Developing A Generic Debugger for Advanced-Dispatching Languages[☆]

Haihan Yin[a], Christoph Bockisch[a],

[a]*Software Engineering group, University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands*

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 sites 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. In this paper, we present a debugger which extends a traditional Java debugger with the ability of debugging an advanced-dispatching language constructs and a user interface for inspecting this.

Keywords:
2000 MSC: 68N15 Programming Languages,
2000 MSC: 68N19 Other programming techniques,
2000 MSC: 68N20 Compilers and interpreters, Debugger, Advanced-dispatching, Eclipse plug-in

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 [1] and predicate dispatching [2], pointcut-advice [3]—a particular flavor of aspect-oriented programming (AOP)—, or context-oriented programming [4]. Because they are beyond the traditional *receiver-type polymorphism* dispatch mechanism, we call these new mechanisms *advanced-dispatching* (AD).

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.

[☆]This work is partly funded by a CSC Scholarship (No.2008613009).

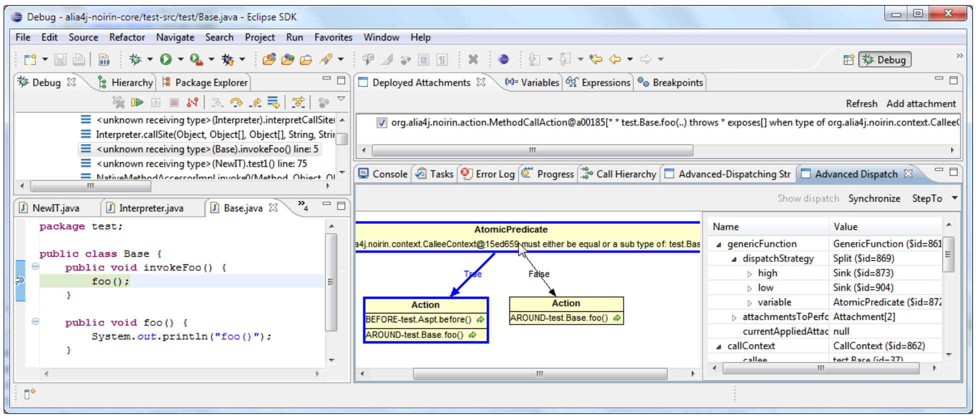
Email addresses: `h.yin@student.utwente.nl` (Haihan Yin), `c.m.bockisch@cs.utwente.nl` (Christoph Bockisch)

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 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 [5][1].

With *advanced-dispatching* we refer 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.

In our work, we have especially investigated the advanced-dispatching languages JPred [6], MultiJava [7], AspectJ [8], Compose* [9], CaesarJ [10], JAsCo [11], and ConSpec [12]. While high-quality tools, such as the AspectJ Development Tools [13], exist to visualize the static structure of programs written in these languages, little to no support is provided related to dynamic language features. Lack of supporting tools significantly hampers the popularity of otherwise valuable new programming languages, thus dedicated tool support is required. In a previous position paper [14] we have already outlined this claim as well as a an implementation of language independent tools for advanced-dispatching languages based on the ALIA4J architecture.

This paper specifically focuses on a generic debugger tool which is aware of advanced-dispatching concepts and allows to inspect such program elements at runtime. This debugger is realized based on a language-independent intermediate representation and execution environment of advanced dispatching and can therefore support developers using all the above-mentioned languages. Instead of implementing it from scratch, we chose to extend the Java debugger of Eclipse with AD-specific features. Figure 1 shows how our debugger graphically represents a dispatch at which the execution is currently suspended. The provided views are discussed in detail in section 5.



```
<unknown receiving type>(Interpreter).interpretCallSite
Interpreter.callSite(Object, Object[], Object[], String, Strin
<unknown receiving type>(Base).invokeFoo() line: 5
<unknown receiving type>(NewIT).test1() line: 75
NativeMethodAccessorImpl.invoke0(Method, Object, Ol
```

```
package test;

public class Base {
    public void invokeFoo() {
        foo();
    }

    public void foo() {
        System.out.println("foo()");
    }
```

org.alia4j.noirin.action.MethodCallAction@a00185[* * test.Base.foo(..) throws * exposes[] when type of org.alia4j.noirin.context.Callee(

a4j.noirin.context.CalleeContext@15ed659 must either be equal or a sub type of: test.Bas

BEFORE-test.Aspt.before()
AROUND-test.Base.foo()

AROUND-test.Base.foo()

Name	Value
genericFunction	GenericFunction ($id=861
dispatchStrategy	Split ($id=869)
high	Sink ($id=873)
low	Sink ($id=904)
variable	AtomicPredicate ($id=87
attachmentsToPerfc	Attachment[2]
currentAppliedAttac	null
callContext	CallContext ($id=862)
callee	test.Base (id=37)

Figure 1: A snapshot of the debugger for advanced-dispatching programs

[1]Some details presented in [5] are outdated, but it may nevertheless act as an introduction to the basic concepts.

The remainder of this paper is structured as follows. In section 2, we present a deeper analysis of the problems when debugging advanced-dispatching programs as well as features we claim are required from a good debugger for such programs. Section 3 outlines the approach of our work, followed by a section presenting the required technical background (section 4). Next we discuss the three corner stones of our debugger, namely the user interface (section 5), the extension to the execution environment (section 6), and the extended debug model (section 7). This paper closes by presenting an example of debugging an AspectJ program with the developed debugger in section 9, related work in section 10, and some conclusions and future work in section 11.

2. Problem Analysis and Goals

The lack of tool support for advanced-dispatching languages is based in the technique typically applied when implementing such languages: in all investigated language implementations, a compiler translates source code from the new language to code—most frequently an intermediate representation like Java bytecode [15] or the .NET common intermediate language [16]—of the base language. After the compilation all source code is mapped to the same intermediate language whose abstractions reflect the module concepts of the base language, but not those of the new language. This compilation strategy is often called *weaving* and may entail that instructions are removed, merged, duplicated, reordered, or interleaved. This is discussed, for instance, by Avgustinov et al. [17] in the context of the AspectBench Compiler for the AspectJ language.

As the new language concepts are compiled to conventional code, the tooling, e.g., the debugger, of the base language can also be used for compiled programs written in the new languages. However, as it is not aware of the new language concepts, the base tooling may not always be adequate. Take the Java debugger for example: When the debugged program is intercepted, the Java debugger offers the possibility to locate the source code that was compiled to the Java bytecode instruction whose execution was intercepted. This feature is supported by the Java bytecode format that stores for every bytecode file the name of the source-code file from which it was compiled, as well as a mapping from instructions to lines in the source file.

When this debugger is used, e.g., for AspectJ programs, problems like the inability of locating source code are easily encountered because programmers debug the transformed, woven code. After the weaving phase, the assumption—which is manifested in the Java bytecode format—that all code of a class is compiled from a single source file may no longer hold.

As a starting point for our investigations, we use Eaddy et al.'s [18] categorization of the debugging problems for aspect-oriented programs into the *code-location problem* and the *data-value problem*. Since aspect-orientation is one special case of advanced-dispatching, we generalize these problems and conclude that existing debuggers are not sufficient for AD languages mainly for the following two reasons:

Code-location problem One important purpose of debugging is locating errors in the source code. As explained previously, the compiled code may undergo a series of transformations during which the source-location information is not maintained by current compilers. This lack of traceability 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*.

Data-value problem Dispatch results in the execution of one of multiple alternative functionalities; which alternative is chosen is evaluated according to the runtime context in which dispatch site is executed. While a conventional debugger will eventually show which functionality is executed, it is unable to present to the developer the reasoning behind choosing this functionality. Take debugging an AspectJ program in Eclipse for example: the debugger does not provide aspect-related information like runtime states accessed during dispatch, e.g., in terms of dynamic pointcut designators.

Inspired by the work of Eaddy et al. and by our own observations when using advanced-dispatching languages, we concentrate on solving the data-value problem and formulate the following debugger capabilities that are desirable for an advanced-dispatching debugger:

1. Inspection of runtime state of the executing AD program.
2. Inspection of program composition and control flow.
3. Inspection of the evaluation result of functions over the runtime state to select alternative meanings.
4. Description the relationship between AD entities.
5. Deployment and undeployment AD features at runtime.

3. A Debugger for Advanced-Dispatching Languages

To enable debugging, the debugger front-end must communicate with execution of the debugged program. Furthermore, the source code must be traceable, i.e., it must be possible to deduce the source code that has lead to a certain execution. Therefore, we base the debugger presented here on our previous work [19, 5], the ALIA4J approach for implementing advanced-dispatching languages. It provides a uniform representation of programs written in different advanced-dispatching languages in order to enable the reuse of implementations between these languages, and it embodies the execution semantics for advanced-dispatching in a language-independent way. Furthermore, its representation of advanced-dispatching stays first-class during the execution. Therefore, we present a debugger for advanced-dispatching languages, which is based on the ALIA4J approach, in order to improve the tooling landscape for multiple existing and future advanced-dispatching languages at once. Basically, our work will allow the developer to debug the ALIA4J representation of advanced-dispatching instead of woven bytecode as in traditional approaches. The ALIA4J representation is much closer to the original source code than the woven bytecode, and preserves AD-related debug information, thus increasing the traceability.

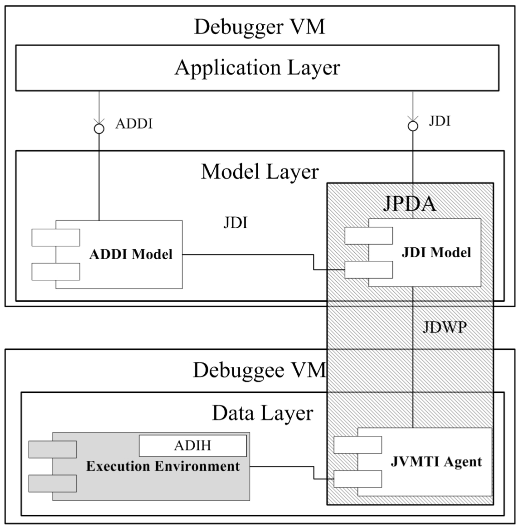


Figure 2: Structure of the AD language Debugger Architecture

We have developed the *AD language Debugger Architecture* (ADDA) as an extension to the *Java Platform Debugger Architecture* (JPDA) [20] based on ALIA4J's representation of advanced-dispatching. Figure 2 shows the overall structure of the ADDA. The *Execution Environment* component 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 JPDA and the AD-related data is represented by the *AD Debug Interface* (ADDI) Model. A debugger front-end for advanced-dispatching can connect to the ADDI and the JDI in order to debug the execution of advanced-dispatching and of standard Java in a program.

The front-end of our debugger is integrated into the Eclipse IDE, although any IDE with a comparable infrastructure would also be applicable. Our 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 requirements motivated in section 2. In some cases, we have also changed the behavior of existing debugger views to adapt to AD features.

After presenting some background on ALIA4J, the JPDA and the Eclipse debugger in the following section, we will present our AD language debugger. In particular, we will present the user interface extensions in section 5. In section 6 we will shortly present the required extensions to an ALIA4J-based execution environment for communicating with the debugger. And in section 7 we will discuss our extensions to Java debugger architecture that reflect the advanced-dispatching extensions to languages and the user interface.

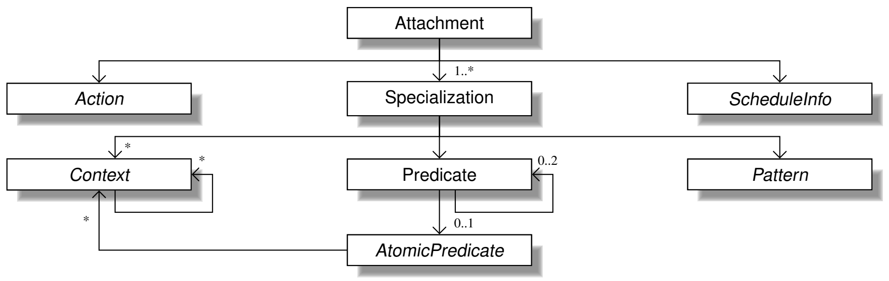


Figure 3: Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

4. Background

4.1. ALIA4J

ALIA4J [19] has two main components: The first component is the *Language-Independent Advanced-dispatching Meta-model* (LIAM) for expressing advanced-dispatching declarations. 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 depend on values in the dynamic *context* of the dispatch. LIAM has to be *refined* with the concrete advanced-dispatching concepts of actual programming languages that are supposed to be mapped to LIAM.

Figure 3 shows the eight meta-entities of LIAM, discussed in detail in [19, Chapter 3.2], which capture the core concepts underlying the various dispatching mechanisms. The meta-entities *Action*, *AtomicPredicate* and *Context* can be refined to concrete concepts.

In short, an *attachment* corresponds to a unit of dispatch declaration, roughly corresponding to a pointcut-advice pair or to a predicate method. *Action* specifies functionality that may be executed as the result of dispatch (e.g., the body of an advice or predicate method). *Specialization* defines static and dynamic properties of state on which dispatch depends. *Pattern* specifies syntactic and lexical properties of the dispatch site; *predicate* and *atomic predicate* entities model conditions on the dynamic state a dispatch depends on. *Context* entities model access to values in the context of a dispatch, like the called object or argument values when the dispatch site is a call to a instance method. Finally, the *schedule information* models partial ordering of the action contributed to a dispatch result in relation to actions contributed by other attachments.

The actual intermediate representation of a concrete program, in turn, is a model conforming to the meta-model refinement for that language—we simplifyingly call these models *LIAM models*. Code of the program *not* using advanced dispatching mechanisms is represented in its conventional Java bytecode form.

We have validated the expressiveness of LIAM by refining it with the concrete concepts of different existing programming languages [19], namely AspectJ [8], Compose* [9], CaesarJ [10], and JAsCo [11]; refinements for MultiJava [21], JPred [6], and ConSpec [12] also exist and can be downloaded from our website[2] but are not otherwise published. For the languages AspectJ, ConSpec, and rudimentarily for Compose* we have implemented an automatic translator from source to a model according to refined LIAM.

[2] `http://www.alia4j.org`

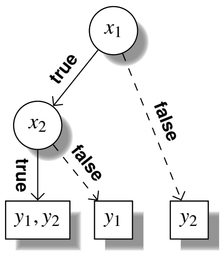


Figure 4: A dispatch function's evaluation strategy.

The second component of ALIA4J is the *Framework for Implementing Advanced-dispatching Languages* (FIAL), a framework for execution environments that allow to deploy and undeploy LIAM dispatch declarations. FIAL implements common components and work flows required to implement execution environments based on a Java Virtual Machine (JVM) for executing LIAM models. Most importantly for the purpose of this paper, it defines how to derive an execution strategy *per dispatch site* that considers all LIAM-based dispatch declarations present in the program.

The execution strategy consists of the so-called dispatch function that characterizes which actions should be executed as the result of the dispatch in a given program state. This function is represented as a binary decision diagram (BDD) [22], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with the actions to be executed. For each possible result of dispatch, the BDD has one leaf node, representing an alternative result of the dispatch, i.e., which actions are to execute and in which order[3]. Figure 4 shows an example of such a BDD with the atomic predicates x_1 and x_2 and the actions y_1 and y_2. For a detailed explanation of this model, we refer the reader to [23].

Importantly for the present paper, the dispatch models and all LIAM models stay first-class during the execution of a program and can therefore easily support dynamic features of an IDE, like debugging, profiling, or testing. We have implemented several execution environments based on FIAL which apply optimizations to a different degree. All extensions to the execution environment that have been made in the course of developing the debugger presented in this paper, are made to our portable but non-optimizing execution environment called NOIRIn. It is future work, to enable similar support also in the optimizing execution environments.

4.2. The Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) [20] is presented in the striped box of figure 2. It defines a system consisting of 2 layers and the communication between them. From the bottom up, this are the Java Virtual Machine Tool Interface (JVMTI) layer, the Java Debug Wire Protocol (JDWP) for communication and the Java Debug Interface (JDI) layer. On top of the JDI, the debugger user interface layer is implemented. JVMTI is a native interface of the JVM. It allows to check runtime states of a program, set call-back functions for events like reaching a breakpoint, and control some environment variables. JDWP is the protocol used for the communication between the debugger—which is written in Java thus executed by a JVM—and the debuggee JVM. JDI is a mirror-based, reflective interface. The mirror mechanism maps

[3]Nested execution of, e.g., around advice in AspectJ that use the *proceed* keyword, is also supported by the execution strategy.

all data including values, types, fields, methods, events, states and resources on the debuggee JVM into mirror objects. For instance, loaded classes are mapped to *ReferenceType* mirrors, objects are mapped to *ObjectReference* mirrors, etc. Accessing fields or invoking methods of an object existing in the debuggee JVM can be performed by the debugger in the reflective way, using the mirrors.

4.3. 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 5. 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.

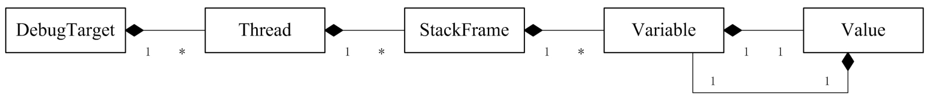


Figure 5: Eclipse platform debug model

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 requires some more interaction between the debugger and the debuggee. The developer also has different options to control the further execution of the suspended program. He/she can, for example, instruct the debugger to “Step Into” or “Step Over” the computation that is currently suspended, to “Resume” the execution or to “Terminate” the debugged process.

The Java Development Tools (JDT) of Eclipse implement the platform debug model for Java based on the JDI standard.

5. User-Interface for the AD debugger

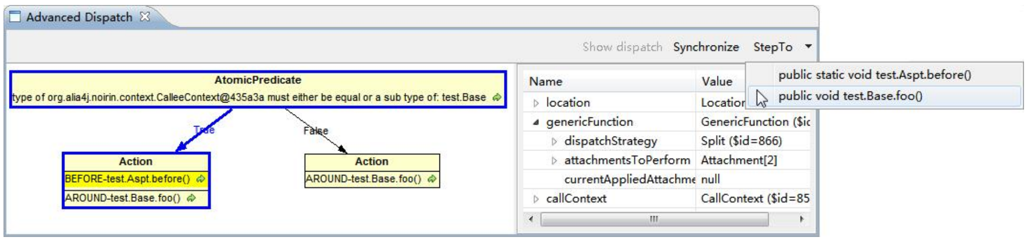
The application layer presented in figure 2 consists of several views like the *Variables* and *Expressions* views which are provided by the default Java debugger in Eclipse. The AD debugger adds three views, namely the *Advanced Dispatch* view, the *Advanced-Dispatching Structure* view and the *Deployed 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.

In order to make our debugger better understood, we give a code example written in AspectJ in listing 1. For simplicity, we omit the base program with the call to *Base.foo()*. Listing 1 declares an aspect with a *before* advice which is executed when *Base.foo()* is called and the callee is an instance of class *Base*. Suppose the program is currently suspended at the point where the next instruction calls *Base.foo()*, we introduce each view in this scenario in the following paragraphs.

```
public aspect Aspt {
  before() : call(* Base.foo()) && target(Base) {
    System.out.println("advice()");
  }
}
```

Listing 1: A code example of an aspect

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 be executed, etc. All values are presented textually in a tree viewer. The execution strategy with the dispatch function, which is represented as a BDD in ALIA4J, is additionally presented graphically. It is a special form of a branching program, for which a graphical representation should be intuitive to the developer.



type of org.alia4j.noirin.context.CalleeContext@435a3a must either be equal or a sub type of: test.Base

BEFORE-test.Aspt.before()
AROUND-test.Base.foo()

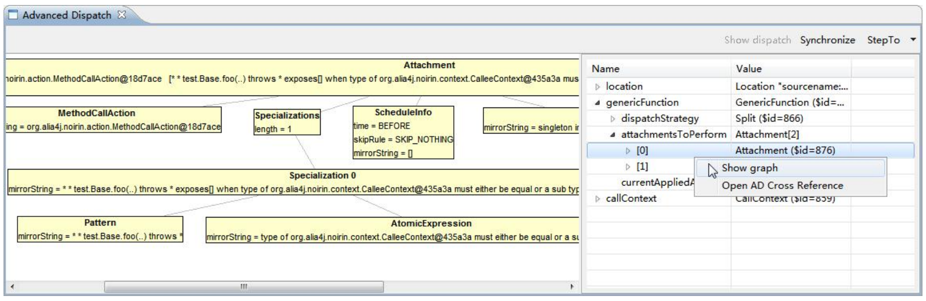
AROUND-test.Base.foo()

Name	Value
location	Locatior
genericFunction	GenericFunction ($ic
dispatchStrategy	Split ($id=866)
attachmentsToPerform	Attachment[2]
currentAppliedAttachme	null
callContext	CallContext ($id=85

Figure 6: 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 figure 6. The left window frame gives a graphical representation of the execution strategy for the next dispatch which is *Base.foo()* in this example. The root node represents an *AtomicPredicate* which requires that the type of the callee should be *Base* or a subtype. 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. At the top right of the view, a "StepTo" button is provided for suspension the program at any performing action. For instance, the entrance of the method *Base.foo()* can be chosen to be the next suspending point so that details of aspect activities are ignored.

Considering the complexity and importance of *Attachment*s, the Advanced Dispatch view also provides a graphical representation for *Attachments*. As shown in figure 7, the user can right click an *Attachment* in the view's tree and select "Show graph".

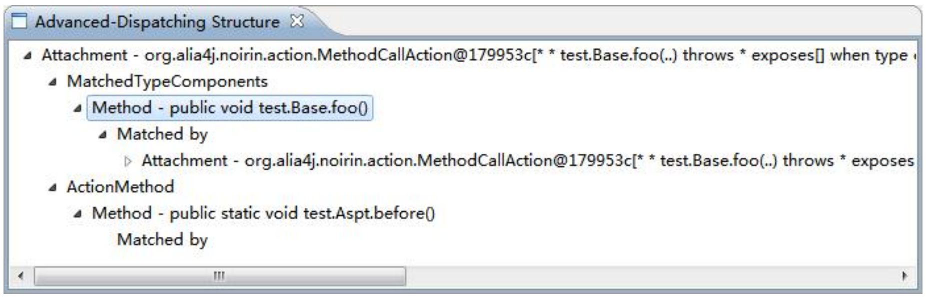


```
Attachment
ioirin.action.MethodCallAction@18d7ace   [* * test.Base.foo(..) throws * exposes[] when type of org.alia4j.noirin.context.CalleeContext@435a3a mus

MethodCallAction
ing = org.alia4j.noirin.action.MethodCallAction@18d7ace

Specializations
length = 1

ScheduleInfo
time = BEFORE
skipRule = SKIP_NOTHING
mirrorString = []

mirrorString = singleton ir

Specialization 0
mirrorString = * * test.Base.foo(..) throws * exposes[] when type of org.alia4j.noirin.context.CalleeContext@435a3a must either be equal or a sub typ

Pattern
mirrorString = * * test.Base.foo(..) throws *

AtomicExpression
mirrorString = type of org.alia4j.noirin.context.CalleeContext@435a3a must either be equal or a su
```

Name	Value
location	Location "sourcename:...
genericFunction	GenericFunction ($id=...
dispatchStrategy	Split ($id=866)
attachmentsToPerform	Attachment[2]
[0]	Attachment ($id=876)
[1]	
currentAppliedA	
callContext	CallContext ($id=859)

Figure 7: A snapshot of the Advanced Dispatch view showing the graphical representation for an attachment

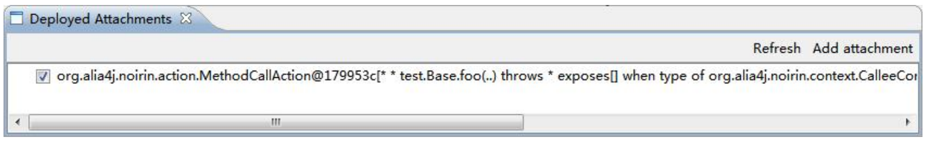
The *Advanced-Dispatching Structure* view assists the developer to explore *static* AD information within the project scope, i.e., all dispatching declarations in the program as well as all the generic functions in the program. This view also allows the developer to navigate from a dispatching declaration to the affected generic functions and vice versa. In this example, the method *Base.foo()* may be matched by multiple *Attachment*s. However, this information can not be shown in the *Advanced Dispatch* view in an explicit way. Figure 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*.



```
Attachment - org.alia4j.noirin.action.MethodCallAction@179953c[* * test.Base.foo(..) throws * exposes[] when type
  MatchedTypeComponents
    Method - public void test.Base.foo()
      Matched by
        Attachment - org.alia4j.noirin.action.MethodCallAction@179953c[* * test.Base.foo(..) throws * exposes
  ActionMethod
    Method - public static void test.Aspt.before()
      Matched by
```

Figure 8: A snapshot of the Advanced-Dispatching Structure 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 9.



```
org.alia4j.noirin.action.MethodCallAction@179953c[* * test.Base.foo(..) throws * exposes[] when type of org.alia4j.noirin.context.CalleeCo
```

Figure 9: A snapshot of the Deployed Attachments view

6. Extensions to the execution environment

As mentioned in section 4.1, different implementations of execution environments exist in ALIA4J but in this paper we focus on the interpreting one called NOIRIn. The following 2 subsections describe the work flow of executing dispatch in NOIRIn. Furthermore, we will show how we have extended this work flow and added new services to enable debugging.

6.1. Debugging-Enabled Work Flow of Executing Advanced-Dispatching

Figure 10 presents the work flow of deploying an advanced-dispatching declaration and of executing advanced dispatch in NOIRIn. The figure also shows the additional steps in the work flow for supporting debugging in NOIRIn. The following list describes each step in the grey box which represents the original work flow.

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

2. After an attachment is sent to NOIRIn, it 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 logic of each dispatch site. After the preparation of the previous two steps, the actual application program starts.

3. When a dispatch site is encountered, e.g., a method call is invoked in the scope of user written code, NOIRIn intercepts this call and performs the following steps to execute the dispatch.

4. The context relating to that call, like the line number in which this call happens, the declaring class of the callee, etc., is collected by NOIRIn. According to the call context, NOIRIn finds the dispatch's execution strategy.

5. Based on the context values, the dispatch function is evaluated to decide which actions to execute.

6. 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.

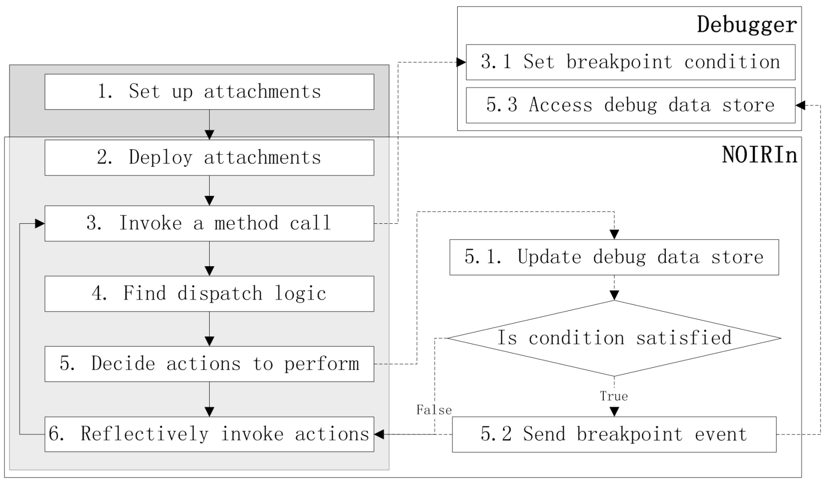

Debugger
3.1 Set breakpoint condition
5.3 Access debug data store
NOIRIn
1. Set up attachments
2. Deploy attachments
3. Invoke a method call
4. Find dispatch logic
5. Decide actions to perform
6. Reflectively invoke actions
5.1. Update debug data store
Is condition satisfied
True
False
5.2 Send breakpoint event

Figure 10: The work flow of intercepting a method call extended with ability supporting debugging in NOIRIn

Additional to the steps presented in figure 10 we have extend NOIRIn to support debugging. First, all needed debugging data, like the call context, the dispatch site, etc., are stored in a singleton store from which all needed data can be retrieved. Every time when the applicable actions are determined at step 5, NOIRIn writes new data into the store (5.1). Second, a conditionally executed statement is added right after the store is updated. We refer to this instruction as the *breakpoint shadow*. The debugger can set a breakpoint here to suspend the execution of NOIRIn just before the execution of the dispatch. The statement is executed conditionally and the condition is set at the time when the method is called at step 3 and the specific condition is configured according to the call context (3.1). When the program runs to the breakpoint shadow, it sends a breakpoint event to the debugger (5.2). Then, the program is suspended and the debugger accesses the store (5.3) until it asks the debuggee VM to resume.

6.2. Services for Enabling Debugging

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. Therefore we implemented the Advanced-Dispatching Information Helpers (ADIH) in NOIRIn such that each task can be performed just by one reflective method invocations where actually a series of methods from the standard NOIRIn API must be called. As the remote and reflective invocation of methods requires marshalling arguments and handling exceptions in a special way, the ADIH simplifies the implementation of the debugger front-end.

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 class members according to a pattern of an attachment. NOIRIn only provides an interface for reading the pattern, the function retrieving class members was added for usage by the ADIH. 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 finding the related *Attachment* according to an *AttachedAction* has been implemented.

To summarise, information helpers simplify the process of performing tasks at the debugger side and add functionalities supporting debugging which have not been available in the original NOIRIn.

7. Advanced-dispatching debug interface

The Advanced-dispatching Debug Interface (ADDI) extends the Java Debug 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 11. The light grey parts are entities defined in JDI. The dark grey parts are entities defined in JDI but extended with advanced-dispatching related features. The white parts are new entities introduced in ADDI. The rest of this section describes each entity which is relevant to advanced-dispatching.

TypeComponent reifies a class member, i.e., a *Field* or a *Method*. 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 function providing a list of matching attachments.

Field reifies a field. It extends *TypeComponent* and further distinguishes the related dispatch sites into field reading and field writing. Therefore, it is extended with functions providing a list of attachments matching its reading and writing respectively.

ClassType reifies a class. It is extended with a function providing a list of matching attachments of its members.

ADMirror reifies an advanced-dispatching related entity. It wraps an *ObjectReference* which is the mirror of an actual object in the debuggee program. Mostly, the “mirror-wrapper” *ADMirror* provides advanced-dispatching 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 the information helpers.

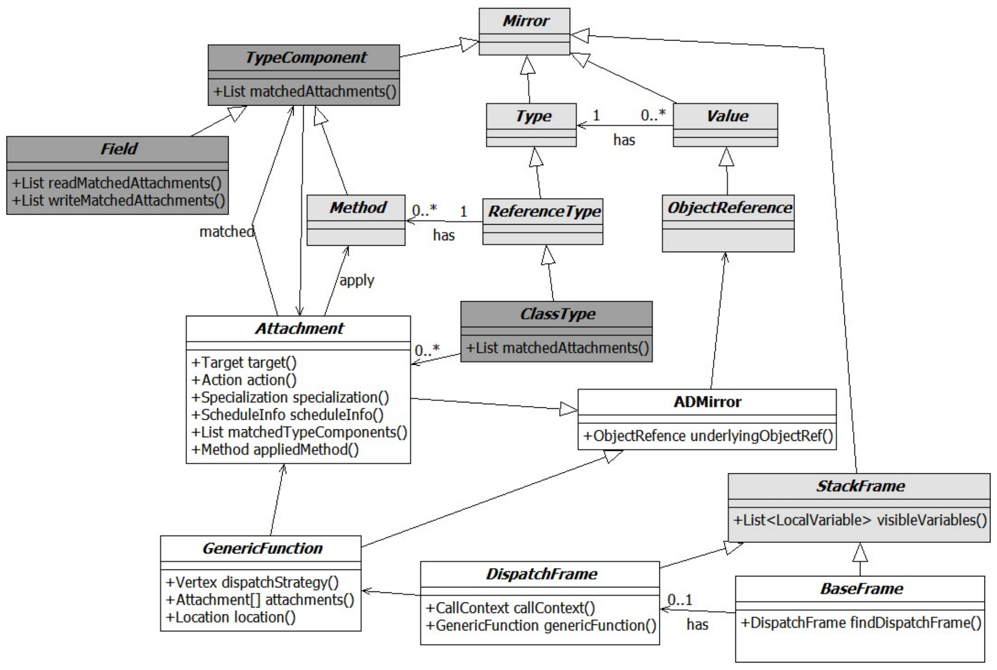

Mirror
TypeComponent
+List matchedAttachments()
Field
+List readMatchedAttachments()
+List writeMatchedAttachments()
Type
Value
1
0..*
has
Method
ReferenceType
ObjectReference
0..*
1
has
matched
apply
Attachment
+Target target()
+Action action()
+Specialization specialization()
+ScheduleInfo scheduleInfo()
+List matchedTypeComponents()
+Method appliedMethod()
ClassType
+List matchedAttachments()
0..*
ADMirror
+ObjectRefence underlyingObjectRef()
StackFrame
+List<LocalVariable> visibleVariables()
GenericFunction
+Vertex dispatchStrategy()
+Attachment[] attachments()
+Location location()
DispatchFrame
+CallContext callContext()
+GenericFunction genericFunction()
0..1
has
BaseFrame
+DispatchFrame findDispatchFrame()

Figure 11: Simplified UML class diagram of ADDI

Attachment reifies an attachment. It provides interfaces for accessing components of *Attachment*. If the *Specialization* is matched, the *Action* will be invoked at the time that the *ScheduleInfo* specifies. Besides, it can provide a list of *TypeComponents* which match a pattern of the attachment and a mirror of the actual method that its *Action* implies.

BaseFrame and DispatchFrame reify different stack frames according to where they resides. *BaseFrame* denotes a frame which is not in the infrastructural code of ALIA4J. *DispatchFrame* is a specific frame which resides at the breakpoint shadow introduced in section 6.1 where the dispatch strategy has been computed but not yet performed. According to the execution flow in the execution environment, the *BaseFrame* is always accompanied with a *DispatchFrame*.

GenericFunction reifies a dispatch. The generic function is a unique identification, e.g., signature, which may be shared by multiple methods. A generic function has a list of attached actions. Besides, it has a dispatch strategy which describes which of them are executed in which order at a site performing this dispatch.

8. Lessons Learnt

8.1. Learning About the Internal Functionality of Eclipse Plug-Ins

Though a tutorial about building an Eclipse plug-in is easy to obtain in the Internet, lacking of documentation makes realizing a desired functionality from scratch troublesome. This is also discussed in [24]. Fortunately, there are a lot of open-source plug-ins with high code quality available which can act as examples. To implement a functionality for a plug-in, our experiences may helps.

1. First, decide which functionality is needed and think of a similar functionality provided by other plug-ins.

2. Second, you need to find the implementation of the functionality which you have located in the user interface. Eclipse provides a handy tool to support you in finding relevant code. After selecting the UI element, like view, editor, etc., press "Shift+Alt+F1" to invoke the Plug-in Spy in order to open the related source file.

8.2. Extending Existing Plug-Ins

We developed our debugger based on the default Java debugger in Eclipse because the functionalities for debugging base programs are the same. The core part of the Java debugger is the plug-in *org.eclipse.jdt.debug* which gives a JDI implementation. Our debugger not only *uses* the JDI implementation but also *extends* its behavior and structure. For example, when the debugger requires to disconnect, it calls *MirrorImpl.disconnectVM()* and carries out a list of disconnecting tasks. Our debugger needs to add another task into this list in order to release some resources before disconnection. Simply importing this plug-in into the library of our project does not help. Thus, we import this plug-in into a source folder.

Because the imported Java debugger plug-in code is a standard and we want to plug, unplug and maintain the extended source code easily in the future, we put all extended code in another source folders and leave the imported code intact. We modify the behavior and structure without touching the source code using AspectJ. In order to organize extended files well, they are in packages whose names are similar to the names of the packages where they are extended. For example, the imported class *StackFrameImpl* is put in the package "org.eclipse.jdi.internal". We put the extended class *BaseFrameImpl* which inherits *StackFrameImpl* in the package "extended.jdi.internal". With this naming convention, the developer can easily find out the correspondence between the imported and the extended files.

9. A debugging example

To demonstrate the usefulness of the presented debugger, we present a walkthrough of one debugging session. 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 by ten percents. Another type is used when a customer has obtained enough credits from previous shoppings, he/she can get one euro bonus then. If two promotion types are applicable, the price of the item is first reduced by the constant bonus and then cut down by 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, the aspect in listing 2 is used for handling special prices. The first advice from line 4 - 6 reduces the price of an item which is on sale according to the first promotion type. The second advice from line 7 - 9 subtracts the bonus.

```
1 public aspect SpecialPrice {
2   pointcut itemGetPrice(Item i) :
    call(* Item.getPrice()) && target(i) && !within(SpecialPrice);
  before(Item i) : itemGetPrice(i) && if(i.isOnSale()) {
    i.setPrice((float) (i.getPrice() * 0.9));
  }
  before(Item i) : itemGetPrice(i) && if(i.isBonus()) {
    i.setPrice((float) (i.getPrice() − 1));
  }
}
```

Listing 2: A code example of an aspect

Running the program in listing 3, 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 3: Main

The following list shows the process how to use the AD debugger to find the bug.

1. Set a breakpoint at line 7 in listing 3 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 the "Show dispatch" button.
4. The dispatch function is shown in the view as in figure 12. 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 the requirements. The bug is found and developer can reverse their literal order to fix this bug.

Following the classification by Eaddy et al. [18], 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 the faults become less obvious.

10. Related Work

Eaddy et al. [18] implemented Wicca which is a dynamic AOP system for C# applications that performs source weaving at runtime. The source code used in debugging is the woven source

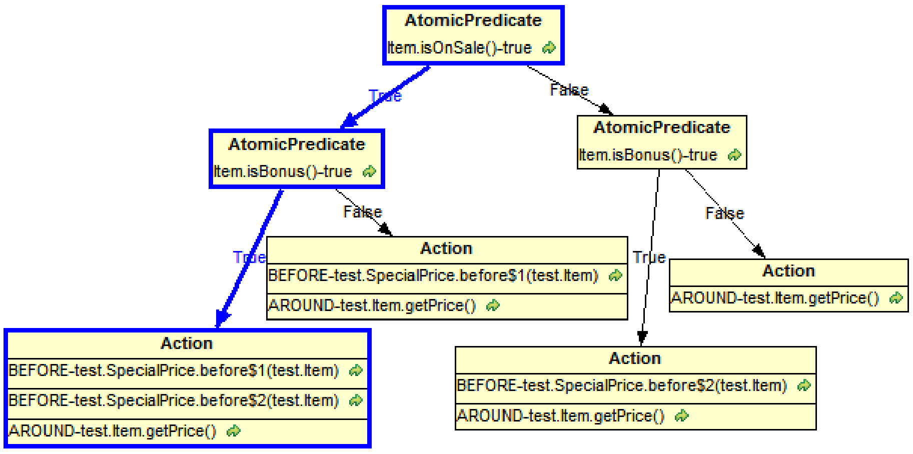

AtomicPredicate
Item.isOnSale()-true
True
False
AtomicPredicate
Item.isBonus()-true
False
AtomicPredicate
Item.isBonus()-true
False
True
True
Action
BEFORE-test.SpecialPrice.before$1(test.Item)
AROUND-test.Item.getPrice()
Action
AROUND-test.Item.getPrice()
Action
BEFORE-test.SpecialPrice.before$1(test.Item)
BEFORE-test.SpecialPrice.before$2(test.Item)
AROUND-test.Item.getPrice()
Action
BEFORE-test.SpecialPrice.before$2(test.Item)
AROUND-test.Item.getPrice()

Figure 12: The dispatch function when Item.getPrice() is called in listing 3

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, or action.

The AspectJ Development Tools (AJDT) [13] enable the Eclipse platform to build, edit, and debug AspectJ programs. It provides a lot of features decreasing the effort in understanding and coding AspectJ programs. The *Aspect Visualiser* and *Cross References* view are most representative. The *Aspect Visualiser* is used to visualize how aspects are affecting classes in a project in a “bars and stripes” style representation. The *Cross References* view is used in the AJDT to show AspectJ crosscutting information, such as which advice a Java method is affected by advice. However, the debugger in the AJDT is problematic because the conventional Java debugger is used.

De Borger et al. [25] found the pointcut-advice models of Java-based AO-technologies, such as AspectJ, JBoss AOP, and Spring AOP to be very similar and thus defined a common debugging interface: the Aspect-Java Debugging Interface (AJDI). The AJDI aggregates JDI mirrors and the information about aspect related structure and behavior. They create events for dynamic AOP and events related to join points and advices by transforming AJDI breakpoints into JDI breakpoints. Based on the AJDI, they built the 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 paper. These other approaches can be used as auxiliary approaches to understand program behavior or structure during debugging.

Pothier et al. [26] implemented an AO debugger based on an open source omniscient Java debugger called TOD [27]. The TOD records all events that occur during the execution of a program and the complete history can be inspected and queried offline after the execution. It is extended to provide *aspect murals* 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 to view which occurrences of join points matched.

The JPred [6] Eclipse plug-in provides a view showing implication relationships between predicates used for methods sharing the same signature. This is shown in terms of a Binary Decision Diagram, similar to ALIA4J's dispatch execution strategy. It indicates that a method with a more specific predicate has higher priority to be executed. Take the JPred program in listing 4 for example, the implication relationship is shown in figure 13. 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 14, 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.

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

Listing 4: A JPred program example

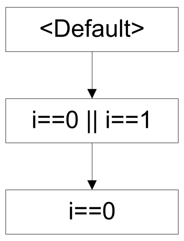


Figure 13: JPred predicate implication for method m() in listing 4

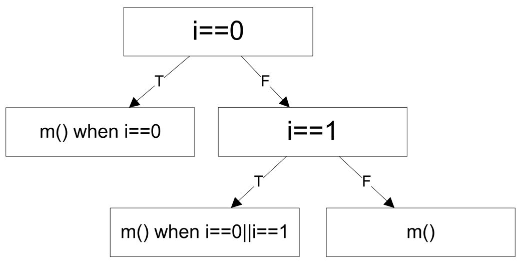


Figure 14: Graphical representation of the dispatch function when m() in listing 4 is called in the AD debugger

CaesarJ [28] 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. The CaesarJ Development Tools (CJDT) extend the Eclipse's Java Development Tools (JDT) plug-in with CaesarJ-specific features. 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.

Also for the ObjectTeams programming language an Eclipse-based IDE exists that enhances the standard JDT Java debugger [29]. The enhancement filters call frames that belong to infrastructural code and adapts the placement of breakpoints respectively the behavior of stepping through the code. While these enhancements hide the details of generated code from the developer, it still falls short in providing additional language-specifi functionality.

JAsCo [30] 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 points 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 [19], we will consider this kind of related work in the future. Here, we only want to mention those related approaches that we are currently aware of.

The TIDE [31] 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, our 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 our work only provides visualizations of ALIA4J's abstractions.

The IDE Meta-tooling Platform (IMP) [32] 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. Nevertheless, we will consider to integrate our work with this project.

Finally, we would like to investigate the Platform-Independent Language (PIL) [33] which is an intermediate layer between DSL source code and the target platform's code. We will compare this language with ALIA4J's intermediate representation.

11. Conclusions and Future Work

In this paper we have presented the Advanced-dispatching Debugger Architecture (ADDA), an accordingly implemented debugger for, and our experiences of implementing it. The ADDA consists of three layers. We have implemented the application layer in terms of three new Eclipse views which offer inspection of advanced-dispatching specific runtime state, present program composition flow, explore affection of advanced-dispatching entities, deploy and undelopy advanced-dispatching features at runtime, etc. In the model layer, we have defined the Advanced-dispatching Debug Interface (ADDI) offering inspection of runtime states of advanced-dispatching entities. And for use in the data layer we adapted the ALIA4J execution environment NOIRIn to be suitable for supporting debugging.

Our future work includes developing an advance-dispatching event model, extending ADDI to language-specific debugger interfaces and supporting debugging in ALIA4J's optimizing execution environments. Furthermore, we will provide additional ALIA4J-based generic IDE tools. We will investigate using Equinox Aspects for modifying the behavior of Eclipse plug-ins.

References

[1] C. Chambers, Object-oriented multi-methods in cecil, in: Proceedings of ECOOP, Springer Verlag, 1992.

[2] M. Ernst, C. Kaplan, C. Chambers, Predicate dispatching: A unified theory of dispatch, in: Proceedings of ECOOP, Springer Verlag, 1998.

[3] H. Masuhara, G. Kiczales, Modeling crosscutting in aspect-oriented mechanisms, in: Proceedings of ECOOP, Springer Verlag, 2003.

[4] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology, March-April 2008, ETH Zurich 7 (2008) 125–151.

[5] C. Bockisch, M. Mezini, A flexible architecture for pointcut-advice language implementations, in: Proceedings of VMIL, ACM, New York, NY, USA, 2007.

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

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

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

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

[10] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, Overview of CaesarJ, in: Transactions on Aspect Oriented Software Development, volume 3880 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin/Heidelberg, Germany, 2006, pp. 135–173.

[11] D. Suvée, W. Vanderperren, V. Jonckers, JAsCo: an aspect-oriented approach tailored for component based software development, in: Proceedings of the International Conference on Aspect-Oriented Software Development, ACM Press, New York, NY, USA, 2003, pp. 21–29.

[12] I. Aktug, K. Naliuka, ConSpec: A formal language for policy specification, in: Proceedings of REM, Elsevier Science Publishers B. V., 2008.

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

[14] C. Bockisch, A. Sewe, Generic IDE support for dispatch-based composition, in: Proceedings of the First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines, volume 564 of *CEUR Workshop Proceedings*, CEUR-WS, 2010.

[15] T. Lindholm, F. Yellin (Eds.), The Java Virtual Machine Specification, Addison-Wesley, 2nd edition, 1999.

[16] Information technology – Common Language Infrastructure (CLI) Partitions I to VI, ISO/IEC, Geneva, Switzerland, iso/iec 23271:2006(e) edition, 2006.

[17] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc : An extensible AspectJ compiler, in: Transactions on Aspect-Oriented Software Development I, number 3880 in Lecture Notes in Computer Science, Springer Verlag, Berlin/Heidelberg, Germany, 2006, pp. 293–334.

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

[19] C. Bockisch, An Efficient and Flexible Implementation of Aspect-Oriented Languages, Ph.D. thesis, Technische Universität Darmstadt, 2009.

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

[21] C. Chambers, W. Chen, Efficient multiple and predicated dispatching, in: Proceedings of OOPSLA, ACM, 1999.

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

[23] A. Sewe, C. Bockisch, M. Mezini, Redundancy-free residual dispatch, in: Proceedings of FOAL, ACM, 2008.

[24] A. W. K. Nick Kirtley, P. Avgeriou, Developing a modeling tool using eclipse, in: International Workshop on Advanced Software Development Tools and Techniques, Co-located with ECOOP 2008, University of Groningen, 2008.

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

[26] G. Pothier, E. Tanter, Extending omniscient debugging to support aspect-oriented programming, in: In Proceedings of SAC, ACM, 2008.

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

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

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

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

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

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

[33] Z. Hemel, E. Visser, Pil: A platform independent language for retargetable dsls, in: M. van den Brand, D. Gasevic, J. Gray (Eds.), Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers, volume 5969 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 224–243.