The Potential of Omniscient Debugging for Aspect-Oriented Programming Languages

Marnix van 't Riet, Haihan Yin, Christoph Bockisch
Software Engineering group, University of Twente, 7500 AE Enschede, the Netherlands
{m.vantriet, h.yin, c.m.bockisch}@cs.utwente.nl

Abstract

Aspect-oriented programming improve program modularity and meanwhile decreases program comprehensibility, because it can alter the program behavior implicitly. Sometimes, the implicit behavior even varies in different runtime context. To fix bugs related to aspect-oriented entities, programmers need to fully comprehend their actual behavior before taking treatments. However, in some commonly encountered debugging scenarios, existing tools fall short in providing desired information.

In this paper, we have described two AO-specific debugging scenarios that require to use the program execution history. We discuss our design ideas, which are (1) a model defining AO-specific events and (2) a visualization consisting of three states, which shows different levels of details.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; D.3.2 [*Programming Languages*]: Language Classifications—Very high-level languages

Keywords Advanced-Dispatching, aspect-oriented programming, omniscient debugging

1. Introduction

Aspect-oriented programming (AOP), on one hand, improves the modularity of code and, on the other hand, complicates the local comprehension of programs. An advise can implicitly and arbitrarily alter the behavior advised program to any extent. A recent study [8] shows that 42 out of 104 reported AOP-related faults were due to the lack of awareness of interactions between aspects and other modules. To analyse faults in AO programs, programmers need appropriate tools, such as debuggers.

Several AO-specific debugging facilities have been proposed. Wicca [7] is a debugger which provides the view of the woven source code. AODA [6] introduced a dedicated AO-specific debugging model. Yin et al. [20] proposed a solution preserving AO-specific information which is lost after compilation. Apter et al. [2] introduced an interface for debugging programs written in multiple AO languages. All these facilities are designed for accessing the real-time runtime information. It is not possible for them to inspect a state in the past execution. However, a programmer needs to locate the root cause of a bug, which is always in the past execution before the bug is observed, and this task is usually time-consuming [19].

Omniscient debugging is a debugging technique that records interesting runtime information during the program execution and debug the recorded information after the execution. Omniscient debugging allows programmers to navigate the runtime information backwards. Therefore, it naturally fits the convention of finding the cause of a bug.

Omniscient debugging for AOP is an area worth to be explored. Nowadays, programming languages are not limited in some main stream languages. New ones become more and more popular in software design and implementation, mainly because (1) they provide new features, such domain-specific vocabularies and advanced modularities, which make the software easier to be programmed and maintained than using a traditional language; (2) tools, such as the AspectBench compiler [4] and Xtext [1], significantly decreases the cost designing a new language. However, the debugging facilities for new languages fall behind. When we develop omniscient debugging for AOP, we need to consider problems, such as which debugging information are needed and how to present it. Our experience and approach can be generalized and used in developing debugging facilities for other new languages.

As far as we know, the paper, which proposed the TOD extension [16], is the only work about the omniscient debugging for AOP. It describes how the extension is designed and implemented. However, it lacks an analysis about whether the AO concepts introduce any unique requirement in the omniscient debugging. In this paper, we performed a preliminary study about the potential of omniscient debugging for AOP. Contributions of this paper are:

- identifying two typical AO-specific debugging scenarios where omniscient debugging is helpful and discussing their generalities, and
- identifying four AO-specific events that should be recorded at runtime, and
- designing a execution navigator that is aware of AO concepts, and
- identifying limitations of the TOD extension.

2. AO-specific Debugging Scenarios

Existing works have discussed the potential changes, which are brought by introducing new AO programs to a system. Rinard et al. [18] classified the interactions between advice and methods into two main categories: *control-flow interaction* and *data-flow interaction*. The control-flow interaction represents how an advice changes the execution of the advised method. The data-flow interaction represents how an advice uses data shared by the advised method. Many works [9, 11, 13, 14] have discussed the problem of

CoCoS'13, March 25, 2013, Fukuoka, Japan.

interactions between aspects and it is called *aspect interferences*. The control-flow changes and the aspect interferences work in the similar way. Therefore, we put the aspect interference to the category of the control-flow change.

This section describes two concrete debugging scenarios with respect to the aforementioned two types of changes. We found that these scenarios are difficult to be debugged without using the execution history. To simplify our description, we use the same base program in Listing 1 for all scenarios. It is an Account class and its value is decreased when the customer withdraws money.

```
1 public class Account {
2   private float value;
3   public void withdraw(float amount) {
4     value −= amount;
5   }
6 }
```

Listing 1. An example base program.

2.1 Control-flow Change

An advice can augment, narrow, replace, or combine the execution of the advised program. For example, a *before* advice always augments the method execution, because it only adds functionalities to the method. Another example, an *around* advice can prevent the execution of other advices with lower-priorities at a shared join point. However, the control-flow changes are not always desired.

Take Listing 2 for example, the around advice conditionally proceed the advised method. This aspect is supposed to constrain the amount of the withdrew money. When the amount exceeds a threshold, the transaction cannot be committed. However, the programmer wrongly specify the condition on line 5.

```
1 public aspect TransactionLimits {
2   Object around(float amount) :
3     call(∗ Account.withdraw(float)) && args(amount) {
4     //FIXTO: if (value <= THRESHOLD)
5     if (value >= THRESHOLD)
6       return proceed(amount);
7     else
8       return null;
9   }
10 }
```

Listing 2. An aspect which conditionally invokes proceed().

In this scenario, programmers observe that some withdraw operations are not performed. The cause is that the **around** advise unexpectedly does not call **proceed**(). To generalize this specific example, the unexpected behavior is caused by a wrong composition at a join point. The composition is not always fixed and it sometimes varies according to the runtime context. To analyse a composition, programmers need the complete execution, which includes the advice executions, at a join point.

The existence of a bug indicates that bugs with the same characteristics may also reside in the system. In the scenario, programmers may want to verify whether all the around advices handle the **proceed**() call properly. To generalize, programmers need to find all the join points with required compositions in the execution. For example, the join points with an around advice which does not call **proceed**, the join points where both advices *A* and *B* are applied. Without using the execution history, it is difficult for programmers to perform these tasks.

2.2 Data-flow Change

An advice execution can modify the data that will be used or has been used by the advised method. For example, AspectJ allows to modify or replace arguments and/or the returned value of a method call by using an around advice. This may lead to the program execution to an unexpected state. In such situations, the programmer needs to know whether the problematic data is modified by advices.

Suppose the aspect TransactionCost in Listing 3 runs with the base program, the **around** advice deducts the transaction fee from the withdrew money on line 5. However, the calculation of the transaction fee is wrongly specified. At a certain point after paying the transaction fee, the program observes that the account state is unexpected. To detect the root cause, she first inspects the execution of withdraw() but finds no defect in it, then notices that the method is advised by an aspect if static tools are used, and finally finds the root cause in the **around** advice.

An advice may not modify any variables of the original system, because it is not dynamically executed or it does not contain any statement modifying a system variable. Therefore, it is not necessary to inspect the execution of such an advice. If execution history is used, the programmer can directly trace if a variable is modified by any advice before or afterwards.

```
1 public aspect TransactionCost {
2   Object around(float amount) :
3     call(∗ Account.withdraw(float)) && args(amount) {
4       //FIXTO: amount −= amount∗RATE;
5       amount −= RATE;
6       return proceed(amount);
7   }
8 }
```

Listing 3. An aspect that alters the state of the base program

3. Design Proposals

In our prior work [5], we classified AOP as one kind of advanced-dispatching (AD) languages and we have shown that AO concepts can be perfectly matched to the AD domain. Our work is carried out based on a framework for executing AD programs. Therefore, our omniscient debugger can be also used for other AD languages even though it is motivated by AOP. For each AD terminology used in the following descriptions, we intuitively explain it by mapping it to its corresponding AO concepts.

3.1 Event-Model for AOP

Existing omniscient debuggers proved to be capable of reconstructing the entire program execution based on recording relevant information of every method call, method exit and field write [10, 12, 15]. In AD programs, AD-specific events are required to be recorded and then inspected during debugging. The following listing introduces these AD-specific events.

(Un)Deployment of AD declarations. AD declarations can be thought as pointcut-advice pairs in AOP. Deployment means enabling the function. In AspectJ, pointcut-advice pairs are deployed by default. Other AD languages such as CeasarJ [3] allow AD declarations to be (un)deployed dynamically. Capturing this type of events allows the debugger to present when and where AD declarations are working.

Predicate evaluation. Predicate can be thought as dynamic pointcut in AOP. The predicate evaluation requires runtime values. Capturing predicate evaluations allows programmers to find why certain advice is or is not unexpectedly applied at a join point.

Composition rule evaluation. Composition rules can be thought as precedence declarations in AspectJ. Advices are ordered at a join point according to the composition rule evaluation. This

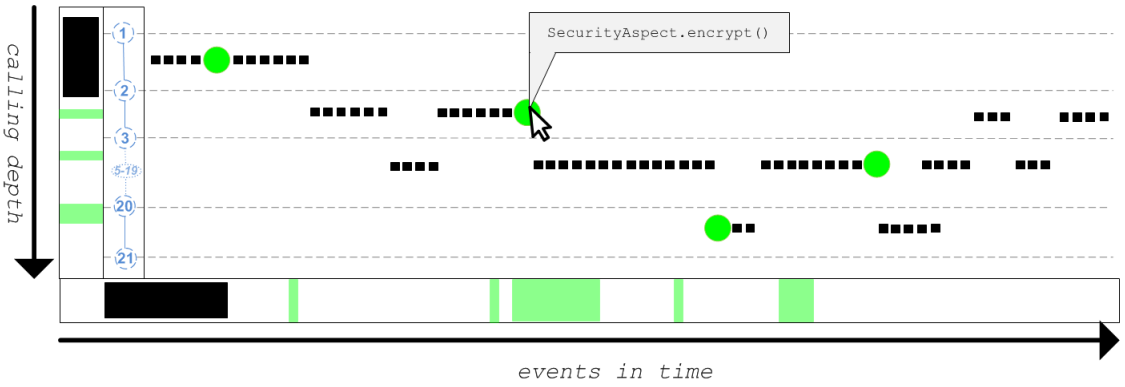

Figure 1. Conceptual design for presenting the program execution history and highlight query results.

event enables to inspect when and how a composition rule influenced the execution.

Action call and action exit. An action can be deemed as an advice body in AOP. Though we define action as AD-specific, it behaves the same way as a normal method does. To distinguish events from actions and from normal methods, it is necessary to record action call and exit.

Which runtime context needs to be collected for each event is still part of future work. Therefore, the event-model presented above is subject to further extension.

3.2 Visualization Proposal

Scalability and navigatability are two important challenges of visualizing a large dataset like the execution of a program. Our design visualizes the execution history by using the calling depth. A constructor or method call increases the calling depth and a corresponding return decreases it.

The visualization process consists of three stages and each stage only present a certain level of detail. Figure 1, which represents the first stage, show an overview of a query result. A query selects a subset of events. Each dot represents an event and the highlighted ones are the query results. Additional information is shown in a tooltip when hovering the mouse over a highlighted event. The vertical scrollbar allows scrolling to different calling depths. The depth count is labelled along side the vertical scrollbar. For example, depths from 5 to 19 are collapsed in the figure because they are irrelevant with the query result. The horizontal scrollbar allows scrolling forward and backward in time. The visualization projects the query result to both scrollbars and highlights the corresponding parts.

Figure 2.a illustrates the second stage. The nearby execution of a selected event from stage one is presented. Each rectangle represents to an event and the colored ones represent an activity of some aspect. For example, the colored *FW* rectangle has four colors and it represent a field-write event advised by four aspect. A legend shows what each color represents. Acronyms are used to show the event-type and rectangles can be expanded on-demand. The runtime contexts of the selected event is shown in a separate view. Therefore, programmers can inspect the values of local variables, fields etc.

The third stage of the visualization is presented in Figure 3. It shows full intimacy, that means all aspect-related events are expanded, based on the stage 2. For example, the colored AC event represents the aspect activity of the SecurityAspect and the normal FC event represents the advised function-call event.

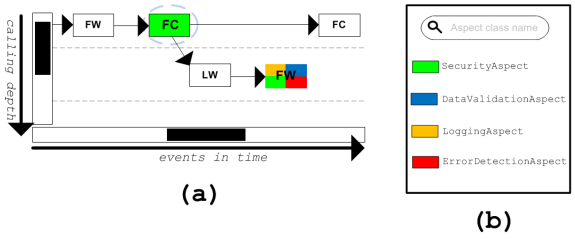

Figure 2. (Stage 2) **(a)** Detailed-oblivious view of the selected event in Figure 1. **(b)** An overview of the aspects that were active during program execution. Event acronyms: Field Write (FW), Function Call (FC), Local variable Write (LW)

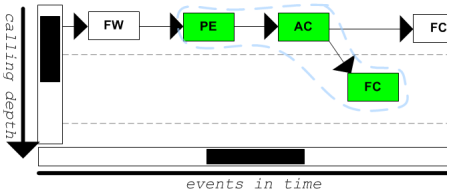

Figure 3. (Stage 3) Detailed-intimate view when inspecting the aspect-oriented intimacy of the highlighted event (encircled in blue) in Figure 2.a. Event acronyms: Field Write (FW), Function Call (FC), Predicate Evaluation (PE), Advice Call (AC)

There are two ways of interaction between the source-code and the visualization:

- Programmers can click a *inspect execution behaviour* button when they selected a line of source-code. The execution behavior associated with the source-code line is highlighted, like 1 shows. Programmers can inspect more details in way as we described.
- Programmers can select an event in the visualization and the corresponding source code line is highlighted.

4. Related Work

To our best knowledge, the TOD extension [16] is the only existing work about omniscient debugging for AOP. As the name suggests, it is built based on TOD [17], which is an omniscient debugger

for Java. To identify AO-specific activities, such as aspect instance selection and advice execution, in the recorded execution history, it uses a tagging scheme. AO-specific activities can be folded or completely expanded in the view of the history. Therefore, programmers can inspect the history in an appropriate intimacy levels. Besides, it provides an aspect mural view to highlight the activity of aspect in the history. We list the main differences between the TOD extension and the design of our work.

- It works on the woven code which does not preserve AO-specific information, such as precedence rules. Therefore, it cannot fully restore the runtime information with source-level abstractions. Our debugger will be developed on a ALIA4J framework [5] which models AO-specific concepts as first classes.
- It serves only AOP, which we classified as one type of advanced-dispatching (AD) mechanism. Our work targets on AD languages and, thus, it is more generic.
- It stores the runtime information in a woven form, that means it does not separate AO-specific information from the rest. Therefore, we can deduce that its underlying storage model does not contain AO-specific attributes. We will support AO-specific storage in our prototype.
- It provides limited ways for querying AO-specific information in the recorded execution history. For example, it cannot show whether a field was or will be modified by any advice, it does not support the evaluation of an arbitrarily given pointcut, etc. Our work will consider these AO-specific queries.

5. Conclusion

An omniscient debugger is helpful to localize the root causes in the two scenarios, because it records all interested events generated at runtime execution and reconstructs them off-line. In this paper, we presented our ideas about applying the concept of omniscient debugging to AOP languages. We described two AO-specific debugging scenarios require to use the execution history.

We discussed two important design choices within the domain of advanced-dispatching mechanism, which includes AOP. First, a model defining what AD-specific debugging data should be recorded. Second, a hierarchical visualization showing the recorded history. The visualizing process consists of three states, which shows different levels of details.

This paper is our preliminary study on the topic. In the future, we will explore more AO-specific debugging scenarios that fit the omniscient debugging, refine the event model accordingly, and implement the designed functionalities.

References

[1] Xtext framework. `http://www.eclipse.org/Xtext/`.

[2] Y. Apter, D. H. Lorenz, and 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*, AOSD '12, pages 47–58, New York, NY, USA, 2012. ACM.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Transactions on aspect-oriented software development i. chapter An overview of CaesarJ, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 87–98, New York, NY, USA, 2005. ACM.

[5] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at alia4j. *Journal of Object Technology*, 11(1):1–28, Apr. 2012.

[6] W. De Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 173–184, 2009.

[7] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Proceedings of the 6th international conference on Software composition*, SC'07, pages 200–215, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *Proceedings of the 32nd ICSE - Volume 1*, pages 65–74, New York, NY, USA, 2010. ACM.

[9] A. Hannousse, R. Douence, and G. Ardourel. Static analysis of aspect interaction and composition in component models. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 43–52, New York, NY, USA, 2011. ACM.

[10] C. Hofer, M. Denker, and S. Ducasse. Design and implementation of a backward-in-time debugger. In R. Hirschfeld, A. Polze, and R. Kowalczyk, editors, *NODe/GSEM*, volume 88 of *LNI*, pages 17–32. GI, 2006.

[11] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, FOAL '08, New York, NY, USA, 2008. ACM.

[12] B. Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.

[13] A. Marot and R. Wuyts. Detecting unanticipated aspect interferences at runtime with compositional intentions. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, New York, NY, USA, 2009. ACM.

[14] I. Nagy. *On the design of aspect-oriented composition models for software evolution*. PhD thesis, Enschede, June 2006.

[15] G. Pothier. *Towards Practical Omniscient Debugging*. PhD thesis, University of Chile, June 2011.

[16] G. Pothier and E. Tanter. Extending omniscient debugging to support aspect-oriented programming. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 266–270, New York, NY, USA, 2008. ACM.

[17] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 535–552, New York, NY, USA, 2007. ACM.

[18] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 147–158, New York, NY, USA, 2004. ACM.

[19] W. E. Wong and V. Debroy. Software fault localization. *IEEE Reliability Society 2009 Annual Technology Report*, 59(3), September 2010.

[20] H. Yin, C. Bockisch, and M. Aksit. A fine-grained debugger for aspect-oriented programming. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 59–70, New York, NY, USA, 2012. ACM.