Debug Interface for Multi-Aspect Debugging

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

Yoav Apter

The Research Thesis Was Done Under
the Supervision of Prof. David H. Lorenz
in the Dept. of Mathematics and Computer Science
The Open University of Israel

Submitted to the Senate of the Open University of Israel
Heshvan 5773, Raananna, October 2012

Acknowledgements

I thank my adviser, Prof. David Lorenz, for his professional assistance along the way. I also appreciate his constant care for various administrative issues. I would also like to thank Dr. Oren Mishali for his invaluable help to my research.

The generous support of the Open University Research Authority is acknowledged. This research was support in part of NSF under Grant No. 926/08.

Abstract

The development of software systems using multiple aspect languages is an approach that involves programming in general purpose languages, such as AspectJ, along side with domain specific languages, like COOL. A major research effort in this area is to control and coordinate the interoperability of the corresponding aspect mechanisms. Unfortunately, less attention has been given to making this mode of development practical. Providing an environment with dedicated tools that make the actual development with multiple aspect languages effective is just as essential for this approach to succeed.

This thesis focuses on one such tool: a debugger for applications written using multiple aspect languages. The debugger allows the developer to investigate the runtime state and behavior of applications of this kind. In addition, the debugger lets the developer query the composition specification, to find out how the different aspect mechanisms are composed and interact. This is essential since incorrect composition may be the reason for certain bugs. We analyze the problem of debugging such applications using several examples. We present a specification for a corresponding debug infrastructure. We provide a concrete implementation over an existing aspect composition framework.

List of Publications

[2] Y. Apter, D. H. Lorenz, and O. Mishali. Toward debugging programs written in multiple domain specific aspect languages. In *Proceedings of the 6th AOSD Workshop on Domain-Specific Aspects Languages (DSAL'11)*, Porto de Galinhas, Brazil, 2011. ACM.

[3] Y. Apter, D. H. Lorenz, O. Mishali. A debug interface for debugging multiple domain specific aspect languages. In *Proceedings of the 11th Annual international Conference on Aspect-Oriented Software Development* , Potsdam, Germany, March 25 - 30, 2012. AOSD '12. ACM

Contents

List of Figures

Listings

Chapter 1

Introduction

An aspect-oriented extension to a programming language is termed *domain specific* when some (general purpose) expressiveness is surrendered for more (domain specific) conciseness in describing a specific crosscutting concern in the terminology of the domain. For example, COOL [14, 15] is a *domain specific aspect language* (DSAL) with high level constructs just for specifying declaratively the synchronization of threads in the program. To regain the lost expressiveness, DSALs are used with other DSALs and collectively with general purpose aspect languages.

The development of aspect-oriented software systems using multiple DSALs has gained attention in recent years (e.g., the DSAL workshop series in AOSD). This mode of development is denoted here as *multi*-DSAL *development* [2]. A major research effort in this area is to coordinate the collaborative operation of the *aspect mechanisms* [11] implementing the various DSALs. The research has led to the creation of *aspect composition frameworks*, e.g., Pluggable AOP [10], Reflex [22], AWESOME [12], JAMI [8], Crosscutting Composition [6], and others. Less research attention has been devoted to making the mode of multi-DSAL development practical. For DSALs to be used in practice, we need tools that make the actual development with multiple DSALs effective. This thesis focuses on one such tool—the *debugger*.

1.1 Runtime State and Behavior

Modern debuggers provide by default standard facilities for examining the state and behavior of an application at runtime. However, when the object of debugging is a multi-DSAL application, additional unique investigation facilities are required from the debugger. For instance, when a breakpoint is placed on a *method-execution* join point, the developer should be able to examine the list of applied advice of *all* the aspect languages. Moreover, the developer needs to be able to distinguish between advice of different languages.

Today, debuggers are unaware of these requirements and do not provide such facilities. One of the few that does provide them is the *aspect-oriented debugging architecture* (AODA) [4]. The AODA debugger lets the developer investigate the application in terms of its AOP abstractions, namely, aspects, pointcuts, and advice. However, AODA, as the other AOP debuggers, is designed for a single aspect mechanism. When AODA is used to debug multi-DSAL applications, only ASPECTJ advice is listed during the debug process. Of course, the source code of the other aspect languages could be translated to ASPECTJ. However, even if such a translation occurs, the developer might be able to observe the whole list of advice, but they would all seem to belong to ASPECTJ. Moreover, source-to-source translation in general may introduce and expose synthetic join points that do not exist in the original source code, resulting in incorrect application behavior [13].Hence, even AODA is not suited for multi-DSAL debugging.

1.2 Composition Specification

A multi-DSAL debugger should also provide the developer with the ability to reason about the *composition specification* [12, 16]. Composing multiple aspect mechanisms into a coherent weaver is a complex process that is facilitated by an aspect compo-

sition framework. In a typical composition process, the framework is provided with multiple aspect mechanisms, one for each aspect language. A *composition designer* then uses the framework to configure the interactions among the mechanisms, based on a particular composition specification. The specification may set, for instance, a specific order on advice of different aspect mechanisms that operate at the same join point. The composition framework produces as an output a single multi-DSAL weaver that behaves according to the specification.

It may be difficult for the composition designer to formulate in advance a complete composition specification that achieves the desired application behavior flawlessly. It may also be difficult to communicate the specification to the developer. Therefore, in addition to normal bugs whose cause lies in the application layer due to improper use of language constructs or incorrect application logic, composition bugs may exist due to an incorrect or misunderstood composition specification. Thus, the debugger should let the developer inspect the composition specification. Again, none of the debuggers currently supports this feature.

1.3 Contribution

This work presents a *Multi*-DSAL *Debug Interface* (MDDI) [3]. The interface comprises two sets of debug operations. The first set of operations can be used to examine the runtime state and behavior of a multi-DSAL application. The operations in this set enhance the debug features found in AODA [4]. The second set of operations can be used to inspect the application from the composition specification point of view. The operations defined in this set are based on prior work that characterized abstract features of an aspect mechanism, and in particular, features of the composition of several mechanisms [11, 13]. These abstract features are implemented in terms of concrete debug operations in MDDI.

We provide an implementation of MDDI over a specific aspect composition framework called AWESOME [12]. We modified the AWESOME framework to produce a multi-DSAL weaver that is "debug aware." The weaver inserts into the woven files special debug attributes, which a debugger can access via MDDI to provide runtime state and behavior information and details about the composition specification.

Our scope of investigation for multi-DSAL development targets a dominant family of *reactive aspect mechanisms* [11]. This family includes the set of AOP languages known as *join point and advice*.

Outline. Chapter 3 presents an introductory example to concretely illustrate the problem and the solution. This example demonstrates use of MDDI to identify and locate the cause for a bug in a multi-DSAL application. Chapter 4 presents the debug operations in MDDI. In Chapter 5.1, a concrete implementation of the MDDI over the AWESOME framework is described. In Chapter 6, the debug interface is evaluated.

Chapter 2

Background

We first discuss works related to debugging of AOP programs. We then discuss the composition of multiple aspect mechanisms.

AOP debugging The debugging of AOP programs has been considered before. One approach is omniscient debugging [21]. Under this kind of debugger, a bytecode level trace is generated for the program execution. The trace includes synthetic code, woven advice, and other technology specific entities. Annotations on the trace indicate the origin of the bytecode, whether it is base code, residue or advice applications. While this kind of debugging can handle the execution of multiple aspects, it does not offer source level debugging, nor does it aid in solving problems that result from feature interactions between aspects or from the composition specification.

Another debugging approach used by Wicca [7] is providing a source-level representation of the woven source code. The representation is generated by a dynamic source weaver. However, even for the case of a single AOP language, this approach is limited, because the code presented is not the original source code written by the developer but the one generated by the source weaver. It also does not provide debugging in terms of source level abstractions, such as aspects and advice. In the case of a multi-DSAL program, a translation of the different source files to a common base

language is required, which brings back the problems related to the composition of multiple aspect mechanisms [10].

Unlike Wicca, AODA [4] enables debugging that is aware of AOP source level abstractions, such as aspects and advice. It features a modular design that allows designing new debug agents to support different AOP languages. However, it does not address the unique problems presented by multi-DSAL programs. First, it does not provide a way for examining the composition specification. Second, it does not recognize AOP artifacts in the context of the original mechanisms, but only in a common base language abstraction (for example, when COOL coordinators are represented as ASPECTJ aspects). In addition, developing a new debug agent for a new mechanism is a difficult task, requiring implementation knowledge of AODA itself and how to add debug attributes to class files.

Yin et al.[23] present another AOP aware debugger. It is based on their own implementation of an ASPECTJ compiler and it uses external XML files to store the debugging information. Their approach allows them to define a finer grained model that enables supporting additional debugging tasks not supported by previous work. However they also do not address the multi-DSAL debugging problem.

In comparison, our debugging infrastructure supports debugging multi-DSAL programs with source level abstractions in the context of each mechanism, including the inspection of the composition specification. We also provide a generic way to support debugging of new mechanisms without having to dive into the details of the debug attributes or class file structure.

Composition Frameworks Pluggable AOP [10] introduced the problem of composing multiple aspect language extensions. The work presents a framework for third-party composition of arbitrary dynamic aspect mechanisms into an AOP interpreter.

AWESOME [12] is a composition framework that directly weaves DSAL code without an intermediate translation to a common base language. AWESOME provides a default composition specification. For example, by default aspects will advise a foreign aspect by advising only JAVA statements within its source code. AWESOME allows the composition designer to specify how to resolve feature interactions between aspect mechanisms in case the default specification is not the desired one. While AWESOME addresses the composition problem, it does not deal with the problem of debugging the woven multi-DSAL programs. The developer is required to use regular debugging tools, such as the JAVA debugger (`jdb`). However, such tools expose the synthetic constructs of AWESOME and of the mechanisms, and they also do not provide debugging in terms of AOP source level abstractions.

A different approach for multi-DSAL composition is presented by Dinkelaker et al. [6]. They propose an architecture for embedded DSLs (EDSLs) that makes use of meta-object protocols and aspect-oriented concepts to support crosscutting composition of EDSLs. This enables writing modularized EDSL programs where each program addresses one concern. Their proposed architecture is implemented in Groovy, and like AWESOME, the architecture does not address the debugging problem at all, relying instead on the standard Groovy debugging tools.

Chapter 3

Motivating Example

To illustrate the issues involved in debugging a multi-DSAL application and to outline our solution, we first present a simple example. The application in our example comprises a base system, written in JAVA, with three concerns, each expressed in a different aspect language. ASPECTJ contributes tracing; COOL handles thread synchronization; and VALIDATE—a DSAL that we have defined ourselves—enforces validation of input parameters.

Recall that the source of bugs or surprising behavior in a multi-DSAL application is either erroneous implementation somewhere in the source code, or incorrect or misunderstood composition of the aspect mechanisms. In this chapter we provide an example for a bug of the latter sort. In the example, an unexpected behavior of the application is observed. We describe how the developer might utilize a multi-DSAL debug interface to: (1) identify a bug caused by an incorrect composition, and (2) understand the essence of the bug.

We provide as we go brief explanations that are necessary for understanding the part of the interface that is being discussed. A more complete specification of the debug interface is presented and explained in chapters 4 and 5.1.

Listing 3.1: A stack implementation in JAVA

```
1  public class Stack {
2          public Stack(int capacity) {
3                  buf = new Object[capacity];
4          }
5          public void push(Object obj){
6                  buf[ind] = obj;
7                  ind++;
8          }
9          public Object pop() {
10                 Object top = buf[ind-1];
11                 buf[--ind] = null;
12                 return top;
13         }
14         private Object[] buf;
15         private int ind = 0;
16 }
```

3.1 A Multi-DSAL Application

Our running example is a multi-DSAL application, which is also multi-threaded. The base system is a JAVA class that implements a bounded stack. The class `Stack` (Listing 3.1) defines two public methods, `push` and `pop`, where an `ArrayIndexOutOfBoundsException` is thrown upon an attempt to pop objects off an empty stack or push objects onto a full stack.

In addition, three aspects are defined, each expressed in a different aspect language. Note that the term *aspect* is used here and throughout the thesis in a broad meaning, and refers to the language construct introduced by *any* aspect language to encapsulate a crosscutting concern. In ASPECTJ this construct is called *aspect*. COOL calls it *coordinator*, and in VALIDATE the aspect construct is denoted *validator*.

The first aspect is defined in ASPECTJ, a general purpose aspect language. The aspect enhances `Stack` with tracing facilities (Listing 3.2).

The second *coordinator* aspect (Listing 3.4) is written in COOL, an *off-the-shelf* DSAL that facilitates synchronization of JAVA methods. The coordinator enforces the

Listing 3.2: A tracing aspect in AspectJ

```
1  public aspect Tracer {
2          pointcut scope(): !cflow(within(Tracer));
3          before(): scope() {
4                  out.println("before " + thisJoinPoint);
5          }
6          Object around(): scope() {
7                  out.println("around " + thisJoinPoint);
8                  return proceed();
9          }
10         after(): scope() {
11                 out.println("after " + thisJoinPoint);
12         }
13 }
```

Listing 3.3: A validator in Validate

```
1  validator Stack {
2          validate Stack(int capacity):
3                  $1 > 0;
4          validate push(Object obj):
5                  string($1), email($1);
6  }
```

Listing 3.4: A synchronization coordinator in COOL

```
1  coordinator Stack {
2          selfex {push, pop};
3          mutex {push, pop};
4          int len=0;
5          condition full=false, empty=true;
6          push: requires !full;
7          on_exit {
8                  empty=false;
9                  len++;
10                 if (len==buf.length)
11                         full=true;
12         }
13         pop: requires !empty;
14         on_entry { len--; }
15         on_exit {
16                 full=false;
17                 if (len==0)
18                         empty=true;
19         }
20 }
```

following synchronization policy for each instance of `Stack`:

- neither `push` nor `pop` may be executed by more than one thread at a time (**`selfex`** declaration);
- `push` and `pop` are prohibited from being executed concurrently (**`mutex`** declaration);
- `push` may be called only if the stack is not full (condition `full`); and
- `pop` may be called only if the stack is not empty (condition `empty`).

The **`on_entry`** and **`on_exit`** clauses express the bookkeeping required to implement the last two items.

The third aspect language is called VALIDATE, a simple *in-house* DSAL that the developer defines. VALIDATE supports validation of input arguments passed to methods, constructors, and fields (field assignments). As a motivation for using such a

language, consider a development team that is interested in involving domain experts in the implementation of the security concern of the system (one facet of security is input validation). Furthermore, assume that the domain experts are familiar with the Unix shell. To accommodate the experts, VALIDATE takes on a syntax that resembles shell commands. The input validation of a particular program element, e.g., a method, is contained in a validation *command*. Within a command, `$`(i) is used to access the i'th input argument. In addition, a library of predicates exists for defining the validation criteria. Validation commands for one or more classes are grouped in a *validator* aspect.

In Listing 3.3, a validator for the `Stack` class is presented. It validates the constructor and the `push` method. The validator specifies that the constructor's first argument (the capacity of the stack) should be a positive integer. It also specifies that the element that is added to the stack via the `push` method should be a **`String`** object conforming to the format of an email address (**`string`** and **`email`** are predicates of the language).

3.2 A Multi-DSAL Debug Scenario

Consider the following scenario. During the development of our multi-DSAL application, it is tested and executed against different input sets. On one of the input sets, an unexpected behavior is observed: an exception is thrown from the `push(Object)` method indicating a validation error. This indicates that the VALIDATE aspect mechanism identified an invalid input argument. However, unexpectedly, the execution (of other threads) does not progress and it seems like the program is stuck. The developer initiates a debug session, ready to investigate the cause of the problem.

The exception was thrown from the `push` method, thus it becomes the natural suspect and a breakpoint is placed on the method entry. When the breakpoint is

reached, the program suspends and waits for additional debugging commands. The developer first asks the multi-DSAL debugger for all the pieces of advice that were applied to this *method-execution* join point. The developer expects the advice of all the aspect mechanisms to be present, but discovers instead that the COOL advice seems to not have been applied. This is surprising, since the program explicitly specifies that `push` should be synchronized (Listing 3.4, lines 2-3).

Understanding the Composition Generally, when debugging a multi-DSAL application, the basic guideline is to understand the feature interactions that are relevant to the portion of code under investigation. The developer should always look for the cause of the problem in the application code, but should also be open to the possibility that the unexpected behavior is a matter of a composition specification. In our case, it should be examined whether or not the unexpected absence of COOL advice originated from the composition. To find this out, the developer uses dedicated debug operations for inspecting the composition.

Join point granularity and join point visibility are two of several features that characterize a reactive aspect mechanism [13]. The join point granularity feature specifies what kinds of join point computations may be intercepted by the aspect mechanism. The granularity of ASPECTJ includes, for example, computations of kinds *method-call*, *method-execution*, *field-set*, etc. The join point visibility feature maps join point computations to actual join point instances. That is, each potential join point in the granularity is classified as either *visible* or *invisible*. For example, the visibility feature of ASPECTJ hides all the join points within the lexical scope of an `if` pointcut expression.

Based on these abstract features, MDDI defines concrete debug operations: *granularity* and *visibility*. These operations operate on code elements in a particular program. In the ASPECTJ language execution model, each dynamic join point has

a corresponding static shadow in the bytecode of the program. Advice code may be inserted at these shadows to modify the behavior of the program [9]. Provided with a code element, e.g., a method, the granularity operation returns all the join point shadows in that method that a particular mechanism may *plausibly* advise. The visibility operation returns the join point shadows in the method that the mechanism may *actually* advise.

In the example, an exception was thrown from the `push` method and then the program deadlocked. Following we will illustrate the debugging process concretely using the AwesomeDebugger.

We use the debugger to observe that Cool related advice are not applied at the `push` method. Next, we investigate whether the absence of the advice is a matter of the composition specification or not. We inspect the shadows in `push` that are visible to Cool:

```
(awdb) show visibilty COOL
[ID]    [Joinpoint type]    [Source location]
-------------------------------------------------
```

An empty shadow set is displayed. This indicates that there are no shadows in the method that are visible to the Cool mechanism. Therefore we proceed to inspect the shadows in `push` that are visible to all the mechanisms:

```
(awdb) show visibility
[ID]    [Joinpoint type]    [Source location]
-------------------------------------------------
0       Field Get       Stack.java:6
1       Field Set       Stack.java:6
2       Field Get       Stack.java:7
3       Field Set       Stack.java:7
```

```
4     Method Execution     Stack.java:5
```

We see that even though that no shadows are visible to the COOL mechanism, they are visible to other mechanism, We then use the granularity operation to understand *why* the *method-execution* shadow is not visible to COOL:

```
(awdb) show granularity COOL
[ID]     [Joinpoint type]     [Source location]
_____________________________________________
```

An empty set is displayed again. This means that the problem is not just with the visibility of the shadows, but also in the granularity of the COOL mechanism. We conclude that the reason that COOL advice are missing is because the COOL mechanism was not designed to affect *method-execution* join point shadows.

Resolving the Bug Armed with this knowledge, the developer can consult the composition designer to confirm that indeed the granularity of COOL includes join points of kind *method-call* (and not *method-execution*). The COOL mechanism defines two types of advice, **`lock`** and **`unlock`**, which are executed before and after the synchronized method, respectively. Thus, COOL's **`lock`** and **`unlock`** advice are inserted in the context of the *caller* method and not in the context of the *callee*.

The developer concludes that this particular organization of the advice is the cause for the bug. When a thread T_i calls `push`, the **`lock`** advice is executed first in the context of the caller. When T_i is allowed to execute `push` (hence acquiring the lock), the **`validate`** advice is executed in the context of the callee. If the argument to `push` is found to be invalid, an exception is thrown (like in our case). This exception causes the termination of T_i, but without releasing the lock. From there on, any other thread T_j which attempts to call `push` is blocked. Hence the program enters a deadlock.

The analysis implies that for the program to function properly, the **`validate`**

advice should execute *before* the **`lock`** advice. Indeed one of the feature interactions that a composition designer should solve is so-called *emergent advice ordering* [13]. It is where the designer specifies an order between advice of different mechanisms. However, in our case the desired order cannot be set. An advice order can only be specified when the advice operate on the same join point. Here, **`lock`** advice will always execute before **`validate`** because a caller's *before* advice precedes any callee's advice. Therefore, it should be first specified that both **`lock`** and **`validate`** operate on the same join point (be it a *method-call* or a *method-execution*). This is done by modifying the granularity of Cool or that of Validate. Then, the desired advice order should be set, i.e., that **`validate`** should occur before **`lock`**.

Note that the analysis also suggests that the Cool mechanism should be refined to release an acquired lock upon a thrown exception. Although such a refinement may solve the bug, the proposed solution is still more desirable, because a solution at the composition level is more robust and does not depend on a specific implementation approach.

Chapter 4

Multi-DSAL Debug Operations

The multi-DSAL debug interface (MDDI) defines debug operations for inspecting a multi-DSAL application at runtime. In this chapter, the debug operations in MDDI are described in abstract platform-independent terms. The description is organized in two parts: debug operations for examining the runtime state and behavior of a multi-DSAL application, and debug operations for inspecting the composition specification. In Chapter 5.1, the concrete implementation of MDDI for the AWESOME composition framework is presented.

4.1 Examining Runtime State and Behavior

Like a typical debugger, the debugging process of the multi-DSAL debugger is based on stopping the program at a certain breakpoint and then examining the runtime state and behavior using dedicated debug operations.

AODA [4] defines an aspect-oriented breakpoint model with debug operations that support stopping at a join point shadow in three modes: before, after, and during the execution of the advice woven at that shadow. When a breakpoint is reached, AODA offers three debug operations for inspecting the list of advice:

1. **Inspection of woven advice.** This operation lists advice that are woven at a join point shadow. Note that a woven (applied) advice does not necessarily get executed, e.g., an advice associated with an `if` pointcut in AspectJ.

2. **Inspection of executing advice.** This operation lists advice that are currently executing on the stack. Note that several advice may be executing simultaneously, e.g., when an `around` advice calls `proceed` and, while it waits for the call to return, another `before` advice takes control.

3. **Inspection of past advice.** This operation lists advice that were already processed, indicating whether each advice on the list was actually executed or not.

The multi-DSAL debugger adjusts AODA breakpoint model for debugging multi-DSAL applications. In comparison to AODA, the multi-DSAL debugger displays the list of advice woven by *all* the aspect mechanisms. The multi-DSAL debugger indicates for each presented advice its originating aspect mechanism and the *type* of the advice. Each aspect mechanism defines its own advice types. AspectJ has three advice types, namely `before`, `after`, and `around`. Cool declares two advice types, `lock` and `unlock`, which are executed before and after the invocation of a synchronized method, respectively. The Validate mechanism has a single advice type called `validate`, which is executed before the validated code element. Indicating the advice type and the originating mechanism may help the developer to get a clearer picture of the interactions involved.

Like AODA, the multi-DSAL debugger produces *mirror objects* for the advice declared in the program. These are objects created during the debug process to reflect the state of corresponding objects in the debugged application [18, 5]. Each advice mirror is linked to the related source code. In AspectJ, the mapping is straightforward. Each advice mirror is simply mapped to the related advice construct in the

source code. However, when DSALs are involved, this mapping is often implicit. For instance, the COOL advice types, **`lock`** and **`unlock`**, are concepts that are a part of the implementation model of COOL, but without an explicit representation in the source code. The COOL language designer should decide to which source code abstractions each advice type is mapped. For instance, a reasonable mapping is to associate a **`lock`** advice with operations defined in **`mutex`**, **`selfex`**, **`requires`**, and **`on_entry`** expressions (Listing 3.4). Therefore, the multi-DSAL debugger provides support for implicit source code mappings of this kind.

4.2 Inspecting the Composition Specification

The other part of MDDI includes operations for investigating the composition specification of a multi-DSAL weaver. These include *granularity* and *visibility* operations for investigating *which* join point shadows within a particular program element an aspect mechanism may affect. A third kind is *advisability* operations for determining *how* the aspect mechanism may affect those shadows.

Granularity Operations The join point granularity feature of an aspect mechanism $\mathcal{M}$, denoted $granularity(\mathcal{M})$, specifies in abstract terms the kinds of join point computations that the mechanism may intercept. For instance, $granularity$(COOL) includes computations of kind *method-invocation*. This indicates that COOL may affect the program only when methods are invoked. When the mechanism is implemented in the context of a specific environment, the granularity is normalized according to a shared join point scheme (in our case ASPECTJ). A *method-invocation* join point computation in COOL may be mapped to either a *method-call* or a *method-execution* join point in ASPECTJ. Both mapping options are reasonable normalization choices, yet, as illustrated in Chapter 3, the decision may change the collective behavior. Therefore, the mapping may be subjected to adjustments by the composition

designer.

The granularity operation in MDDI is defined in relation to a particular code element in the program, such as a method, a class, or an aspect. The debug operation is of the form:

$$GRANULARITY_{\mathcal{M}} : Elements \rightarrow P(Shadows)$$

Given a code element $\mathcal{C}$ and an aspect mechanism $\mathcal{M}$, the operation returns the join point shadows in $\mathcal{C}$ that are in the granularity of $\mathcal{M}$:

$$GRANULARITY_{\mathcal{M}}(\mathcal{C}) = \left\{\mathcal{S} \in \mathcal{C} |\, shadow(\mathcal{S}) \wedge \mathcal{S}.kind \in granularity(\mathcal{M})\right\}$$

Note that the operation returns join point shadows, since it relates to code elements and not to the dynamic execution of the program.

As a usage example for this granularity operation, consider the method `push` defined in the `Stack` class (Listing 3.1, lines 5–8). The method includes five join point shadows: a *field-get* and a *field-set* in line 6, another *field-get* and *field-set* in line 7, and a *method-execution* shadow. The granularity operations of the different mechanisms evaluate to:

$$\begin{aligned} GRANULARITY_{\text{AspectJ}}(\texttt{push}) = \big\{&\textit{field-get}(\texttt{ind}),\\ &\textit{field-set}(\texttt{buff}),\\ &\textit{field-get}(\texttt{ind}),\\ &\textit{field-set}(\texttt{ind}),\\ &\textit{method-execution}(\texttt{push})\big\} \end{aligned}$$

$$GRANULARITY_{\text{Cool}}(\texttt{push}) = \big\{\textit{method-execution}(\texttt{push})\big\}$$

$$\begin{aligned} GRANULARITY_{\text{Validate}}(\texttt{push}) = \big\{&\textit{field-set}(\texttt{buff}),\\ &\textit{field-set}(\texttt{ind}),\\ &\textit{method-execution}(\texttt{push})\big\} \end{aligned}$$

Note that the result returned by the Cool granularity operation reflects a change that was made to the composition in order to resolve the bug detected in Chapter 3. Cool and Validate were reconfigured to operate on the same *method-execution* join

points. The granularity of VALIDATE includes join point computations in which input validation makes sense, namely, computations of kinds *method-invocation* (mapped to *method-call* or to *method-execution*), *object-creation* (*constructor-execution*), and *field-assignment* (*field-set*).

Another unified operation, $GRANULARITY(\mathcal{C})$, returns the join point shadows in code element $\mathcal{C}$ that any of the mechanisms may affect. It is the union of all the mechanism-specific granularity features. In the case of `push`,

$$GRANULARITY(\texttt{push}) = GRANULARITY_{\text{ASPECTJ}}(\texttt{push})$$

However, should ASPECTJ be excluded from the composition, the resulted set would be different: the *field-get* shadows would not be included, since shadows of this kind are neither part of the granularity of COOL nor of VALIDATE.

Visibility Operations The join point visibility feature of an aspect mechanism classifies join points in the granularity as either *visible* or *invisible.* Invisible join points are not available for advising. The visibility operation in MDDI has the form:

$$VISIBILITY_{\mathcal{M}} : Elements \rightarrow P(Shadows),$$

where $VISIBILITY_{\mathcal{M}}(\mathcal{C})$ denotes the set of join point shadows in $\mathcal{C}$ that are visible to $\mathcal{M}$. A unified visibility operation, $VISIBILITY(\mathcal{C})$, returns the set of shadows in $\mathcal{C}$ that are visible to any of the aspect mechanisms. For a mechanism $\mathcal{M}$ and a code element $\mathcal{C}$, the following proposition holds:

$$VISIBILITY_{\mathcal{M}}(\mathcal{C}) \subseteq GRANULARITY_{\mathcal{M}}(\mathcal{C})$$

To illustrate the difference between granularity and visibility, consider a `ConditionalTracer` aspect in ASPECTJ (Listing 4.1). In ASPECTJ by design join point shadows within an `if` pointcut (line 4) are invisible. Therefore, whereas applying the ASPECTJ granularity operation on the aspect `ConditionalTracer` results in:

Listing 4.1: A conditional tracer in AspectJ

```
1 public aspect ConditionalTracer {
2   public static boolean trace = false;
3   before() : execution(* BoundedStack.pop())
4    && if(trace) {
5      System.out.println(thisJoinPoint);
6   }
7 }
```

$$GRANULARITY_{\text{AspectJ}}(\texttt{ConditionalTracer}) = \{\textit{field-set}(\texttt{trace}), \textit{advice-execution}(\texttt{before}), \textit{field-get}(\texttt{trace}), \textit{method-call}(\texttt{println}), \ldots\},$$

the visibility operation returns:

$$VISIBILITY_{\text{AspectJ}}(\texttt{ConditionalTracer}) = \{\textit{field-set}(\texttt{trace}), \textit{advice-execution}(\texttt{before}), \textit{method-call}(\texttt{println}), \ldots\}.$$

Note that the *field-get*(`trace`) shadow, which corresponds to the read operation of the Boolean **`trace`** field in line 4, is in the granularity but not in the visibility.

Advisability Operations The join point advisability feature defines advising constraints for various types of join points, or even for specific join points [13]. Recall that each mechanism declares one or more advice types. For instance, AspectJ defines three types of advice: **`before`**, **`after`**, and **`around`**. By default, an aspect mechanism may apply any type of advice on a visible join point. However, in some cases the composition designer may be interested in restricting the potential effect of a particular aspect mechanism, that is, preventing advice of a certain type from being applied at certain join points. For instance, it may be defined that AspectJ cannot declare an **`around`** advice at executions of Cool's **`lock`** and **`unlock`** advice. This may be essential in order to prevent AspectJ aspects from overriding Cool's

synchronization logic.

$ADVISABILITY_{\mathcal{M}}$ lets the developer investigate how an aspect mechanism may affect a particular join point shadow in the program. The operation returns the advice types that the mechanism may apply at the shadow:

$$ADVISABILITY_{\mathcal{M}} : Shadows \rightarrow P(AdviceTypes)$$

Given an aspect mechanism $\mathcal{M}$ and a shadow $\mathcal{S}$, the advisability operation is defined by:

$$ADVISABILITY_{\mathcal{M}}(\mathcal{S}) = \Big\{T \in \mathcal{M} |\, adviceType(T) \ \wedge \ applicable(T, \mathcal{S})\Big\}$$

Continuing the example, if $\mathcal{S}$ is *advice-execution*(`lock`) shadow in Cool, then its advisability in relation to AspectJ would be:

$$ADVISABILITY_{\text{AspectJ}}(\textit{advice-execution}(\texttt{lock})) = \Big\{\text{before}, \text{after}\Big\}$$

This means that AspectJ may only apply **`before`** or **`after`** advice at a **`lock`** execution. Another unified advisability operation, $ADVISABILITY(\mathcal{S})$, returns the advice types that may be added by any aspect mechanism. In a configuration that includes AspectJ, Cool, and Validate,

$$ADVISABILITY(\textit{advice-execution}(\texttt{lock})) = \\ = ADVISABILITY_{\text{AspectJ}}(\textit{advice-execution}(\texttt{lock}))$$

The same result is returned because *advice-execution* join points are not in the granularity of Cool nor of Validate, hence by specification they cannot affect the join point.

Chapter 5

Implementation

5.1 MDDI Implementation

MDDI is implemented as a multi-DSAL extension to AJDI, a debug interace for aspect-oriented applications introduced in AODA. AJDI itself is an extension to the Java Debug Interface (JDI). The implementation of MDDI requires debug information, which should be attached to the target multi-DSAL application. A common technique which we use is to add debug attributes to the class files of the application. We formulate the debug attributes that are needed for implementing the MDDI operations. For the operations that query the runtime state, we extend existing AODA debug attributes. For the composition specification operations, we define new debug attributes.

The debug attributes should be added to the class files of the application during the weaving process. In a multi-DSAL setup, the weaving process is controlled by an aspect composition framework. Since none of the existing frameworks handles debugging as part of its weaving process, we extend the AWESOME composition framework for that purpose.

The overall multi-DSAL debug process over AWESOME is illustrated in Figure 5.1. AWESOME is provided with multiple aspect mechanisms and with a composition spec-

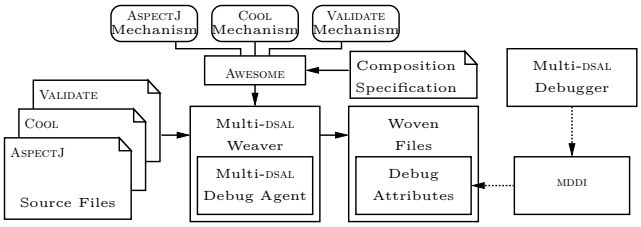

Figure 5.1: The multi-dsal debugging process

ification, and outputs a single multi-dsal weaver. Awesome is customized such that the produced weaver is embedded with a multi-dsal debug agent. The weaver is provided with an application written in multiple aspect languages, and during the weaving process the agent adds the dedicated debug attributes to the woven class files of the application. These attributes are then consumed by mddi, which is utilized by the multi-dsal debugger.

The embedded multi-dsal debug agent is significantly different than that of aoda. In aoda, in order to add debug support for a particular aspect language, one must implement an appropriate debug agent. This is a difficult task since it requires to understand the structure of the debug attributes, as well as knowing how to embed them in the resulting class files. In contrast, the multi-dsal debug agent that is added to Awesome relieves the designer of the aspect mechanism from this tedious implementation task. Instead, the process of adding the debug attributes is handled by the agent. The information needed for producing the attributes that enable the composition operations is extracted by the agent from the composition specification. The information needed for the attributes of the state and behavior operations is fetched from dedicated interfaces that should be implemented by the developer of each mechanism.

In the rest of this chapter, we present the implementation of the debug operations for inspecting the composition specification, and for investigating the runtime state and behavior. For each kind of operation, we describe the corresponding mddi elements, and the changes made to the Awesome weaving process and to AJDI.

5.1.1 Composition Specification Operations

MDDI extends AJDI with several methods and types for inspecting the composition specification. The methods, listed here, realize the debug operations that were described in Chapter 4.

```
1 JoinPointComputation[] *.granularity();
2 JoinPointComputation[] *.granularity(Mechanism);
3 JoinPointComputation[] *.visibility();
4 JoinPointComputation[] *.visibility(Mechanism);
5 AdviceType[] JoinPointComputation.advisability();
6 AdviceType[] JoinPointComputation.advisability(Mechanism);
```

The methods let the debugger inspect the granularity, visibility, and advisability of the composition in relation to specific program elements. Each of the granularity or visibility methods (lines 1-4) is added to the AJDI elements `ClassType`, `Aspect`, and `Method`. The first granularity method (line 1) returns the granularity with respect to all aspect mechanisms. The second (line 2) returns the granularity in relation to a specific mechanism. The same applies to the visibility methods (lines 3-4). The advisability methods operate on a join point computation (shadow). The first method (line 5) returns all the advice types that may be applied at the shadow, of all aspect mechanisms. The second method (line 6) returns the advice types of a specific mechanism.

The implementation of the methods is facilitated by three debug attributes that are added to the woven class files of the multi-DSAL application during the weaving process: `GranularityAttribute`, `VisibilityAttribute`,
and `AdvisabilityAttribute`.

GranularityAttribute During the extended AWESOME weaving process, the granularity of each method, class, or aspect, is calculated with respect to each aspect

mechanism in the composition. The result is saved in a debug attribute (`GranularityAttribute`) that is attached to each corresponding code element in the woven class files. The following pseudo code shows the calculation of the granularity attribute for a method or a class element $\mathcal{C}$ in the base system:

```
GranularityAttribute att;
Shadow[] shadows = reify(C);
foreach s in shadows
        att.append(s);
        foreach mech in mechanisms
                if( granularity(mech) includes s.kind )
                        att.append(mech);
        endforeach
        att.append(newline);
endforeach
```

The reify method in line 2 returns all the join point shadows in $\mathcal{C}$. In line 4, the signature of each shadow is appended to the debug attribute (a debug attribute is simply a string that is later added to a class file). Afterward, we check for each of the mechanisms in the composition whether its granularity includes the kind of the current shadow (lines 5-6). If so, the name of the mechanism is written to the attribute (line 7). By that, we are able to tell, for each of the shadows in $\mathcal{C}$, to which of the granularities of the different aspect mechanisms it belongs. We may also infer the shadows that do not belong to any granularity.

This calculation works for methods and classes in the base system. The calculation of the granularity attribute for aspects is slightly different. For an aspect $\mathcal{A}$, the second line in the calculation is replaced by the line:

```
Shadow[] shadows = exposed_shadows(A);
```

An aspect language, in particular a domain-specific one, may operate in a higher level of abstraction. As a result, some of the shadows of its aspects may be considered *internal*. It is the responsibility of the composition designer to decide which shadows in the aspects of each mechanism should be exposed for the use of others. For instance, a COOL coordinator reads and writes to local conditional and ordinary variables, and to fields of the coordinated object. Therefore, it is reasonable to expose in a coordinator the corresponding *field-get* and *field-set* shadows. However, the designer may decide not to expose the *constructor-execution* and *initialization* computations because they do not reflect COOL's visible operation process. Hence, the controlled `exposed_shadows` method is used instead of `reify`. The method returns all the shadows in the aspect $\mathcal{A}$ that the designer decided to expose.

VisibilityAttribute The calculation of the visibility attribute is similar to that of the granularity attribute. For a given program element, all its shadows are first retrieved (either by `reify` or by `exposed_shadows`). Then, for each shadow and aspect mechanism, it is checked whether or not the shadow is in the *visibility* of the mechanism.

The check is made against the composition specification, which is available during the AWESOME weaving process. The designer of each aspect mechanism defines the granularity, i.e., the kinds of join points that the mechanism may potentially affect. The designer also provides, for each join point kind in the granularity, a predicate that tells in which circumstances join points of this kind are not visible.

AdvisabilityAttribute This attribute indicates, for each code element to which it is attached, the advice types that may affect the visible join point shadows in the element. Each line in the attribute describes the advisability of a particular visible shadow. It holds the shadow's signature, and a list of the advice types that may be applied to it.

Also here, the calculation of the attribute is based on consulting the composition specification. The composition designer defines in the composition specification *advisability restrictions*. Each restriction consists of a join point kind, and a list of the disallowed advice types. For example, consider the following restriction:

`advice-execution(validate)`	`aspectj.around`

This restriction specifies that an AspectJ **`around`** advice cannot be applied at executions of a **`validate`** advice. The restriction prevents the **`validate`** advice from being overridden.

5.1.2 Runtime State and Behavior Operations

In this chapter we describe the modifications made in order to implement the part of MDDI that deals with the inspection of runtime state and behavior.

Extending the AspectAttribute An extended AODA debug attribute called `AspectAttribute` is attached to each class file that represents an aspect of any mechanism. This debug attribute includes general information about the aspect (e.g., the defining mechanism), and about the different advice that it defines (e.g., their type and source code locations). The attribute is generated by the multi-DSAL debug agent. For that, the agent queries each mechanism for information about its aspects via an extended Awesome API, which is described next.

Extending the Awesome API In Awesome, the interface `IMechanism` represents an abstract aspect mechanism. `IMechanism` is implemented by each of the concrete aspect mechanisms in the composition. We extended `IMechanism` with several methods needed by the multi-DSAL debug agent to retrieve structural information about aspects in the application. Examples of methods that were added are:

```
String getName();
```

```
boolean handledByMe(Aspect azpect);
List<IEffect> getEffects(Aspect azpect);
```

The debug agent, when producing an **AspectAttribute** for a particular aspect, first calls the `handledByMe` method of each mechanism for determining to which of them the aspect belongs. Then, information is retrieved from the relevant mechanism, e.g., the name of the mechanism (by calling `getName`), and the list of the effects (advice) that the mechanism may apply to program code (via method `getEffects`).

The interface `IEffect` is also extended:

```
AdviceType getType();
ISourceLocation[] getSourceLocations();
```

The first method returns the type of the advice. For example, it returns **lock** or **unlock** for a COOL advice. The second method returns the locations in the source code to which the advice is mapped. The method returns an array type since in some cases an advice may be mapped to several different locations in the source code (recall Chapter 4.1).

The author of each aspect mechanism must implement these methods to enable the creation of the debug attributes. However, the implementation effort is reasonable, since the data that the methods need is already required for the weaving process. In our multi-DSAL composition example, a total addition of 25 lines-of-code were needed for the ASPECTJ mechanism to be debuggable; for the COOL mechanism 35 new lines-of-code were added, and making VALIDATE debuggable required 30 lines-of-code.

Chapter 6

Evaluation

To evaluate the MDDI implementation, we built a simple command-line, AwesomeDebugger, capable of debugging multi-DSAL programs. We demonstrate the debugging process on the `Stack` example (Chapter 3).

6.1 Debugging Foreign Advising

Foreign advising [13] refers to the case where an aspect of one mechanism advises join points in foreign aspects, i.e., aspects that belong to other mechanisms. The **`scope`** pointcut defined in the `Tracer` aspect (Listing 3.2) includes *advice-execution* join points contained in other foreign aspects, e.g., executions of Cool's **`lock`** and **`unlock`** advice. When applying an advice *around* join points in the scope, `Tracer` calls **`proceed`** (line 8). This is essential in order to resume the execution of the traced join points.

Consider a case where `Tracer` defines another advice that only monitors executions of Cool's **`unlock`** advice; and the call to **`proceed`** is mistakenly omitted. As a result, the **`unlock`** advice is not executed and thus the acquired lock is not released. The next thread that requests the lock is halted, and eventually the program may enter a deadlock.

When attaching the debugger to the deadlocked program we see the following stack trace in one of the halted threads:

```
(awdb) where
[0] Stack.pop Stack.java:13
[1] WriteReadThread.accessBuffer WriteReadThread.java:14
[2] BufferClientThread.run BufferClientThread.java:8
```

We further examine the advice that affect the execution of the **pop** method (in frame 0):

```
(awdb) show advice
[Aspect]    [Location]    [Type]    [Skipped]    [Mechanism]
-------------------------------------------------------------
cool.StackCoord (2, 3, 13, 14) Lock 0 COOL
aspectj.Tracer (3) BEFORE 0 AJ
aspectj.Tracer (6) AROUND 0 AJ
aspectj.Tracer (10) AFTER 0 AJ
cool.StackCoord (2, 3, 15) Unlock 1 COOL
```

The **Skipped** column indicates whether an advice was executed or skipped ('0' means executed, '1' means skipped). We can see (in the last line) that the COOL **unlock** advice was skipped. We can either fix our **around** advice to always proceed, or change the composition specification so that COOL operations cannot be advised by an **around** advice.

6.2 Debugging Co-Advising

Co-advising [13] refers to the case where advice belonging to different mechanisms are applied at the same join point. Often, a specific advice order should be set or the program may behave unexpectedly. For instance, if ASPECTJ advice are allowed to execute before COOL's **lock** advice or after COOL's **unlock** advice, then the ASPECTJ advice may unsafely access program resources.

To illustrate such a situation, suppose we add a new `top` method to class **Stack**:

```
Object top() {
    return buf[ind-1];
}
```

Calling `top` from such an AspectJ advice can yield a wrong result or an **`ArrayIndexOutOfBoundsException`**, since the access of both **`buf`** and **`ind`** is not synchronized.

When examining the advice executed at `top` we get:

```
(awdb) show advice
[Aspect]    [Location]    [Type]    [Skipped]    [Mechanism]
-------------------------------------------------------------
aspectj.Tracer (3) BEFORE 0 AJ
cool.StackCoord (2, 3, 13, 14) Lock 0 COOL
aspectj.Tracer (6) AROUND 0 AJ
cool.StackCoord (2, 3, 15) Unlock 0 COOL
aspectj.Tracer (10) AFTER 0 AJ
```

It may be inferred that the cause for the problem is an incorrect advice execution order that allows unsafe stack accesses. Hence we change the specification and set **`lock`** (**`unlock`**) to execute before (after) any AspectJ advice.

6.3 Debugging Advice Code

Another source for bugs is coding errors in the base program or in the aspects of the different DSALs. As an example, the coordinator in Listing 6.1 contains a simple bug: **`len`** is mistakenly decremented instead of being incremented (line 9). As a result, **`full`** is never set to **`true`** (line 10). The **`requires`** condition in line 6 is thus always met, allowing new elements to always be added to the stack. However, **`buff`** has a limited capacity and an **`ArrayIndexOutOfBoundsException`** will eventually be thrown.

We suspect that the problem lies in some advice code. We begin with checking the advice applied at the **`push`** method:

```
(awdb) show advice
[Aspect]    [Location]    [Type]    [Skipped]    [Mechanism]
-------------------------------------------------------------
validator.Stack (4) Validate 0 Validator
cool.StackCoord (2, 3, 6) Lock 0 COOL
```

Listing 6.1: Stack coordinator with a bug

```
1  coordinator Stack {
2    selfex {push, pop};
3    mutex {push, pop};
4    int len=0;
5    condition full=false, empty=true;
6    push: requires !full;
7    on_exit {
8      empty=false;
9      len--;
10     if (len==buf.length) full=true;
11   }
12   pop: requires !empty;
13   on_entry { len--; }
14   on_exit {
15     full=false;
16     if (len==0) empty=true;
17   }
18 }
```

```
aspectj.Tracer (3) BEFORE 0 AJ
aspectj.Tracer (6) AROUND 0 AJ
aspectj.Tracer (10) AFTER 0 AJ
cool.StackCoord (2, 3, 7) Unlock 0 COOL
```

The **`Location`** column links each advice to the corresponding source code (the numbers in parenthesis indicate the source lines relevant for each advice). The information helps in locating the specific advice code segments where the bug should be searched for. We check the code of each advice for errors, and eventually the bug is located in the **`on_exit`** declaration of the **`unlock`** advice.

Chapter 7

Conclusion

In order for DSALs to be used in practice, multi-DSAL development has to be cost-effective. Cost effectiveness is a requirement that applies not only to the implementation of DSALs, but just as much to the effective use of these DSALs [17]. While significant progress has been made on the language implementation front, less attention has been given to making the development of applications with multiple DSALs practical.

Effective development of a multi-DSAL application requires appropriate tool support. One standard tool is a dedicated debugger. A multi-DSAL debugger should support the inspection of a running application in terms of the AOP abstractions introduced by the different DSALs, as well as their collaborative interaction. Additionally, the debugger should support inspection of the composition specification, since the composition of the various aspect mechanisms itself may be the source for unexpected behavior in the composed program.

In this thesis the unique problems associated with debugging multi-DSAL applications were illustrated. A multi-DSAL debug interface (MDDI) was specified, and a corresponding implementation for the AWESOME composition framework was presented. The different implementation parts of MDDI include the formulation of dedi-

cated debug attributes, and a generic multi-DSAL debug agent that is integrated into the AWESOME weaving process. MDDI consumes the debug attributes and offers a set of debug operations to be used by a multi-DSAL debugger. An AWESOMEDEBUGGER command line tool was implemented to validate the overall debug infrastructure. The tool was used to analyze the source of different bugs that may be found in multi-DSAL programs.

We have focused on the debugging of a dominant family of reactive aspect mechanisms known as join point and advice [11]. The multi-DSAL debug infrastructure was implemented for an ASPECTJ-based environment. Yet, a major portion of the implementation may be reused in other setups as well. For example, MDDI may be utilized in a JBoss AOP environment [20]. For that, one would need to implement a multi-DSAL debug agent that provides the defined debug information to MDDI. While the multi-DSAL debug agent in AWESOME is integrated into the weaver, the JBoss debug agent will be a remote agent included in the JBoss AOP runtime, similar to the approach taken in AODA [4]. Debugging other non-reactive aspect mechanisms is a topic left for future work.

Appendix A

Developer Guide

The guide describes the internal structure of the debugger framework. It can be used by developers wishing to further develop the framework or port it to newer version of AspectJ.

A.1 Code Structure

The framework is made up from several projects that implement the various components, plus an example project.

A.1.1 awesome.platform

This project implements the base Awesome framework. The implementation is based on the AspectJ code (version 1.6.5). It includes the basic platform code without the different individual weavers. The base framework coordinates the overall weaving process and writes the binary class files that include the debug information. It was originally created by Kojarski et al [12] by refactoring AspectJ version 1.5.2 and ported to a newer version as describer in section A.2.

A.1.2 aw.coolajval

This project implements the "coolajval" weaver – a weaver that handles the AspectJ, Cool and Validate languages. It is made up of the individual weavers plus a configuration aspect.

A.1.3 awesome.adbbcel

This project implements the classes that read and process the debug attributes from the binary class files. It is part of the AODA framework and was extended to support multi-dsal debugging. We added support for the retrieval of new attributes introduced by the Awesome debugger. The new attributes are:

To support the retrieval of the new attributes, the following ADBBCEL classes were modified:

1. `AspectAttribute`: added support for the following properties
   (a) Mechanism name.
   (b) For each effect: its type in DSAL terms (i.e. `COOLLock`) and the originating DSAL source code lines.
2. `JoinPointGranularityAttribute`: this is a new class that represents a new debug attribute. It contains the required information for the granularity, visibility and advisability operations. It lists for each method all of the join point shadows it contains. For each such shadow it lists which mechanisms may advise it, and with which kind of advice.

A.1.4 AJDI

This project implements AJDI which is the interface which is used by the debugger application. It is an extension to the standard JDI used by Java debuggers and

originated from the AODA framework. It has been extended to support multi-DSAL debugging. AJDI uses ADBBCEL to read the debug information from the class files. The following changes were made:

1. Interface `AspectInfoProvider`: added `getMechNameForAspect` - this resolves the DSAL mechanism name for a given aspect. It is fetched directly from the aspect attribute of the aspect class.

2. Class `AdviceDescriptor` and the `Advice` interface, added two data members with accessors. These members are initialized from the values read from the Aspect Attribute.

   (a) `effectType` - the effect type in the DSAL terms, for example, COOLLock.

   (b) `sourceLines` - the line numbers in the DSAL source file represent this effect.

3. Interface `Advice`: added the `mechanismName` method. This is read directly from the Aspect Attribute.

4. Interface Method`Method`: added two methods. These methods are implemented by directly reading from the new `JoinPointGranularityAttribute`.

   (a) `exposedJoinPoints` - returns all join points in the method (used by the granularity operation)

   (b) `visibleJoinPoints` - returns all visible join points for a given mechanism (used by the visibility operation)

A.1.5 awdb

A simple debugger application that is based on AJDI (see appendix B). This project depends on AJDI and ADBBCEL.

A.2 Refactoring AspectJ to Create Awesome

The following classes should be refactored:

- aspectj\weaver\bcel\BcelAdvice.java
- aspectj\weaver\bcel\BcelClassWeaver.java
- aspectj\weaver\bcel\BcelMethod.java
- aspectj\weaver\bcel\BcelShadow.java
- aspectj\weaver\bcel\BcelWeaver.java
- aspectj\weaver\bcel\BcelWorld.java
- aspectj\weaver\bcel\LazyClassGen.java
- aspectj\weaver\bcel\LazyMethodGen.java
- aspectj\weaver\bcel\ProceedComputation.java
- aspectj\weaver\bcel\Range.java
- aspectj\weaver\bcel\ShadowRange.java
- aspectj\weaver\Advice.java
- aspectj\weaver\Shadow.java

The suggested way to apply the refactoring is to compare the current version of the platform against the AspectJ version it was derived from and apply the same changes to these files.

Appendix B

User Guide

This guide is intended for developers who wish to introduce a new aspect mechanism. It also explains how to use AwesomeDebugger.

B.1 Developing an Aspect Mechanism with Debug Support

An aspect mechanism is implemented as an aspect that extends the **`AbstractWeaver`** aspect. To add functionality for the mechanism, several of its methods should be implemented. Some of them handle the weaving process of the mechanism, and others add debugging support. In addition, an aspect mechanism may advise – and by that to refine – the overall weaving process of the Awesome platform. For example, advising the **`reify`** process may control which shadows are exposed from the aspects of the mechanism. The base aspect **`AbstractWeaver`** defines several useful pointcuts for that purpose.

The following methods control the weaving process of the mechanism:

- `public List<IEffect> match(BcelShadow shadow)`: should return a list of effects that advise the given shadow.

- `public List<IEffect> order(BcelShadow shadow, List<IEffect> effects)`: should sort the given effects that advise the shadow.
- `public void setInputFiles(IClassFileProvider input)`: this method is called by the weaver after the compilation of the source file is completed and the weaving is just about to start. This provides the mechanism with the compiled class files of the program.

These methods should be implemented if a debugging support is desired:

- `public boolean handledByMe(LazyClassGen aspectClazz)`: is called for aspects only; should return true if this aspect is processed by the mechanism.
- `public List<IEffect> getEffects(LazyClassGen aspectClazz)`: returns a list of all effects defined by the aspect.
- `public PerClause.Kind getPerClause(LazyClassGen aspectClazz)`: returns the per type of the class, e.g. singleton, per object.
- `public String getName()`: returns the name of the mechanism.

In addition, each effect ("advice" in AspectJ terms) introduced by the mechanism should be represented as a class. The class should implement the interface **`IEffect`** having the following methods:

- `public void transform(BcelShadow shadow)`: applies the effect for the given shadow.
- `public void specializeOn(Shadow shadow)`
- `public Member getSignature()`: returns the effect's method's signature.
- `public String getPointcutString()` :returns the pointcut string.

- `public ISourceLocation getSourceLocation()`: returns the location in the source of the effect.
- `ublic UnresolvedType getDeclaringType()`: returns the aspect that defines the effect.
- `public Test getPointCutTest()`: returns the pointcut test.
- `public String getType()`: return the type of the effect (in DSAL terms).

B.2 Compiling COOL Sources

COOL source files are compiled and weaved in two phases. In the first phase, the COOL source file is compiled into an annotated Java source file. This source file can be used as an input file to the COOL weaver. The COOL front end is part of the aw.coolajval project (see A.1.2) and is invoked by launching the `cool.frontend.translator.CoolFrontEnd` class. It will convert the source files given on the command line.

B.3 AwesomeDebugger User Guide

AWESOMEDEBUGGER is a simple debugger for multi-DSAL programs used to validate MDDI. It uses a command line interface similar to the original JAVA debugger (**`jdb`**). Once started, it displays the command prompt **`(awdb)`** and waits for user input. The **`help`** command lists all possible commands, and can also provide command specific help. The different commands examine the state of the remote VM while it is suspended (for example, after hitting a break point). When the remote VM is running the debugger does not accept input.

B.3.1 Supported Commands

B.3.1.1 `help [command name]`

If **`[command name]`** is given, short description of the command is printed. Otherwise it lists all commands that are available.

B.3.1.2 `attach <address>:<port>`

Attaches to a remote VM (using JDWP over TCP/IP) that is running in the specified address. The VM must be suspended and listening for debugger connections on the specified port. After the remote VM is attached it remains suspended.

For example, to make the JVM suspend on start and listen for incoming debugger connections it should be run with he following parameters: **`-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=4000`**

B.3.1.3 `detach`

Detaches from the remote VM. The remote VM resumes running.

B.3.1.4 `threads`

Lists all the running threads in the debugged VM.

B.3.1.5 `thread`

Switches the current context to the given thread ID (as given in the output **`threads`** command).

B.3.1.6 `cont`

Continues the execution of the program until it is finished or a break point is reached.

B.3.1.7 `classpath`

Specifies the class path where the compiled classes are found. Specifying the correct class path is necessary to enable multi-DSAL aware debugging.

B.3.1.8 `break <method>`

Places a break point at the begging of the given method.

B.3.1.9 `breakat <class id>:<line number>`

Places a break point at the the given location.

B.3.1.10 `where`

Prints the stack frame of the frame currently in context (including hook frames).

B.3.1.11 `show`

The **`show`** commands queries the application for AOP related information.

B.3.1.12 `show alljoinpoints`

Lists all joinpionts in the current examined method.

B.3.1.13 `showadvisability <joinpoint id> <mechanism name>`

Lists the mechanisms for how the given joinoint (with the ID returned from **`(show alljoinpoints`**) may be advised by the given mechanis.

B.3.1.14 `showvisiblejoinpoints <mechanism name>`

Lists all the joinpoints current examined method that are visible to the given mechanism.

B.3.1.15 `show advice <joinpoint ID>`

Lists all advice relevant for the given joinpoint.

B.4 Example Debug Session

This section explains how to:

- Compile and weave an example multi application using a multi-mechanism.
- Debug the application using AwesomeDebugger.

The files for the demonstration program are available with the rest of the MDDI source code at `http://aop.cslab.openu.ac.il/research/awesome/debugger/`. They are all included in "demo.zip". This zip file includes 3 Eclipse projects, which are required for the demo session. To use them:

1. Install Eclipse version 3.6.2 with the AJDT plugin version 2.2.0.
2. Start Eclipse with a new empty workspace.
3. Import the projects into the new workspace:
   (a) Right click in the empty Package Explorer
   (b) Import ... General ... Existing Projects in Workspace ... Select the archive file ...
   (c) Import all three projects

There are 3 projects now in the workspace:

1. aw.coolajval.example: This project holds the source code of the example application and provides different launch configurations for the different stages of the multi-DSAL development process.

(a) verify.buildstack.launch - builds the stack library (into stack.jar)

(b) verify.buildapp.launch - build the multi-threaded application (app.jar)

(c) verify.execute.launch - launches the application

(d) verify.execute.debug.launch - launches the application in debug mode (it will start suspended)

2. aw.coolajval: this project holds the source code of the three different aspect mechanisms used in the application.

3. awdbg: this project implements the command line debugger used in the example. It is run by launching the awdb.Main class.

To build the stack library and the application we use verify.buildstack.launch verify.buildapp.launch, respectively. This is done by right clicking on the launch file and selecting "Run As" and then the requested launch configuration.

After the stack library and the application are built and the corresponding jars are created, we launch the application using the verify.execute.debug.launch configuration. Once started, the application is suspended and waits for an incoming debugger connection. We are ready to execute AwesomeDebugger (`debugger.Main` class in the awdb project) and start debugging.

At the debugger command line, we first specify our class path (in this example, the full path to app.jar). This is required and enables the debugger to locate the compiled class files with the debug information.

```
(awdb) classpath
C:\aop\awesome_svn\workspace\awesome.coolaj.example.
verify\app.jar
```

Next, we attach to the suspended VM (that is running locally):

```
(awdb) attach 127.0.0.1:4000
Attached successfuly. Use 'cont' to resume
```

Now we place a break point:

```
(awdb) break base.BoundedStack.push
Added breakpoint [1] at base.BoundedStack.push
```

And resume the program:

```
(awdb) cont
```

Once the break point is hit we can start examining the running program as we saw in section 6.

```
(awdb)[89] Hit breakpoint [1] at base.BoundedStack.push
1 time(s)
        base\BoundedStack.java:21
(awdb) where
Thread 89 GoodWriter [0] base.BoundedStack.push
BoundedStack.java:20
TargetExecuted: false

Mechanism Aspect Type Source Location Skipped Effect Type
-----------------------------------------------------------

[1] base.GoodWriter.run base\GoodWriter.java:44
[2] . Source not availble
```

Bibliography

[1] *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD'12)*, Potsdam, Germany, March 2012. ACM.

[2] Y. Apter, D. H. Lorenz, and O. Mishali. Toward debugging programs written in multiple domain specific aspect languages. In *Proceedings of the 6th AOSD Workshop on Domain-Specific Aspects Languages (DSAL'11)*, Porto de Galinhas, Brazil, 2011. ACM.

[3] Y. Apter, D. H. Lorenz, and O. Mishali. A debug interface for debugging multiple domain specific aspect languages. In AOSD'12 [1].

[4] W. D. Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 173–184, Charlottesville, Virginia, USA, March 2009. ACM.

[5] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 331–344, Vancouver, British Columbia, Canada, October 2004.

[6] T. Dinkelaker, M. Eichberg, and M. Mezini. An architecture for composing embedded domain-specific languages. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*, pages 49–60, Rennes and Saint-Malo, France, 2010. ACM.

[7] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Proceedings of the 6th International Symposium on Software Composition (SC'07)*, number 4829 in Lecture Notes in Computer Science, pages 200–215. Springer Verlag, 2007.

[8] W. Havinga, L. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, number 5142 in Lecture Notes in Computer Science, pages 180–206, Paphos, Cyprus, July 2008. Springer Verlag.

[9] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 26–35, Lancaster, UK, March 2004. ACM.

[10] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, pages 247–263, San Diego, CA, USA, October 2005. ACM Press.

[11] S. Kojarski and D. H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 212–221, Shanghai, China, May 2006. ACM Press.

[12] S. Kojarski and D. H. Lorenz. Awesome: An aspect co-weaving system for

composing multiple aspect-oriented extensions. In OOPSLA'07 [19], pages 515–534.

[13] S. Kojarski and D. H. Lorenz. Identifying feature interaction in aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 147–157, Minneapolis, MN, May 2007. IEEE Computer Society.

[14] C. V. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, Northeastern University, 1997.

[15] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, Xerox PARC, Palo Alto, CA, USA, Feb. 1997.

[16] D. H. Lorenz and O. Mishali. SpecTackle: Toward a specification-based DSAL composition process. In *Proceedings of the 7th AOSD Workshop on Domain-Specific Aspects Languages (DSAL'12)*, Potsdam, Germany, March 2012. ACM.

[17] D. H. Lorenz and B. Rosenan. Cedalion: A language for language oriented programming. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'11)*, pages 733–752, Portland, Oregon, USA, October 2011. ACM.

[18] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 3–13, Portland, Oregon, May 2003. IEEE Computer Society.

[19] *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*, Montreal, Canada, October 2007. ACM Press.

[20] R. Pawlak, J.-P. Retaillé, and L. Seinturier. *Foundations of AOP for J2EE Development*. APress, 2005.

[21] G. Pothier, Éric Tanter, and J. Piquer. Scalable omniscient debugging. In OOPSLA'07 [19], pages 535–552.

[22] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, number 4089 in Lecture Notes in Computer Science, pages 98–113, Vienna, Austria, March 2006. Springer Verlag.

[23] H. Yin, C. Bockisch, and M. Aksit. A fine-grained debugger for aspect-oriented programming. In AOSD'12 [1], pages 59–70.

תקציר

פיתוח מונחה היבטים המשלב מספר שפות מונחות היבטים הוא גישה המערבת פיתוח בשפות מונחות היבטים כלליות כגון AspectJ בשילוב שפות ייחודיות לתחום (domain specific languages) כמו COOL. כל שפר ממומשת על ידי מנגנון היבטים יחיד (aspect mechanisms) כאשר המנגנונים השונים מתואמים ביניהם על פי מפרט ההרכבה (composition specification). מערכת המאפשרת הרכבה כזו מכונה מערכת הרכבת היבטים (aspect composition framework).

מאמץ מחקר עיקרי בתחום הוא השילוב והתיאום בין המנגנונים המממשים את השפות הללו (מנגנוני היבטים - aspect mechanisms) במערכות הרכבת ההיבטים. למרות זאת, כמעט ולא הוקדש מאמץ לפיתוח כלים אשר יהפכו שיטת פיתוח זו למעשית. יצירת סביבת פיתוח עם כלים ייעודיים לפיתוח מונחה היבטים ומרובה שפות הנה חיונית לצורך ההצלחה של גישה זו.

עבודה זו מתמקדת בכלי שכזה: מנפה שגיאות לתכנית מונחית היבטים מרובת שפות. מנפה השגיאות מאפשר למפתח לחקור את מצב התכנית בזמן ריצה ואת ההתנהגות שלה. בנוסף, המפתח יכול לתחקר את מפרט ההרכבה, ולתחקר את יחסי הגומלין בין מנגנוני ההיבטים השונים. יכולת זו היא חיונית מאחר ומפרט שגוי עלול לגרום לשגיאות מסוימות שלא קיימות כשאר מפתחים בשפה יחידה מבין השפות. בעבודה זו הצגנו מפרט לתיאור תשתית לניפוי שגיאות. כמו כן סיפקנו מימוש בעזרת מערכת הרכבת היבטים קיימת - Awesome.

מנשק לניפוי שגיאות בתכנות מונחה היבטים ומרובה שפות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים במדעי־המחשב

יואב אפטר

המחקר נעשה בהנחיית פרופ' דוד לורנץ

במחלקה למתמטיקה ומדעי־המחשב

האוניברסיטה הפתוחה

הוגש לסנט האו"פ

חשוון תשע"ג, רעננה, אוקטובר 2012