A Graphical Tool for Observing State and Behavioral Changes at Join Points

Haihan Yin (ACM Member Number: 5775898)
Software Engineering group, University of Twente, 7500 AE Enschede, the Netherlands
h.yin@cs.utwente.nl

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; D.3.2 [**Language Classifications**]: Very high-level languages; D.2.2 [**Design Tools and Techniques**]: User interfaces

Keywords

Aspect-Oriented Programming, Debugging, Visualization

1. MOTIVATION

To comprehend programs or to fix a bug, programmers always mentally simulate the program execution by reading the source code. Aspect-oriented programming (AOP) increases this mental effort, because it can alter the state and the behavior of the base program at a join point to any extent by executing advices. Advices are implicitly invoked in the source code and their compositions at a join point may vary according to the runtime context. They can access and even change the context values of join points. As an example, consider listing 1. When Ship.rotate() is called, the **around** advice is executed. If Ship.rotate() is unexpectedly not executed, there are two causes: it is not called or it is bypassed by the **around** advice. Besides, the **around** advice modifies the parameter of the advised method call on line 3 and then passes the modified parameter to the original call. Without appropriate tools, it is difficult to notice the effects of the implicitly executed **around** advice.

```
1 void around (Ship ship): Ship.rotate() && target(ship) {
2   if ( ship.isAlive() ) {
3     ship.setPosition(...);
4     proceed(ship);
5   }
6 }
```

Listing 1: An advice example

The goal of my work is to increase the comprehensibility of AO programs by using a graphical tool, that can succinctly visualize the state and behavioral changes at join points.

2. RELATED WORK

AspectJ Development Tools (AJDT) can mark the places where join point shadows (JPS) are and uses a Seesoft view to navigate JPSs within a project. Noticing that AJDT does not scale well for large programs, Pfeiffer and Gurd proposed Asbro [8], which utilizes Treemaps to compactly show the distribution of JPSs. Except showing the existence of JPSs, AspectMaps [4] also provides static inspections to the composition at a single JPS. However, the static information cannot accurately show what actually happens at runtime.

Pothier and Tanter [9] implemented an omniscient debugger for AspectJ. It records runtime information executed by the woven bytecode. Therefore, some AO information, such as precedence declarations and pointcut evaluations, is lost after the compilation. My previous work [11] proposed a breakpoint-based debugger with an AO model that preserves all AO concepts at runtime. It can visualize the program composition at each join point at the granularity of advices. However, programmers need to manually investigate what happens within each advice.

Many works analysed the changes brought by introducing an aspect to an existing system. The changes may exist between aspects. Works [7, 6, 5, 3] have discussed the problem of aspect (or advice) interference. The changes can also exist between aspects and base programs. Works [2, 10] classify aspects according to how an aspect changes the control flow or the data flow of the base program.

3. APPROACH

Figure 1 shows some initial idea of the visualization. Suppose executing Ship.rotate() modifies property of the ship instance. Figure (a) visualizes the expected execution of listing 1. It abstracts away all details except those changes states and behavior. Rectangles represents executions of methods or advices. The black dot represents a join point and the arrow points to the actual runtime behavior when the corresponding join point is reached. A label is attached to the black dot and it is written what data is changed after the join point. Figures (b) and (c) show the situations when Ship.rotate() is not called and the around advice does not call **proceed**(). To realize the described functionalities, following components are required.

First, an omniscient debugger for providing runtime information of the interesting join points specified by programmers. The debugger obtains information from NOIRIn [1], which is an execution environment that models advanced-dispatching (AD) concepts as first class objects. AOP languages is one type of AD languages. Therefore, the debugger component can be also used for other AD languages. The obtained information is mainly salient runtime events, such as method calls and pointcut evaluations. Besides, the de-

AOSD'13 Companion, March 24–29, 2013, Fukuoka, Japan.
ACM 978-1-4503-1873-0/13/03.

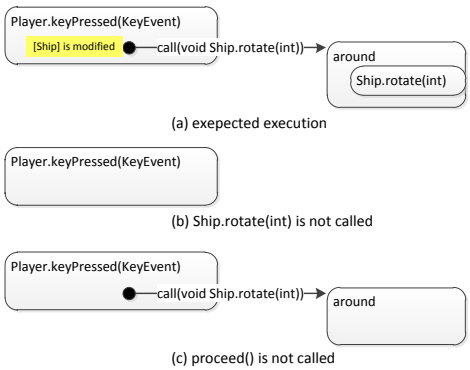

Figure 1: Initial design of graphical representation of AO-related executions.

bugger needs a well-defined interface to allow higher-layer applications, such as the graphical part of this tool, to use the recorded data.

Sometimes, inspecting one join point is not enough. It is helpful to know whether there are repeated cases in the whole execution history. An example is finding all the join points where the executions are similar to a known join point with undesired execution. Therefore, a query language for finding information in the recorded data is needed. To minimize the effort for learning the tool, I intend to reuse most of presentations in figure 1 to express queries. Figure 2 shows an example of the graphical query. It 2 matches all the join points where Ship.rotate(**int**) is called in the scope of Player.keyPressed() and an **around** advice is applied without calling **proceed**(). The dashed lines indicates the absent of the **proceed**() call. The graphical approach has two advantages: (1) queries, that are difficult to be described textually, can be specified easily by using graphs with expressions, (2) programmers do not have to learn a new query language. Specific user interface that allows programmers to compose graph elements are required.

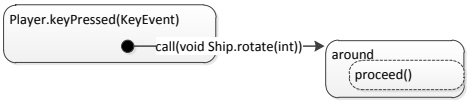

Figure 2: An example of the graphical query

Third, an algorithm for filtering information that needs to be rendered. There are many situations at a join point, some may skip the original call and some may end up with an infinite loop. In all situations, information, which is relevant with the state and behavioral changes, should be kept and rendered. A change is always related to what is supposed to happen and what actually happens. For example, an *around* advice can replace a parameter in *proceed()*. Programmers should be informed what is the replaced and the replacing parameters, and where the replacement took place. The information, which is irrelevant with the changes, should be hidden, like the **if** test on line 2 in listing 1.

4. RESULT AND CONTRIBUTIONS

The tool will be implemented as Eclipse plug-ins. To evaluate the tool, I intend to conduct an experiment measuring how far our tool decreases the comprehension effort for AO programs. Before the experiment, some common tasks requiring comprehension effort are designed. Examples of such tasks are determining whether a value is modified at a join point and finding out whether an advice is executed at a join point. In the experiment, subjects are assigned with the same set of tasks. They are asked to perform each task repeatedly by using the tools, which are AJDT, Asbro, AspectMaps, the omniscient debugger for AspectJ, and our tool. Suppose the longer the time is used, the more comprehension effort is paid. Therefore, I will measure the used time for each task and compare them at the end. The threats to validity include: (1)Subjects have different expertise in AO languages. Experts are likely to spent less time in finishing tasks. (2)The experience obtained from finishing the same task in prior tools may shorten the time spent on the following tools.

This is the first work that focuses on the visualization of changes at join points. Expected contributions, which correspond to the approach steps, of my work: an omniscient debugger, a graphical query language, and a filtering algorithm. Using the tool, programmers can view information, such as why a method is not executed at a join point, which values are modified by advices. Implicit behavior of AO programs can be abstracted and then explicitly visualized. In this way, the program comprehensibility is increased.

5. REFERENCES

[1] C. M. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Akşit. An in-depth look at alia4j. *Journal of Object Technology*, 11(1):7:1–7:28, April 2012.

[2] C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *In FOAL Workshop*, 2002.

[3] C. Disenfeld and S. Katz. A Closer Look at Aspect Interference and Cooperation. AOSD '12. ACM, 2012.

[4] J. Fabry, A. Kellens, and S. Ducasse. AspectMaps: A Scalable Visualization of Join Point Shadows.

[5] A. Hannousse, R. Douence, and G. Ardourel. Static Analysis of Aspect Interaction and Composition in Component Models. GPCE '11. ACM, 2011.

[6] A. Marot and R. Wuyts. Detecting Unanticipated Aspect Interferences at Runtime with Compositional Intentions. RAM-SE '09. ACM, 2009.

[7] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, 2006.

[8] J.-H. Pfeiffer and J. R. Gurd. Visualisation-Based Tool Support for the Development of Aspect-Oriented Programs. AOSD '06, pages 146–157. ACM, 2006.

[9] G. Pothier and E. Tanter. Extending Omniscient Debugging to Support Aspect-Oriented Programming. In *Proceedings of SAC*, 2008.

[10] M. Rinard, A. Salcianu, and S. Bugrara. A Classification System and Analysis for Aspect-Oriented Programs. SIGSOFT '04/FSE-12.

[11] H. Yin, C. Bockisch, and M. Aksit. A Fine-Grained Debugger For Aspect-Oriented Programming. AOSD'12, 2012.