Toward Debugging Programs Written in Multiple Domain Specific Aspect Languages*

Yoav Apter
Open University of Israel
1 Univeristy Rd.,
Raanana 43107, Israel
yoav.ap@gmail.com

David H. Lorenz
Open University of Israel
1 Univeristy Rd.,
Raanana 43107, Israel
lorenz@openu.ac.il

Oren Mishali
Open University of Israel
1 Univeristy Rd.,
Raanana 43107, Israel
omishali@openu.ac.il

ABSTRACT

Debugging an application written in multiple domain-specific aspect languages (DSALs), one for each domain, is a complex task. Each DSAL introduces its own source level abstractions, which should be visible and traceable during the debugging process. A debugging infrastructure for multiple DSAL applications should also enhance the viewing and tracing of the interactions between aspects implemented in the different DSALs. We report on initial steps to define and implement a debugger for AWESOME, a co-weaving framework for composing multiple DSALs. The problem is illustrated through several scenarios, and design principles for a multiple DSAL debugging infrastructure are highlighted.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Design, Languages

Keywords

Aspect-oriented Programming, Domain-specific Languages, Debugging

1. INTRODUCTION

A domain-specific aspect language (DSAL) is an AOP language of limited expressiveness aimed at describing a specific crosscutting concern in the terminology of the domain. Multiple DSAL development refers to programming in several DSALs simultaneously. In addition, multiple DSAL development usually involves programming in one or more general-purpose languages.

*This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08.

DSAL '11, Porto de Galinhas, Pernambuco, Brazil.

An effective development environment for multiple DSAL programming needs to support not only the execution but also the debugging of multiple DSAL programs. While multiple DSAL execution is tackled by several works (e.g.,[5, 4, 2]), debugging has not attracted much attention. In this paper, we focus on multiple DSAL debugging. In particular, we focus on making the different DSAL source code abstractions visible and traceable. Visibility is the ability to represent the executing software system in terms of the programming language abstractions [1]. Traceability is the ability to trace a specific executing behavior back to its source code segment [1, 3, 8].

Our work is in the context of a DSAL composition framework called AWESOME [5]. AWESOME facilitates the integration of multiple DSAL weavers into a single composite weaver. AWESOME provides a composition platform and plug-ins that support the ASPECTJ, ASPECTWERKZ, and COOL [6] AOP languages. Developers may add their own DSALs by implementing additional AWESOME plug-ins.

The AWESOME framework handles two kinds of aspect interactions:

- *Foreign advising* [7]: the manner in which an aspect in one DSAL advises foreign aspects of other DSALs.
- *Co-advising* [7]: the manner of applying multiple DSAL pieces of advice to the same join point.

AWESOME offers a default resolution for these interaction types, and in addition allows users of the framework, who define the composite weaver, to customize the default interaction specification. However, errors in the specification of the aspect interactions might result in unexpected behavior, which is difficult to debug using current tools. Even if the specification is correct, errors in the application itself may be difficult to debug, because of the shortcomings of existing debugging tools that support only single-language programs. While it is possible to debug these programs using OOP debuggers, they do not reflect runtime visibility in terms of AOP abstractions. They also expose unintentionally some of the implementation details of the respective weavers [1].

The goal of our work is to extend AWESOME to produce not only a composite weaver, but also to provide debugging facilities for the different DSALs being composed. This extension of AWESOME builds on the work of De Borger et al. [1] where a reflective aspect-oriented debugging architecture (AODA) is introduced that facilitates AOP enabled debugging. A natural integration is for AWESOME to lay out the needed debug infrastructure for the different DSALs involved, and for AODA to provide the debugging services

```
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 }
```

Listing 1: A Stack implementation in Java

```
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 }
```

Listing 2: A synchronization coordinator in Cool

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

Listing 3: A logging aspect in AspectJ

themselves in the form of dedicated APIs. Unfortunately, AODA supports only the debugging of single language programs (either AspectJ or JBoss only programs) and it is not extensible to support the debugging of multiple DSAL programs. Therefore, the APIs offered by AODA need to be extended before they can be applied to multiple DSAL development.

2. PROBLEM ILLUSTRATION

In this section we present debug scenarios that illustrate the limitations of a standard debugger as well as the limitations of the AODA debugger in handling multiple DSAL debugging. We explain how a multiple DSAL debugger might handle these scenarios. The scenarios are based on an example given by Kojarski and Lorenz [5] in which the Awesome framework is used to implement a composite weaver for the AspectJ and Cool languages. Cool [6] is a DSAL designed for adding synchronization capabilities to Java programs. The composite weaver for AspectJ and Cool is denoted CoolaJ.

2.1 AspectJ/Cool Example

In Listing 1, a Java implementation of a bounded stack is presented. `Stack` defines two public methods, `push` and `pop`, where an `ArrayIndexOutOfBoundsException` is thrown upon an attempt to pop objects from an empty stack or push objects onto a full stack.

Suppose that this simple stack is to be non-intrusively enhanced with the following features:

- Multi-threading safety (synchronization), and
- Logging capabilities.

By applying AOP to the job, the original code is left unmodified. The core logic is separated from the added concerns. The synchronization concern is implemented in Cool. A separate `Stack` coordinator (an aspect in Cool terms) imposes the synchronization logic over `push` and `pop` in an aspect-oriented manner (Listing 2).

The coordinator enforces the 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`).
- `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 logging concern is implemented by an AspectJ aspect (Listing 3).

Finally, the Awesome framework is used to define a weaver for CoolaJ. The composite CoolaJ weaver is provided with the `Stack` class (Listing 1), the `Stack` coordinator (Listing 2), and the `Logger` aspect (Listing 3), and produces a composite program that implements a logged thread-safe stack.

The weaving process in Awesome does not resolve to a translation of coordinators to intermediate AspectJ code. Such a translation would have introduced and exposed in the target aspects (and in the Java classes produced from

them) synthetic join points that do not exist in the source, and thus would potentially result in incorrect behavior [7]. Instead, a different approach is taken where COOL coordinators are compiled directly. During the weaving process, COOL's synchronization is implemented by inserting a call to a lock (unlock) method before (after) each guarded method. Using this approach, any synthetic code is under our control and can be filtered out by the framework.

2.2 Debugging Foreign Advising

In COOLAJ, a composition specification dictates how aspects from the two DSALs interact. One of the specified restrictions that controls foreign advising states that an ASPECTJ aspect cannot around advise COOL lock/unlock computations. This ensures that the COOL locking and unlocking operations are always applied and not overridden by aspects. To demonstrate some of the problems that may arise during the debugging of multiple DSAL programs, assume a debugging scenario in which this restriction is ignored. That is, we define such an ASPECTJ around advice, and in addition, the advice does not call `proceed`. Consequently, the lock computation is omitted, and the program might behave unexpectedly.

A standard JAVA debugger severely limits the runtime visibility of the debugged program. Since the debugger is not aware of AOP abstractions, it cannot link the current state of the debugged program to the original source code abstractions. In our scenario, the developer may only indirectly infer that the synchronization mechanism is not functioning correctly, e.g., by stepping through the code and noticing that the guarded code is accessed by several threads at the same time. However, such an inference depends on a specific execution order, and often a collision may not be present in the examined execution. Moreover, when the same code is accessed by different threads in parallel, stepping through the code is difficult because at each step the current instruction is selected from a different thread. This causes the debugger to skip between source code lines in an unpredictable and confusing manner.

AODA lists several inspection features that can be applied to a join point such as inspection of all applied pieces of advice, inspection of past pieces of advice, and inspection of the program structure. Potentially, these features may help in resolving the bug. The developer may mark a method that is expected to be synchronized with an entry and/or an exit join point breakpoint. Then, by inspecting the applied and actually executing pieces of advice, the bug may be detected. However, since the AODA debugger does not support multiple DSALs, the developer is not provided with a complete picture. AODA does not consider the lock/unlock computation as an advice execution and thus will not list it as not being executed.

On the other hand, in a multiple DSAL debugging environment, the COOL extension will expose that computation as an advice. Hence, the developer will see it in the list of past advice as not executed and will conclude that the synchronization was not applied. Consequently, the erroneous around advice may be fixed, or the configuration may be changed to avoid such errors.

2.3 Debugging Advice Code

Other bugs may be the result of coding errors in the COOL coordinators themselves. For example, a bug may be accidentally introduced in the COOL coordinator presented in Listing 2 (line 9) by incrementing the `ind` variable instead of `len`:

```
7   on_exit {
8     empty=false;
9     ind++;
10    if (len==buf.length) full=true;
11  }
```

It would be difficult to locate this bug using a regular debugger. The developer may notice that the `ind` variable does not match the actual number of objects in the stack. But the cause for that discrepancy will be unclear, since the **on_exit** block is not visible to the JAVA debugger. This problem of visibility appears also when the AODA debugger is used, since the **on_exit** block is not considered an advice.

A multiple DSAL debugger recognizes the block as an advice and thus the developer will see the **on_exit** block as a past advice that has been executed, thus identifying it as a suspect. Then, the developer may either examine its code and look for the error, or place a breakpoint inside it and discover the error by running it step by step.

2.4 Debugging Co-Advising

According to the COOLAJ specification, when COOL and ASPECTJ co-advise the same join point, the lock (unlock) advice of COOL is executed before (after) the before, around, and after advice of ASPECTJ. Assuming this ordering is not explicitly set, an ASPECTJ advice may unsafely access a `Stack` object from multiple threads. If the advice actually modifies the object, we will get unexpected results.

Using a JAVA debugger the developer may tell that the `Stack` object is accessed without any locks. The developer will see that the guarded variables are accessed simultaneously from different threads. However the developer will not be able to tell why the lock computation was not executed. The single language AODA debugger does not give any more information here, as it does not consider the lock and unlock computations as a (past) advice.

In a multiple DSAL debugger a breakpoint may be set in the lock computation and it may help to realize that the ASPECTJ advice was executed before the computation. Consequently, the developer will reorder the advice or change the ASPECTJ advice code so it does not call any non thread-safe methods.

3. DEBUGGING INFRASTRUCTURE

In this section, an overall design for a multiple DSAL debugging infrastructure is described. As mentioned in Section 1, the infrastructure involves extending the AWESOME framework and the AODA infrastructure. Extending the AODA infrastructure involves modifying the APIs to be multiple DSAL aware. Extending the AWESOME framework involves the addition of debug information during the weaving process of each DSAL. It also facilitates the retrieval of that information during debugging.

3.1 Extending the AODA Infrastructure

De Borger et al. [1] suggest that anyone who is interested in implementing an aspect-oriented debugger should use the AJDI interface offered by AODA. AJDI is an aspect-oriented extension to the Java Debugging Interface (JDI). It extends some of JDI existing types and also introduces new ones.

AODA is divided into two layers. The first layer is abstract and language independent. The second layer is concrete and language specific. AJDI belongs to the first abstract layer. Other interfaces that should be implemented by those interested in defining debugging support for a specific language also belong to this layer. For example, writing a debugger for AspectJ first involves the implementation of the above mentioned interfaces in the abstract layer, where the implementation itself resides in the concrete layer. Only then the debugger should be defined using AJDI.

The necessary adaptations to AODA include modifying both the abstract and the concrete layers. In the abstract layer, AJDI should be refined to meet the new requirements that multiple DSAL development introduces. For instance, AJDI introduces a generic `Advice` type, yet in a multiple DSAL setup there is a need to distinguish between advices from different DSALs. Hence, the `Advice` type should be augmented with an indication about the originating DSAL.

AJDI was designed with AspectJ in mind and thus it introduces types such as `Aspect`, `Pointcut`, and `Advice`. A natural question is whether or not these AspectJ-like types are suitable for representing DSALs in general. The advantage of a common representation is obvious. Its limitation is that it may hinder visibility of the various DSALs involved. For example, when debugging programs developed in CoolaJ, both Cool coordinators and AspectJ aspects will be mirrored by the same `Aspect` type, instead of, e.g., using a dedicated `Coordinator` type for Cool. This is also relevant for other Cool code blocks which will be mirrored by generic types, e.g., an `Advice`.

Modifications to the concrete layer of AODA mainly involve adjusting the different components that manage the interfaces implemented by a debugger extension. Currently, these components control the operation of a single debugger extension. They should be refined to handle multiple ones.

3.2 Extending the Awesome Framework

An Awesome weaver is composed of several plug-ins, each corresponds to a specific DSAL. The platform provides weaving services shared by all plug-ins where in addition each plug-in provides its own unique weaving process. The debugging infrastructure shares similar characteristics:

- A central platform component handles the debugging work common to all DSALs.
- A DSAL extension, in addition to introducing its specific weaving process, may provide additional debugging functionality and metadata.

More specifically, the Awesome contribution to the debugging infrastructure is made up of two parts, a front-end and a back-end. The front-end is in the form of a compiler agent that is added to the compiler and weaver code. It adds debugging related metadata (e.g., line numbers, start and end of join points) to the bytecode during compile and weave time. This can be in the form of tags or annotations. The central platform component adds common debug information to the class files. In addition, each extension, when weaving its corresponding bytecode may provide additional extension specific information needed for debugging it.

The back-end is responsible to parse the metadata added by the front-end and to interpret the appropriate events generated by AJDI during the debug session at runtime. Here, the central platform component handles the common metadata, and dispatches extension specific events to the specific back-ends of the DSALs. Each back-end interprets the metadata generated by the front-end by directly reading the compiled class files. Using this metadata the debugger can build the static AOP relationships it needs. The back-end uses the real-time events generated by AJDI and the static AOP relationships to build the dynamic state of the debugged process in terms of AOP abstractions. For example, when entering the bytecode generated for an **on_exit** block, it will match it with the source code and add or remove the appropriate AJDI elements. This protocol between the front-end and the back-end is specific to each extension and is specified by the author of the DSAL extension.

4. CONCLUSION

The development of an application with multiple DSALs requires appropriate tool support, one of which is a dedicated debugger. The debugger should be aware of the different abstractions introduced by the DSAL extensions, and support their visibility and traceability during the debugging process. It should also facilitate the understanding of the various aspect interactions involved. In this paper the problem of debugging programs in a multiple DSAL environment was illustrated, and a corresponding extension to the Awesome framework and AODA was described. We believe that such a DSAL debugger is necessary to making a significant progress toward realizing the multiple DSAL vision.

5. REFERENCES

[1] W. D. Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *AOSD'09*, pages 173–184, Charlottesville, Virginia, USA, 2009.

[2] T. Dinkelaker, M. Eichberg, and M. Mezini. An architecture for composing embedded domain-specific languages. In *AOSD'10*, pages 49–60, Rennes and Saint-Malo, France, 2010.

[3] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *SC'07*, number 4829 in Lecture Notes in Computer Science, pages 200–215. Springer Verlag, 2007.

[4] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *OOPSLA'05*, pages 247–263, San Diego, CA, USA, 2005.

[5] S. Kojarski and D. H. Lorenz. Awesome: An aspect co-weaving system for composing multiple aspect-oriented extensions. In *OOPSLA'07*, pages 515–534, Montreal, Canada, 2007.

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

[7] D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *ADI'07*, pages 23–28, Berlin, Germany, 2007.

[8] G. Pothier and É. Tanter. Extending omniscient debugging to support aspect-oriented programming. In *SAC'08*, pages 266–270, Fortaleza, Ceara, Brazil, 2008.