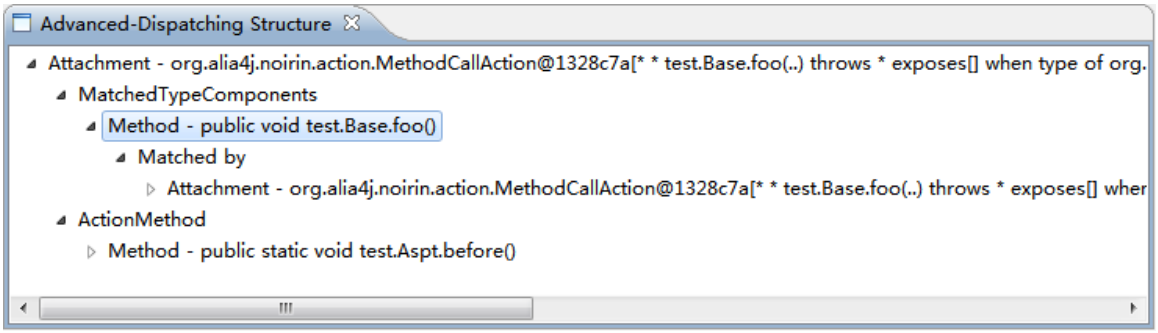

Figure 5.8: A snapshot of the Advanced-Dispatching Structure view

The UML diagram of the model underlying this view is shown in figure 5.9. *CrossRefTarget* adapts the exploring target object into an instance of *CrossRefSubject* because the view does not present *CrossRefTarget*. Instead, it starts at a *CrossRefSubject*. A *CrossRef-*

Subject has a list of named *Relation*s and each *Relation* has a list of *CrossRefSubject*s. In this way, the exploration can be performed between these two types infinitely.

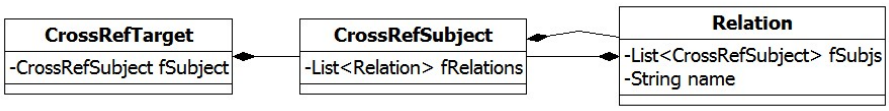

Figure 5.9: UML diagram of the model behind the Advanced-Dispatching Structure view

5.4.3 Deployed Attachments view

In order to dynamically deploy and undeploy attachments during runtime, the *Deployed Attachments* view is provided. It shows a textual representation of all attachments that are defined in the executing program together with a check box indicating whether the attachment is currently deployed or not. Unchecking or checking one of the items manually will lead to undeployment or deployment of the corresponding *Attachment* in the debugged program. A snapshot of the *Deployed Attachments* view is given in figure 5.10

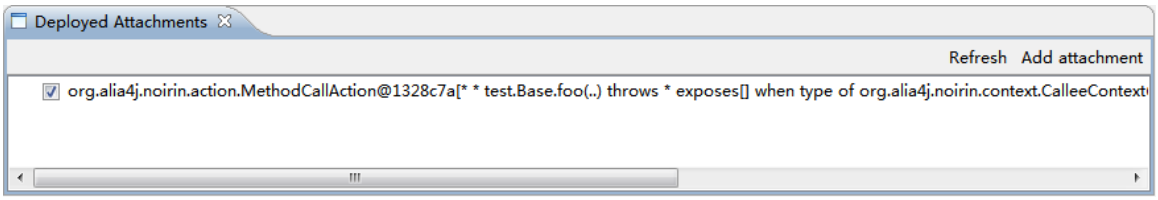

Figure 5.10: A snapshot of the Deployed Attachments view

For introducing new attachments at runtime, an attachment creation panel is provided when pressing the button "Add attachment" on the top right of the view. As figure 5.11 illustrates, pattern, expression, context, action, schedule are required to create a new attachment. Currently, simple panels for creating an attachment and its method pattern are shown in figure 5.11 and figure 5.12 respectively.

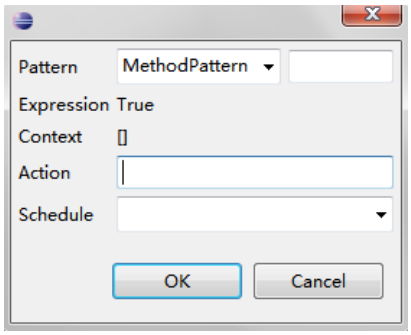

Figure 5.11: A snapshot of the attachment creation panel

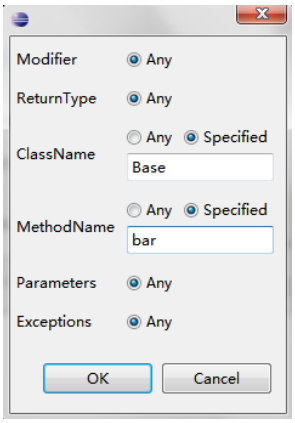

Figure 5.12: A snapshot of the method pattern creation panel

Chapter 6

Evaluation

To demonstrate the usefulness of the presented debugger, a walkthrough of one debugging session is presented. And then how well each property is met in the implementation is discussed.

6.1 An example

Supermarkets offer special prices for certain items and there are two types of promotion prices in this example. One type is used when an item is on sale, its price is decreased ten percents. Another type is used when a customer has enough credits obtained from previous shoppings, he can get one euro bonus. If two types are applicable, the price of the item is first deducted by the constant bonus and then cut down buy ten percents. Let us call it double-cut price. For example, the double-cut price for a 10-euro item is $(10-1)*0.9 = 8.1$ euro.

In a supermarket system, an aspect in listing 6.1 is used for handling special prices. The first advice from line 4 - 6 sets price of item which is on sale. The second advice from line 7 - 9 is for the bonus price.

```
1 public aspect SpecialPrice {
2   pointcut itemGetPrice(Item i) :
3     call(* Item.getPrice()) && target(i) && !within(SpecialPrice);
4   before(Item i) : itemGetPrice(i) && if(i.isOnSale()) {
5     i.setPrice((float) (i.getPrice() * 0.9));
6   }
7   before(Item i) : itemGetPrice(i) && if(i.isBonus()) {
8     i.setPrice((float) (i.getPrice() - 1));
9   }
```

```
10 }
```

Listing 6.1: A code example of an aspect

Running the program in listing 6.2, the printed price is 8.0 euro instead of the expected 8.1 euro.

```
1 public class Main {
2   public static void main(String[] args) {
3     Item item = new Item();
4     item.setPrice(10);
5     item.setBonus();
6     item.setOnSale();
7     System.out.println(item.getPrice()); // unexpected price
8   }
9 }
```

Listing 6.2: Main

Following list shows the process how to use the AD debugger finds out the bug.

1. Set a breakpoint at line 7 in listing 6.2 and launch the AD debugger.
2. The line contains multiple dispatch site. First the field System.out is read, next the method Item.getPrice is called and finally PrintStream.println is called. Thus, when the program is suspended at line 7, the developer must step over the first dispatch to suspend the JVM at the dispatch of item.getPrice().
3. Open the *Advance Dispatch* view and press "Show dispatch" button.
4. The dispatch function is shown in the view and it is given in figure 6.1. The bold lines tell the developer that three actions are going to be executed at this dispatch site. However, the order of the two advices is wrong according to requirements. The bug is found and developer can reverse their literal order to fix this bug.

This example introduces an *incorrect program composition* fault which occurs in the activity *dispatch function evaluation*. Only two advices from the same aspect are involved, the bug may be easily found by reading code. However, if more advices match the same call site and they are from different aspects, the conventional debugger is unable to show the program composition explicitly so that so that the faults become less obvious.

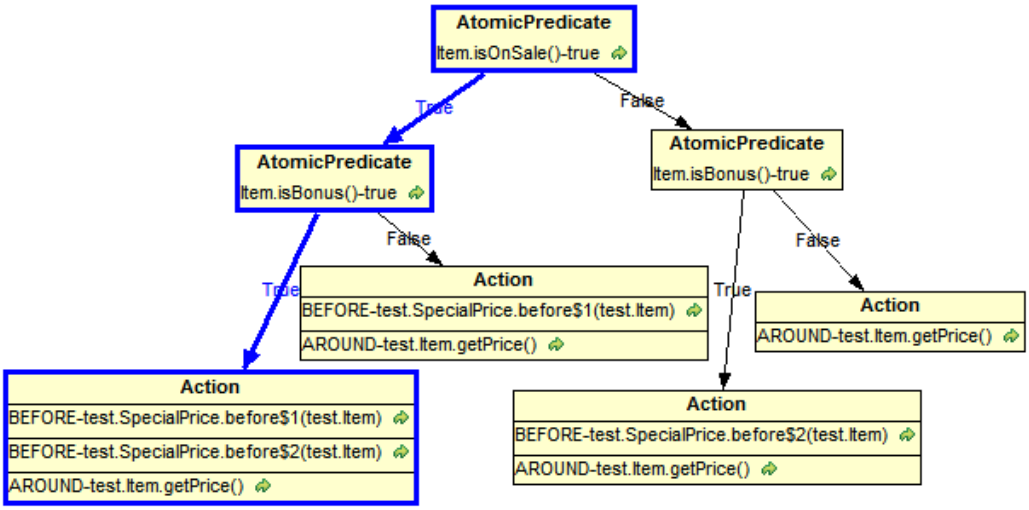

Figure 6.1: The dispatch function when Item.getPrice() is called in listing 6.2

6.2 Capabilities and Limitations

6.2.1 Idempotence

Capabilities. The AD debugger extends a conventional Java debugger, all required debugging information for Java programs can be provided by using the Java debugger.

Limitations. Because of a lack of documentation about the Java debugger plug-in in Eclipse, unexpected behavior occurs in the implementation when using Eclipse platform debug model. For example, variables in frames cannot be retrieved anymore after activating the extended debugger. This problem decreases the ability of idempotence. One possible reason of this bug is that Eclipse infrastructure has its own way handling stack frames. When altering creation of a *StackFrame* to creation of a *BaseFrame* or a *DispatchFrame* by using AspectJ, Eclipse loses the consistency between the platform debug model and the underlying debug architecture.

Summary. This property is supported but hampered by unknown technical problems.

6.2.2 Debug obliviousness

Capabilities. As described in section 5.4, the AD debugger adds a new step strategy which is stepping to the entrance of any performing action at a dispatch site. Which action to step to is chosen by developer at runtime so that the developer can decide whether to apply obliviousness at each dispatch site.

Limitations. To apply the new step strategy, the debugger needs inspection of the evaluation result at each dispatch site. Otherwise, there is no way foreseeing performing actions at a certain dispatch site. That means the debugger is unable to ignore all AD activities.

Currently the developer is required to add breakpoints manually and the debugger checks whether there is a breakpoint at the required location before performing the new stepping strategy.

Summary. This property is partially supported.

6.2.3 Debug intimacy

Capabilities. As introduced in section 5.2, AD-related debug information are provided at a specific frame. The debugger presents all obtained information in a tree view and uses a graph for some complicated constructs. At a certain dispatch site, provided information includes:

1. The location and call context of the dispatch site.
2. The dispatch function with its all related constructs, such as predicates and actions.
3. The evaluation result indicating which actions are going to be performed, in what order and why.
4. All currently deployed attachments in system.

Limitations. The debugger uses three ways simplifying the debug architecture and masks currently unimportant details. This is just a temporary measure for fast development.

Summary. This property is fully supported.

6.2.4 Dynamism

Capabilities. As shown in section 5.4, the *Deployed Attachment* view is implemented for this property. Attachments can be deployed and undeployed by checking or unchecking items shown in this view.

Limitations. Currently, there is no place storing created and then undeployed attachments in NOIRIn. However, the *Deployed Attachment* view only shows deployed attachments. Unchecked items will be removed if the view is refreshed so that the attachment it represents cannot be deployed again unless being recreated.

Summary. This property is supported but can be improved.

6.2.5 Attachment introduction

Capabilities. Panels are provided for creating and introducing new attachment at runtime.

Limitations. These panels only create limited attachments and following paragraphs discuss challenges specifying a complete attachment by using panel.

Patterns can be classified into method patterns, field patterns, etc. Each pattern can be divided into more specific elements. Take the method pattern for example, it consists of patterns of modifiers, return type, declaring class, method name, parameters and exceptions. Besides, each pattern may have wildcards so that the difficulty of checking whether specified patterns are valid or not is significant.

To specify a complicated expression with nested sub-expressions, a module creating a basic expression which contains an atomic predicate and a module constructing the expression with created basic expressions are required. One of the challenges is that the constructing module needs a flexible way adding, editing and deleting relationships between expressions, such as *and* and *or*.

There are various kinds of context and each kind may requires different auxiliary variables, like the argument context needs the index of the argument. Besides, the context should be compatible with the specified pattern, e.g. the dispatch matched by a field read pattern does not have a argument context.

The challenge for specifying actions is checking whether the given action exists or not. Using the *content assistant* may help to decrease the possibility specifying a wrong action. Finally, scheduleInfo is relatively easier because it only requires schedule time and priority.

Summary. The current AD debugger does not fully support this property.

6.2.6 Locating

Limitations. The *locating* here only means the ability locating AD constructs. Current development is carried out on intermediate model level instead of source code level. The compiling is bypassed so that no location information is stored.

Summary. This property is not supported but one solution is given in chapter 7.

6.2.7 Affection exploration

Capabilities. An *Advanced-Dispatching Structure* view is provided for supporting this property. Developer can use this view to find out attachments matching a type component, attachments matching a class, action that an attachment performs, etc. Table 4.1 has listed affected places for some frequently used constructs. Affected places of an entity is actually

an open list and some of them can be explored by other tools, like the *Hierarchy* view in Eclipse. Therefore, the *Advanced-Dispatching Structure* view only selects elements related to advanced-dispatching as exploring target.

Limitations. Attachment is currently unable to find out matched type components according to a pattern with wildcard names, such as `*.set*(..)`.

Summary. This property is basically supported.

6.2.8 Summary

The following table summarizes the discussion above and shows how well each property is supported in the AD debugger.

Property	**Fully supported**	**Partially supported**	**Not supported**
Idempotence	√		
Debug obliviousness		√	
Debug intimacy	√		
Dynamism	√		
Attachment introduction		√	
Locating			√
Affection exploration	√		

Table 6.1: How well each property is supported in the implementation

Chapter 7

A Solution for Supporting Locating

As explained in section 3.1, code may be transformed after an advanced-dispatching program is compiled. The location information stored in a class file is generated from the transformed code. This leads the conventional debugger to show wrong source locations during debugging. Section 3.2.3 has pointed out that the locating is one the most important functionality for debuggers. Section 6.2.6 explains why locating in the AD debugger is currently not implemented. In this section, a solution is designed for supporting locating.

NOIRIn performs weaving at the model level during runtime instead of at code level. Therefore, all location information should be stored in LIAM entities instead of class files. LIAM is built based on *semantics* which describes the behavior that a computer follows when executing a program in the language [16]. Different AD languages perform same dispatching behavior in different syntax. However, syntax constructs are not shared by all AD languages. Take the listing 2.1 and listing 2.2 for example, they will be transformed into LIAM models much the same in ALIA4J. In the AspectJ example, the time scheduling the advice is explicitly specified as "around". While the schedule time is implicit in the JPred example.

Besides, a LIAM construct is likely involved with multiple locations. For instance, an atomic predicate in the attachment corresponding to the AspectJ example requires that the first *argument* is an instance of class *Open*. The argument and required type are specified separately. Correspondingly, several *location ranges* should be given for describing the location of one LIAM entity.

Figure 7.1 presents a structure using the Composite pattern [15] in order to provide a feasible solution for storing location information. An interface *Locatable* is introduced and it is a super interface of *LIAMConstruct*, *LocationRange* and *ImplicitLocation*. All

classes of LIAM constructs, like *Attachment*, *AtomicPredicate*, etc., should inherit class *LIAMConstruct*. *LIAMConstruct* has a field *debugInfo* containing a name of the file where this construct lies and a list of *Locatable* entities. Besides, *LIAMConstruct* has a method involvedLocations() returning all involved *LocationRange*s. A *LocationRange* has two *LocationPoint*s representing a start position and an end position respectively. A *LocationPoint* consists of a line number and an offset within that line. For a construct which does not have an explicit source location, it is assigned with a *ImplicitLocation* as its symbolic location.

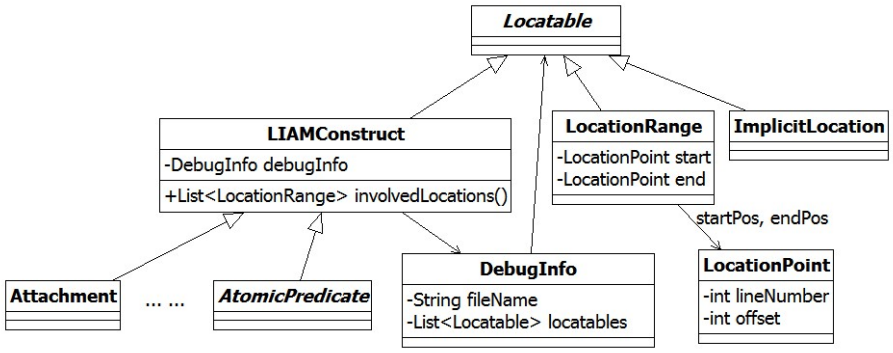

Figure 7.1: UML class diagram about adding debugInfo into LIAM constructs

Take the listing 7.1 for example, table 7 shows how *debugInfo* in each construct stores location information. Some notations are used in the table for simplification. "FN" means fileName, curly bracket "{}" represents a list, rectangular bracket "[]" represents a *LocationRange* and parenthesis means a *LocationPoint*. For example, "{[(1,2),(2,5)]}" describes a list with one element which is a location range. This location range starts at the 2nd character of line 1 and ends at the 5th character of line 2.

```
1 public aspect Aspt {
2   before(Square s) : call(* Shape.intersect(..)) && args(s) {
3     s.printNotification();
4   }
5 }
```

Listing 7.1: A code example of an aspect

Construct	DebugInfo	Related Code
attachment	FN, {scheduleInfo, specialisation[0], methodCallAction}	
scheduleInfo	FN, {[(2,3),(2,8)]}	before
specialisation[0]	FN, {methodPattern, basicExpression, argumentContext}	
methodPattern	FN, {[(2,27),(2,47)]}	* Shape.intersect(..)
basicExpression	FN, {instanceOfPredicate}	
instanceOfPredicate	FN, {argumentContext, [(2,10),(2,17)]}	Square s
argumentContext	FN, {[(2,36),(2,41)]}	args(s)
methodCallAction	FN, {[(2,61),(4,3)]}	{ s.printNotification(); }

Table 7.1: Locations of each LIAM construct generated from code in listing 7.1

To obtain all related location ranges of a LIAM construct, this construct needs to aggregate all location ranges of each *Locatable* entity stored in its *debugInfo*. The process recursively aggregates location ranges until a *LocationRange* or an *ImplicitLocation* is met. Take the *instanceOfPredicate* in table 7 for example, it relates two locatable entities, an entity specifying argument context and a location range contains `Square s` in the source code. The first entity *argumentContext* has only one involved locations which contains **args**(s). Therefore, the *instanceOfPredicate* has locations {[(2,36),(2,41)], [(2,10),(2,17)]} and relates to code `Square s` and **args**(s). An implementation of `LIAMConstruct.involvedLocations()` is given in the following listing.

```
1 public class LIAMConstruct {
2   private DebugInfo debugInfo;
3   private List<LocationRange> locations = new ArrayList<LocationRange>();
4   public List<LocationRange> involvedLocations() {
5     List<Locatable> lctbs = this.debugInfo.getLocatables();
6     for(Locatable lctb : lctbs) {
7       if(lctb instanceof LIAMConstruct) {
8         LIAMConstruct cons = (LIAMConstruct)lctb
9         this.locations.addAll(cons.involvedLocations());
```

```
10          } else if(lctb instanceof LocationRange) {
11            this.locations.add((LocationRange)lctb);
12          } else { // this locatable entity does not have location
13            // do nothing
14          }
15        }
16        return locations;
17      }
18  }
```

Chapter 8

Related Work

The proposed AD debug model is inspired by the work of Eaddy et al. [13] They implemented a tool called Wicca based on the AOP debug model. Wicca is a dynamic AOP system for C# applications that performs source weaving at runtime. The source code used in debugging is the woven source code and only contains Object-Oriented (OO) concepts. The AD debugger is designed only for Java based languages. It uses source code written by developer as the source view and defines debugging constructs in terms of AD abstractions, such as attachment, action.

AspectJ Development Tools (AJDT) [3] enables Eclipse platform to build, edit, debug AspectJ programs. It provides a lot of features decreasing the effort in understanding and coding AspectJ programs and the *Aspect Visualiser* and *Cross References* view are most representative ones. The *Aspect Visualiser* is used to visualize how aspects were affecting classes in a project in a "bars and stripes" style representation. The *Cross References* view is used in AJDT to show AspectJ crosscutting information, such as when a Java method is affected by advice. However, the debugger in AJDT is problematic because the conventional Java debugger is used and related problems are described in section 3.1. Compared to the AJDT, the AD debugger provides AD-specific information for extended programs and explores affection of an entity in a broader scope.

Borger et al. [11] found the pointcut-advice models of Java-based AO-technologies, such as AspectJ, JBoss AOP, and Spring AOP are very similar and thus defined a common debugging interface: Aspect-Java Debugging Interface (AJDI). The AJDI interface aggregates JDI mirrors and the information about aspect related structure and behavior. It is mentionable that they created events for dynamic AOP and events related to joinpoints and advices by transforming AJDI breakpoints into JDI breakpoints. Based on the AJDI,

they built up Aspect Oriented Debugging Architecture (AODA) which supports runtime visibility and traceability of aspect-oriented software systems. Compared to their work, the AD debugger is built for more general concepts which are also applicable for predicate-dispatching languages. However, locating is relatively easier in AODA because constructs defined in the architecture are language-specific.

AD-specific information provided by tools or systems for AD languages are not only provided as online debuggers as the work presented in this thesis. These other approaches can be used as auxiliary approaches to understand program behavior or structure during debugging.

Pothier et al. [21] implemented an AO debugger based on an open source omniscient Java debugger called TOD [22]. The TOD records all events that occur during the execution of that program and the complete history which is presented in terms of bytecode can be inspected and queried offline after the execution. In order to support full debugging intimacy, TOD registers the outcome of all the tests that occur during that activity. For enabling debug obliviousness, they used a tagging scheme to identify different aspect activities, like advice execution. Based on characters of omniscient debugger, TOD is extended to provide *aspect mural*s that show the activity of an aspect during the execution of the program. They also provide a view showing the execution history of the join point shadows of a particular pointcut so that which occurrences of join points matched and which ones did not can be viewed.

The JPred [19] Eclipse plug-in provides a view showing implication relationship between predicates used for methods sharing the same signature. It indicates that a method with a more specific predicate has a higher priority to be executed. Take the JPred program in listing 8.1 for example, the implication relationship is shown in figure 8.1. Compared to this view, the graphical representation of dispatch function decomposes each predicate into a set of atomic predicates and then repeated ones are removed. As shown in figure 8.2, 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.

```
1 class Test {
2   void m(i) {}
3   void m(i) when i==0 || i==1 {}
4   void m(i) when i==0 {}
```

```
5 }
```

Listing 8.1: A JPred program example

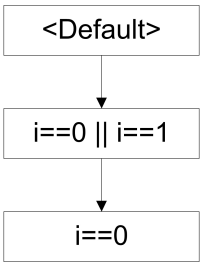

Figure 8.1: JPred predicate implication for method `m()` in listing 8.1

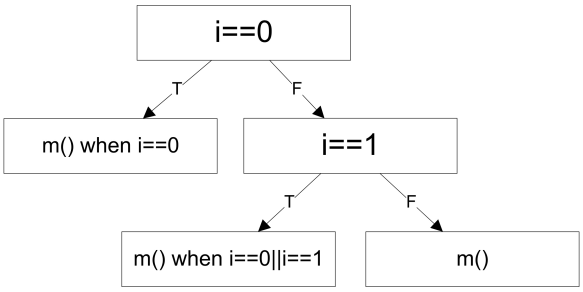

Figure 8.2: Graphical representation of the dispatch function when `m()` in listing 8.1 is called in the AD debugger

CaesarJ [4] is a Java based programming language, which facilitates better modularity and development of reusable components. The components are collaborations of classes, but they can modularize crosscutting features or non-functional concerns. CaesarJ Development Tools (CJDT) extends the Eclipse's JAVA Development Tool (JDT) plug-in with CaesarJ specific features. CJDT combines the crosscutting relationship with class member structure in the *Outline* view. Like the AJDT, the CJDT also uses the Java debugger because CaesarJ programs can be compiled into Java bytecode. Therefore, language-specific features cannot be shown and source locations are lost in some cases.

JAsCo [1] is an advanced AOP language tailored for the component-based field. It makes aspects reusable and provides a strong aspectual composition mechanism for managing combinations of aspects. Besides, it allows to add, change and remove aspects from the system at runtime. JAsCo Development Tools (JAsCoDT) is a tool for editing, running and debugging JAsCo-enabled applications. JAsCoDT provides an *Introspector* which displays the connectors found within the system. It also has a *Joinpoint Lookup* view which is used for statically exploring matched join point of a hook instantiation. In the AD debugger, the *Deployed Attachment* view is similar to the *Introspector* view, attachments can be activated or deactivated by checking or unchecking. Compared to the *Joinpoint Lookup* view, the *Advanced-Dispatching Structure* view can find not only matched type components for a given attachment but also the other way around.

The approaches presented above all target specific general-purpose programming languages of the aspect-oriented or predicate-dispatching paradigms. There is another field

of related work, namely work that aims to provide language-specific debugging support for domain-specific languages (DSLs). Since the ALIA4J approach also can be used to implement domain-specific language [6], I will consider this kind of related work in the future.

The TIDE [25] environment is a generic debugging framework that can be instantiated for new DSLs. While it simplifies the development of a debugger for a new language, it cannot take the complete effort from the language developer. In contrast, my work provides a completely generic solution for any language that is implemented in terms of ALIA4J. Furthermore, TIDE does not specifically support advanced-dispatching features. Nevertheless, it enables a more language-specific user interface while the user interface in my work only provides visualizations of ALIA4J's abstractions.

The IDE Meta-tooling Platform (IMP) [5] is an Eclipse project aiming at providing meta-implementations of typical IDE tools. Examples are a re-usable infrastructure for syntax highlighting, refactoring support, semantic or static analyses, execution and debugging. Their focus is on providing an infrastructure for the IDE integration and the graphical user interface, but not on providing an infrastructure for the runtime part of actual debugger implementations.

Chapter 9

Conclusions & Future Work

The behavior of an advanced-dispatching (AD) program is determined at runtime and it may be totally different with what is read from a source-code fragment. A debugger is an important tool for developers to understand the program behavior. However, problems occur when using conventional debuggers to debug AD programs, like loss of AD-related information, showing transformed code instead of source code, locating source code wrongly.

In order to design and implement a debugger for AD programs, a dedicated debug model has been proposed based on Eaddy's work [13] and it consists of three components. In this model, five AD activities have been classified and each of them introduces new types of faults that are absent in the base program. Based on the activity classification and the new fault types, seven properties that an ideal AD debugger should support are proposed. They are idempotence, debug obliviousness, debug intimacy, dynamism, attachment introduction, affection exploration and locating. This thesis has discussed how to meet these properties in the developing AD debuger and come to a decision extending the Java Platform Debugger Architecture (JPDA). Extensions to the JPDA concentrate on the debuggee side and the debugger side. On the debuggee side, the original workflow of the execution environment NOIRIn has been extended with step that provides support for the debugger side to inspect desired information. Besides, many functions are added in order to help the debugger accomplish its task easier, such as data adaptation. On the debugger side, a dedicated Advanced-dispatch Debug Interface (ADDI) has been implemented. Each AD-specific entity is a wrapper of an *ObjectReference* which stands for a mirror of an object in the debuggee program. AD-specific functions are realised by forwarding calls to methods of wrapped objects. Based on the ADDI, three views has been implemented for presenting obtained

information, namely the *Advanced Dispatch* view, the *Advanced-Dispatching Structure* view and the *Deployed Attachments* view.

Then this thesis showed an example using the AD debugger to debug a buggy AspectJ program and we have witnessed that it explicitly gives some AD-specific information where the bug lies. Discussions are given to evaluate how well each property has been met in the implementation and it draws a conclusion that four are fully supported, two are partially supported and one not supported. The *locating* is not supported because this development lacks of output of compiler. Finally, the thesis designed a feasible solution for supporting locating.

As mentioned in previous chapters, many desired functionalities are simplified and need to be improved. Further work will focus on three key areas: Firstly, the AD debugger is currently only suitable for debugging limited AD features, like only mirrors of *instanceOf* and *boolean* atomic predicates are implemented. Corresponding `ADMirror`s should be reified in order to fit all AD features. Secondly, add an event module to the Advanced-Dispatching Debug Interface (ADDI) to handle AD-specific event, such as pointcut evaluation. Thirdly, cooperate with compiler and apply the approach discussed in chapter 7 in the debugger.

Bibliography

[1] Jasco. `http://ssel.vub.ac.be/jasco/index.html`, 2005.

[2] Java platform debugger architecture (JPDA). `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`, 2009.

[3] Aspectj development tools. `http://www.eclipse.org/ajdt/`, 2010.

[4] Caesarj project. `http://caesarj.org/index.php/Caesar/HomePage`, 2010.

[5] The IDE Meta-tooling Platform. `http://eclipse.org/imp/`, 2010.

[6] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.

[7] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of VMIL*, New York, NY, USA, 2007. ACM.

[8] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35, 1986.

[9] C. Chambers. Object-oriented multi-methods in cecil. In *Proceedings of ECOOP*. Springer Verlag, 1992.

[10] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3), 2006.

[11] W. De Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of AOSD*. ACM, 2009.

[12] A. de Roo, M. Hendriks, W. Havinga, P. Dürr, and L. Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *Proceedings of WASDeTT*, 2008.

[13] M. Eaddy, A. V. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. 2007.

[14] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP*. Springer Verlag, 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[16] B. L. K. Kenneth Slonneger. *Syntax and Semantics of Programming Languages, A Laboratory Based Approach*. Addison-Wesley Publishing, 1995.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 327–353, Berlin/Heidelberg, Germany, 2001. Springer Verlag.

[18] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*. Springer Verlag, 2003.

[19] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 31(2), 2009.

[20] B. Y. Min Xie. A study of the effect of imperfect debugging on software development cost. In *IEEE Transactions on Software Engineering*, pages 471 – 473, 2003.

[21] G. Pothier and E. Tanter. Extending omniscient debugging to support aspect-oriented programming. In *In Proceedings of SAC*. ACM, 2008.

[22] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, pages 535–552, Montreal, Canada, Oct. 2007. ACM Press. ACM SIGPLAN Notices , 42(10).

[23] J. B. Rosenberg. *How debugger works: Algorithms, Data structures and Architecture*. Wiley, 1996.

[24] A. Sewe, C. Bockisch, and M. Mezini. Redundancy-free residual dispatch. In *Proceedings of FOAL*. ACM, 2008.

[25] M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework — tool demonstration —. *Electron. Notes Theor. Comput. Sci.*, 141(4):161–165, 2005.