

Towards a standard interface for runtime inspection in AOP environments

Katharina Mehner*
Department of Computer Science
University of Paderborn
D-33095 Paderborn, Germany
mehner@upb.de

Awais Rashid
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
awais@comp.lancs.ac.uk

Abstract

It has been pointed out many times that the integration of aspects with the programming environment is important. In this position paper we discuss support for runtime inspection of aspect-oriented programs. When developing software it is important to be able to access runtime information. It is even more critical for aspect-oriented systems where components may be unaware of crosscutting aspects potentially changing their behaviour. A particularly interesting area is the use of runtime information to detect inconsistencies between aspects which cannot be analyzed at compile time. Debuggers are the most prominent example for tools using runtime inspection. However, such support is also required by other runtime-based techniques such as tracing and monitoring. These techniques play an important role in testing, maintenance and also in debugging and have to be included in a general discussion of runtime inspection support. In order to avoid proprietary implementations for each tool relying on runtime information a standard interface is needed. We give an account of the state of the art in runtime inspection support for OOP environments before we examine the additional requirements imposed by AOP environments. From the requirements we derive an initial description of a standard interface for runtime inspection. We argue that such a standard interface should be grounded in a common foundation for AOP.[?]

1 Motivation

The key problem is that it is difficult to get an intuitive idea of the runtime behaviour of an aspect-oriented program. This is because components which are cut across by aspects are oblivious to them while the aspects may potentially change their behaviour by inserting additional code into the control flow. Furthermore, conflicts and dependencies between aspects are difficult to detect at compile time. In particular, aspect-oriented programs have powerful dispatch mechanisms, e.g., when intercepting method calls for executing additional behaviour. These dispatch mechanisms have a highly dynamic nature. Firstly, aspects in object-oriented environments are subject to the object-oriented mechanism of dynamic binding of method bodies to method calls which affects the binding of aspects, too. Secondly, in some AOP environments the definitions of join points employ conditions which are evaluated at runtime. Lastly, some AOP environments support dynamic aspects. All these factors make it extremely difficult to deduce the runtime effects and thus to develop and maintain code. If one adds a new aspect, a new class or creates new instances one can never be totally aware of the effects. A new class or instance may be unwantedly affected by existing aspects. Moreover, a new aspect can interfere with existing ones.

To date, most AOP environments only provide development support through so called structure browsers. AspectJ [1], for instance, provides support for IDEs such as JBuilder or GNU Emacs. The idea of a class browser is extended by showing which classes and method declarations are affected by aspects. Affected method call sites are also indicated. However, in general, affected method call sites can only be determined through data flow analysis because of dynamic binding and dynamic evaluation. Therefore, a structure browser can only show the static relations between the pieces of code. It remains difficult to obtain an intuitive idea of the behaviour at runtime. Therefore, we see a need for providing the programmer with such intuitive understanding. While formal methods are useful in this context, runtime inspection and analysis of a program is always needed. This is due to

* This work is supported by Lancaster University Research Committee Grant FAsOP-Foundations of Aspect-Oriented Programming.

current development practices which are quite often based on an incremental programming style. Furthermore, there are theoretical limitations of formal verification for large programmes.

We further motivate the need for runtime inspection with a simple example which shows that ordinary dynamic binding makes it difficult to understand which aspects are going to be used at runtime. The example is based on AspectJ which allows distinguishing between the *call* to a method and the *execution* of a method. We define the following classes A and B which are instantiated in main(), however under two references of class A.

<pre>class A{ public void m(){ //do something } }</pre>	<pre>class B extends A { public void m(){ //override A.m() } }</pre>	<pre>class Application{ static void main(String args[]){ A a1 = new A; A a2 = new B; a1.m(); a2.m(); } }</pre>
---	--	--

It should be clear that by means of dynamic binding the call `a1.m()` leads to the execution of the method `m()` from class A, while the call `a2.m()` executes the method `m()` as defined in B. We extend the example with two aspects. We assume they are executed in the order of their definition. Each aspect contains one after advice. The advice in aspect TraceA matches all calls to `A.m()`. The advice of TraceB matches the execution of `B.m()`.

<pre>aspect TraceA{ after(): call (void A.m()){ System.out.println("call to A.m()") } }</pre>	<pre>aspect TraceB after(): execution (void B.m()){ System.out.println("execution of B.m()") } }</pre>
---	--

Now let us consider support for understanding the example program. The AspectJ extensions for GNU Emacs or JBuilder will perform some static analysis to indicate two kind of relations between the aspects and the classes.

- ? Firstly, it is shown which method declarations are affected by the aspects. The aspect TraceA affects method `m` in class A and method `m` in subclasses B. Affected declarations are shaded in the same colour as the aspect TraceA. The aspect TraceB affects the method `m` declared in class B which is highlighted in the same colour as TraceB.
- ? Secondly, it is shown which method call sites are affected. The aspect TraceA affects the method call `a1.m()` and `a2.m()` in method `main()`. No information can be statically computed for effects from aspect TraceB. Again, the affected method call is shaded in the same colour as the aspect TraceA.

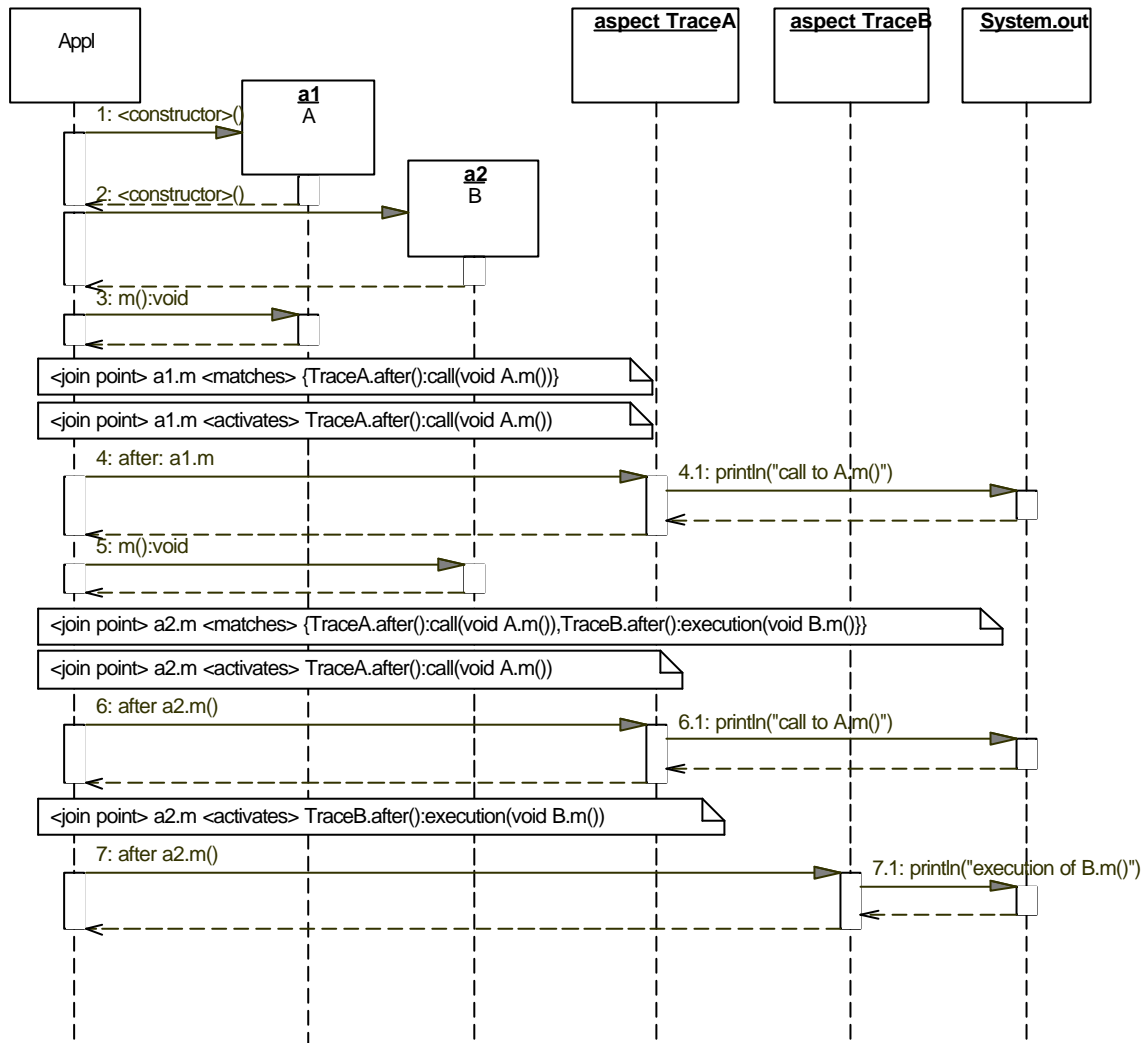
This gives the developer an insufficient picture of what will happen at runtime and, possibly, the illusion that nothing else might happen than what is indicated by the tool. The information derived in the static support has to be dealt with very carefully. In this example, at runtime the execution advice in aspect TraceB is activated for the call `a2.m()` in `main()`.

In such circumstances, where the behaviour of the code is difficult to understand, programmers often use debuggers and manual instrumentation (using printline-statements or macros). An understanding of the runtime behaviour is achieved by looking at typical scenarios, especially when an incremental programming style is used, also known as code-debug-fix-cycles. However, debuggers have shortcomings (in particular, they only provide access to one program state at a time but not to the runtime history). Manual instrumentation, on the other hand, is error prone and thus may lead to a wrong understanding (when important branches of the code are not covered by printline-statements).

Automatic tracing, i.e., generating a log of the program execution, can overcome these problems but there is a lack of proper tool support. New tracing tools should not restrict tracing information to plain statement level information but provide navigation, abstraction, and visualisation to foster better understanding. Similarly, instead of manually inspecting traces online or offline, which is again an error prone activity and might not be possible in the presence of huge amount of data, this could be

automated through monitoring tools. They also provide support for taking actions on the running program depending on the output of the analysis which is a very important feature.

Concerning our example we would like to run it with a tracing tool to understand its dynamic behaviour. Tracing tools primarily generate textual traces which are not very readable. Therefore we visualise a fictitious trace of the example with a UML sequence diagrams which of course has to be extended for that purpose. The diagram shows very nicely the actions of aspects and gives a complete picture of what happens at runtime.



Not only debuggers but also tracing and monitoring tools as described above rely on access to the runtime environment for gathering information about the execution. To avoid each tool coming up with a proprietary and limited solution (transforming code or changing the runtime environment) a standard interface is desirable. Such a standard is also important for interoperability between tools and AOP environments. Therefore, we see the need for establishing such a standard for runtime information. As can be seen from the example of debuggers, not only the inspection of the program state is important but also a certain degree of control over the running program is essential. Therefore, control issues also have to be addressed by a standard. Lastly, such a standard can be used as a reference model for comparing the runtime inspection capabilities of different tools.

For OOP environments there are runtime inspection interfaces, e.g., the Java Platform Debugger Architecture provided with the Java SDK [2]. It is used by many commercial debuggers but rarely by tracing and monitoring tools. IBM Jinsight is a powerful industrial prototype for visualising Java programs and analyzing problems related to OO [3]. JaVis is a research prototype for visualising and analyzing Java threads [4]. Both are based on the JPDA.

2 Proposed Approach

We first describe the AOP techniques falling within the scope of our approach. Then we give an account of OOP runtime inspection support as this is the closest background for our work and secondly, because many AOP environments extend OOP environments. We argue that in order to agree on an interface for runtime information we must first agree on a common model for AOP. Discussing such a common model in depth is not the focus of this paper. Therefore, we provide a coarse grained and short description of what we view as essential features of an AOP approach. Then we describe the requirements for a standard interface for runtime inspection.

2.1 Scope

The intended scope of this work is linguistic approaches to AOP such as CompositionFilters [5] and AspectJ [1] and non-linguistic approaches based on OOP and e.g. reflection mechanisms such as JAC [6]. Our aim is to define a generic model for runtime inspection capturing their commonalities while leaving room for adaptations to their variability. However, this presupposes more research on foundations of AOP than can be found in literature at present.

The proposed run-time inspection interface is also independent of a particular implementation technique for an AOP environment. Implementation techniques range from code or bytecode transformation to completely dynamic systems based on reflection. Any of the environments should be able to expose the proposed interface and make the desirable runtime information accessible. The implementation of the interface will highly depend of the implementation technique of the AOP environment.

2.2 Runtime inspection in OOP environments

There are two main directions for runtime inspection and analysis in OOP environments. One focuses on analysis of the functional behaviour, i.e. the program logic. This includes the internal state of a program and also program errors such as pointer problems or division by zero. Functional behaviour is addressed, e.g., by debuggers or tracing tools. The other direction deals with the analysis of resources like time and memory, also known as profiling. Here we focus on the behavioural part of runtime inspection. To date, the most comprehensive and comfortable API for the runtime inspection of OOP environments is the Java Platform Debugger Architecture (JPDA) [2]. The main part of the JPDA is an API implemented by each Java Virtual Machine. In the following we describe its main functionality and the conceptual model behind it.

The JPDA supports two main strategies for inspecting a running program. Firstly, it provides information about a running program without interrupting it by means of an *event model*. This model contains a set of predefined observable events for which event listeners can be defined by providing callback methods. The JPDA supports events such as method call entry and method call exit. Together with the information that an event occurred additional information about the event is delivered to the callback, e.g., caller, callee, and arguments of a method call. The information delivered with the event is limited. It does not contain the entire state of the program when the event occurred.

Secondly, the JPDA provides powerful means of *inspecting the entire state* of a program including heap and stack as well as call stacks. However, this information is only accessible if the program is stopped. Therefore, the JPDA provides different *means of control* over a running program. Many of them are used in debuggers. Here we give a complete account of them.

1. The control methods allow you to stop the program on any of the predefined events and to continue the program.
2. There are methods to define breakpoints and watchpoints at which the program will stop when they are evaluated positively.
3. Furthermore, the control methods allow you to stepwise execute the program either by stepping in or over a method.
4. There is also dedicated functionality for concurrent programs which allow you to stop and continue individual threads.
5. Limited support is available to force a thread to execute an arbitrary method.

There are also more advanced issues in runtime inspection which are not addressed by the JPDA. In a nondeterministic environment features such as enforcing a specific execution and replay of a recorded execution are needed, especially during testing. We consider this as an optional feature. Executing arbitrary code or changing the code while monitoring the runtime information can be seen as a specific control function over the running program. However, we think that this is not an essential but an optional feature for runtime inspection. Furthermore, reflection is so important and so difficult that it needs a separate consideration. However, any runtime inspection API should be able to allow the simultaneous use of a reflective API.

2.3 A generic AOP model

A standard interface for runtime inspection should be grounded in a common AOP model. Such a common model will help us to understand and clarify the crucial points in the execution of an aspect-oriented program. Existing AOP approaches support ideas of AOP to a different extent. Looking at several approaches will help us to reveal the nature of runtime behaviour which may be obscure or not clearly addressed or only partially supported by some approaches. The common model should be generic in the sense that it captures basic commonalities but can adapt to variability. In the following we identify features which we feel are essential to most AOP environments.

Common to most AOP approaches is the notion of crosscutting. Capturing crosscutting code in one place is achieved by encapsulating and modularizing the crosscutting code in *aspects*. Often one can find the idea of aspect classes and instances. Usually, aspects have an internal structure. They consist of weaves which contain the crosscutting code. Often, two forms of weaves are supported. One is the *advice weave* which executed when an event in the control flow is intercepted such as a method call. The other form is often called *introduction weave* and adds methods or fields to classes.

The next issue is the composition of the advice weaves. Sometimes there is composition with a dominant piece of code called. The most common notion for integration can be described in a layered fashion. Firstly, there is a *joint point model*, i.e., potential events in the control flow where composition can occur. A joint point event contains parameters related to the event. Secondly, the connection to the weaves which contain the additional code has to be made. This is by declaring, usually as part of the aspect code, which join points during program execution are to be *matched* by advice weaves. Definition of matches are often separated from the above mentioned advice weaves because they can be used by several advices. Advices can also be assigned to several matches. Thirdly, the advice weaves have to declare to what matches they apply. A last issue is dealing with the type of the advice weave which can be executed before, after or around the method call or similar things which it intercepts. In many environments the type is bound to the additional code in the advice but there is no need for this. Very little support can be found for matching advices to the occurrence of more than one join point, e.g., allowing to match a join point if another join point occurred before that in the execution history. However, this important feature should be explored in more depth.

We think that dynamicity is an important feature. AOP approaches provide different degrees of dynamicity. Many approaches incorporate dynamic evaluation of conditions during the match of a join point to decide whether a weave is to be executed at a join point. Other approaches use dynamic aspects. In some environments aspects can even change the structure of a program.

When more than one advice weave can match a join point questions about aspect order and dependencies arise. AOP approaches provide a certain degree of control over the interaction between aspects both at compile time and at runtime. If there is a runtime resolution of conflicts between aspects then this must be part of behaviour which can be inspected.

A discussion of runtime inspection should reflect the following commonalities of AOP environments:

1. Means for structuring aspect code: classes and instances, weaves
2. Different forms of weaves and types of advice weaves (before, after, around)
3. A model of join points
4. Declarations for matching join points
5. Binding matches to advice weaves
6. Dynamic evaluation of join points
7. Dynamic aspects

8. Dynamic conflict resolution for aspects

2.4 Runtime inspection for AOP environments

We have pointed out that the common model is particular useful for identifying which runtime behaviour has to be observable. Next, we derive from that model the required support for runtime inspection. We are particularly interested in extending the behaviour analysis and follow the idea of the JPDA model supporting observable predefined events, access to complete runtime state, and different means of control. We will not consider enforced execution, replay and reflection. Although profiling is also of interest we are of the view that it does not differ much from OOP profiling. Profiling is, therefore, not our focus here. Also we will not deal in depth with debugging support because we think that debuggers are very well understood while other runtime techniques lack support.

The model of the *predefined observable* events certainly has to be extended to cover events from the evaluation of join points and the execution of advice weaves, introduction weaves and other aspect code. It has to be defined what information is delivered with those events. The typical events from OOP should be extended to cover ordinary methods and fields in aspects and instantiation of aspects. This is the preliminary list of events and event information which we feel is essential.

1. Join point matches are events. Event information should describe the join point and its parameters. It should also contain all advice weaves matching the join point and the order in which they will be called. If dynamic conflict resolution was applied to produce this order rules or predicates and their evaluation should be supplied. It could also be of interest to supply the static rules involved in determining this list.
2. Execution of a weave is an event. The corresponding information should contain the type of advice (before, after, around) and distinguish entry or exit to the advice.
3. Call to aspects constructors and to ordinary methods within aspects are events. The information delivered with these events is similar to information delivered with object oriented method call, only the object called is replaced by the aspect called.

The general *control model* is not very different from OOP environments. Concerning more sophisticated control, extensions to breakpoints and watchpoints as well as stepwise execution are needed. It must be possible to set breakpoints and watchpoints at potential join points, also if they are currently not matched by aspects, at the evaluation of join point matches, at weaves and at ordinary aspect code. The stepwise execution must stop at join points, matches, advice weaves and aspect code.

The *inspection of state* should provide access to the state of all aspect classes and aspect instances as well as the state of a join point if the program stops in a state where a join point is involved. In the presence of dynamic aspects it must be possible to explore them.

3 Summary

In this position paper we have argued that support for runtime inspection is very important and that it has to be based on a general model. Access to runtime information can be essential to make aspect-oriented programming succeed in everyday programming, particularly for large programs. Therefore, a common understanding and tool support of runtime inspection is needed. Our initial description of a runtime interface is fairly basic and further research is needed. A runtime inspection interface has a complex nature not only supporting access to runtime information but also providing means of control.

Literature

- [1] <http://www.aspectj.org>
- [2] <http://java.sun.com/products/jpda>
- [3] <http://www.research.ibm.com/jinsight>
- [4] K. Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In S. Diehl (Ed.), Software Visualization Dagstuhl Seminar/Revised Papers. LNCS 2269.
- [5] http://trese.cs.utwente.nl/composition_filters
- [6] R. Pawlak et al.. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka (Eds.), Proceedings of Reflection 2001. LNCS 2192.