Trace Analysis for Aspect Application

Maximilian Störzer, Jens Krinke, Silvia Breu
Universität Passau
Passau, Germany
{stoerzer, krinke, breu}@fmi.uni-passau.de

May 30, 2003

Abstract

AspectJ is a language implementing aspect oriented programming on top of Java. Usually aspect application influences not only observable behavior but changes program flow internally. To test if an aspect works as intended, we suggest trace analysis to capture these internal changes. We demonstrate how trace analysis can be used for impact analysis. It can also be used to validate that refactorings which replaced scattered code by an aspect did not change system behavior.

1 Motivation

Aspect oriented programming (AOP) is a new paradigm in programming, extending traditional programming techniques, first introduced in [5]. Its basic idea is to encapsulate so called *cross cutting concerns* which influence many modules of a given software system in a new module called *aspect*.

Aspects can be used to add new functionality to an existing system which cannot be added without invasive changes to the whole system using conventional techniques. To see if the aspect works as intended testing is usually the only practicable approach, as program verification is often too costly and techniques applicable for AOP are not mature yet. However, tests are of limited value as they can only show the presence of bugs but never show their absence. Additionally, tests are of limited usefulness, if the changed behavior[1] of a system is not observable directly in the output.

Consider a simple example: An aspect should be applied to add an observer for some data to a system and the data provided by the system should be displayed by an additional GUI window. Instead of modifying the system to explicitly add the observer pattern an aspect can be applied instead, performing this task.

What remains is to check whether the observer implemented by the aspect really captures all necessary state changes of observed data. Unfortunately, this is not necessarily visible in any case. Static program analysis could be used for this purpose but data flow analysis of aspect oriented programs has not yet evolved enough. Thus, we propose a low-cost low-tech dynamic analysis approach: Using program traces, changed behavior becomes observable as here analysis of the internal program flow is possible. So this can be an adequate mean to analyze the effects of advice application to a given system.

For the methods presented here, one should always keep in mind that the quality of the results is strongly dependent on the coverage achieved by the test suite. Basically, traces are test results—only on a more fine grained level. Despite chances for false negatives[2] are low, we cannot guarantee their absence. The more traces are examined, the more reliable results of trace analysis are. Generally, the chances to miss a change in behavior strongly depend on the program coverage provided by the test suite.

Our approach relies on comparison of two traces for a single test: One trace is generated with the (base) system without applied aspects and the other is generated by the woven program. By identifying patterns of differences, we can observe changed behavior in terms of *divergence* and *superimposition*.

The remaining of this paper is organized as follows. Section 2 describes how aspect application impact can be determined by analyzing trace differences, including an example. The next section explains how this approach can be used to validate systems where cross cutting concerns have been refactored into aspects. Related work is discussed in section 4, followed by conclusions.

[1] Changed behavior throughout this paper refers to a changed input/output behavior of a given system.

[2] For this paper, by using the term *false negatives*, we refer to changes in behavior which are not reported by the analysis. *False positives* would be reported behavioral changes, which are actually not present in the program (which is impossible for our analysis).

2 Determining Aspect Impact

For the following we assume that for a given system S, a set of regression tests $\mathcal{T}$ is available so that traces $T(S,I)$ and $T(S',I)$ can be generated for identical input $I \in \mathcal{T}$. System S' is generated from S by applying an aspect A. The weaving operation is denoted as $S' = S \oplus A$.

2.1 Determining Impact from Traces

When applying an aspect to a given system S, the programmer should know exactly how and where the aspect influences the behavior of a system. Unfortunately, AspectJ language elements usually have global effects, so that aspects can influence large parts of a given system. As a result, the possibility to modularly reason about code is lost [2]. On the other hand, the influence of an aspect is not visible in the code of the modified system (obliviousness of base [3]).

Although with the AspectJ Development Tools `AJDT` a helpful development environment for AspectJ (e.g. in Eclipse) is available, more sophisticated tool support to determine aspect application impact on program behavior is not available. So, programmers have to rely on detailed tests instead. To tackle the problem that aspects do not always change the observable behavior of a given system, traces of test cases can be used. The generated traces reveal the internal program flow and thus are more fine grained than traditional output based tests.

To determine the impact of an aspect A which is applied to a system S, traces of the original system and the system with applied aspect $S' = S \oplus A$ can be compared. However, this comparison requires some constraints:

Identical Input. Certainly, the input for both systems has to be identical when generating the traces. This is usually the case when a regression test suite $\mathcal{T}$ is available.

Deterministic Execution. Traces are incomparable if system behavior includes random behavior—this case must be excluded. This is an important restriction as usually any concurrency in a system can lead to nondeterministic behavior.

In the following, we assume that these constraints hold for the generated traces $T(S,I)$. To analyze the impact of aspect A, we now generate traces for both versions of the program: $T_S = T(S,I)$ for the base system S and $T_A = T(S \oplus A,I)$ for the woven system.

Aspects are often used to add some additional behavior to a system while keeping the overall program flow (in a sense of *superimposition*). Thus, an aspect usually results in a relatively small set of additional calls in the trace T_A (depending on how much the aspect changes system behavior). These additional calls reflect the impact of the aspect on program flow. The programmer now can examine these *changes* to check whether all changes are expected or some of the changes are unexpected side effects of aspect application. However, would he have to examine T_A *completely*, he unneccesarily had to examine large parts of the trace which did not change at all.

Diff-like comparison of the generated traces can drastically reduce the size of the parts to be analyzed by revealing where program flow has changed. The result of a trace comparison of T_S and T_A is only a small—compared with the size of the traces—set of trace differences D.

In general, evaluating D derives very strong *hints* for the impact of aspect application. Two kinds of differences can be identified:

Primary effects: These effects capture *direct* aspect influence due to advice. Primary effects are always embedded in advice execution, so they can easily be identified in the traces and mapped to the source location where they originate from. An example of a primary effect is given in figure 1.

For primary effects, two sub-cases apply:

- *Superimposition:* Often, aspects are used to add some additional behavior to a given system. This is visible in the traces as a primary effect, which *extends* the original trace. So all calls of the original system still occur in the same order.
- *Changes or Amputation:* Using `around`-advice it is possible to prevent the execution of the originally called method and to replace the call with something else. In this case, parts of the original trace are *replaced* with different calls.

Secondary Effects: Besides primary effects, the trace difference D includes changes which cannot be identified directly as an effect of advice or another source code location. These effects are usually the result of a changed system state due to aspect application resulting in additional or other method calls. As secondary effects in general are a strong hint for changed system behavior, we consider secondary effects as *divergence* of traces.

Secondary effects can occur due to three reasons:

- *Introduction of fields:* AspectJ allows introduction of fields including *static initializations*. Execution of such initializations results in new calls to constructors which are not present in the original trace.

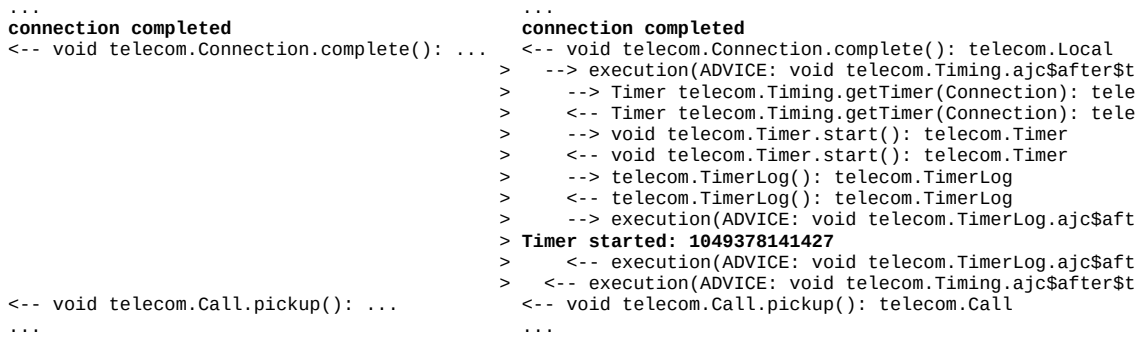

Figure 1: Primary effect—execution of advice.

This is the reason for the additional constructor call in figure 2.

- *Introduction of methods and ancestors:* At first sight introduction should not change the semantics of a given system. However, this is wrong as method introduction can override an implementation of a superclass and thus change the method actually executed by a virtual call.

  This effect—called *binding interference* in [9]—is observable in the trace difference, although the static analysis to reveal binding interference proposed in [9] is preferable as no false negatives can occur. Anyhow, this analysis is restricted to binding interference only.

- *Change of behavior:* Due to a prior advice application the system state—i.e. field values of some objects—might have changed. An altered system state can result in a completely changed program flow which obviously results in trace differences.

By examining only the trace differences *D*, a programmer can easily find the parts of his system influenced by the aspect. He can thus validate that his aspect was not unexpectedly applied to some parts of the system.

As a drawback, the proposed trace analysis does not show when an aspect has accidentally *not* been applied to some parts of the system. However, this is a general problem of any impact analysis. As an advantage of our approach, the programmer only has to check the trace differences for the intended changes—he does not need to examine the whole trace. If an expected change is not present, the aspect has to be improved.

To summarize, analysis of trace differences can show which impact the application of an aspect has on a given system by explicitly showing changes in program flow. So, the search space to validate aspect correctness is greatly reduced. For testing aspects that do not directly influence the observable behavior of a system, trace analysis is the only possible way to see how an aspect has been integrated into a given system.

2.2 Tracing the AspectJ Telecom Example

To create traces for Java programs, we used AspectJ and applied a specialized tracing aspect. Strictly speaking, we would have to verify that the tracing aspect does not change behavior of a given system—thus distorting the analysis results. But we think this assumption is reasonable as the tracing aspect never calls any method or changes any values of the base system. On the other hand, this aspect supplies a very simple way to generate traces for any given system. We could have used any other tracing technique without any effect on the results instead.

To evaluate the usefulness for trace analysis, we created traces for the `telecom` example available with the AspectJ distribution for two system configurations: the base `telecom` system and the woven system with applied `Timing` aspect. The input is always generated by the `BasicSimulation` test driver. Excerpts of the traces for the two versions is shown in figures 1 and 2.

The traces only differ by *blocks of additional calls*. Besides that, the trace is not modified, in particular it does not diverge. All calls traced in the base system preserved their order in the system with applied aspect. This suggests that application of the `Timing` aspect does not modify the overall system behavior but only superimposes some additional behavior.

A closer examination of the additional calls supports this assumption as from the base system only *getters* are called; any modification only affects data of the aspect or variables introduced by it.

Figure 2 shows how the new field `timer` introduced by the `Timing`-aspect is initialized by calling the constructor of the `Timer`-class. This represents a secondary effect, although our assumption that the semantics of the base system is only extended is still valid,

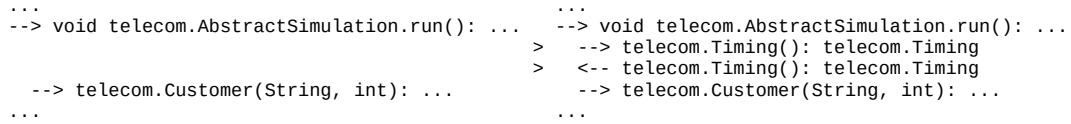

Figure 2: Secondary effect—static initializations, constructor calls.

because the constructor only affects fields of the new `timer`-object and does not call any other methods.

Indeed, the `Timing`-aspect is an observer, so this aspect only superimposes additional behavior for the base `telecom` application: It captures the duration of phone calls. Timing is always started before a connection is established and ends after the connection terminated (and indeed is implemented as after advice for call completion and hangup).

We also conducted some experiments with the `Billing`-aspect, which is basically an observer that calculates some derived data (the costs for each call), leading to similar results. Due to space limitations we do not present this example in more detail here.

As an example for diverging traces we added a new aspect `CostLimit` to the `telecom`-example which checks the total costs of all calls for a customer and denies connections if a certain limit is exceeded. In a first test run, the system did not act as expected, it crashes soon after a call has been denied:

```
jim calls crista...
Limit exceeded - no call possible!
crista accepts...
Exception in thread "main" java.lang.NullPointerE...
  at telecom.Customer.pickup(Customer.java:...
  at telecom.AbstractSimulation.run(Abstrac...
  at telecom.BasicSimulation.main(BasicSimu...
```

Although this example is very small, evaluation of the printed call stack does not reveal enough information to track down the problem. Obviously the `pickup`-method gets a `null`-value for the expected `Call`-object and crashes, but where does this value originate from?

Now consider the trace for this example, which is shown in part in figure 3. Here the problem becomes apparent: The trace shows that no connection object is created any more due to the around advice—the whole part of the trace creating this object is missing. When calling `pickup`, the trace immediately returns, reflecting moving up the call stack. Finally the stack trace is printed.

The advice application is a primary effect which results in the exception as secondary effect. The exception can be traced back to the `around`-advice. As a matter of fact, our `CostLimit`-aspect returns `null` when denying a call. This changed return-value causes the `NullPointerException` as the program proceeds. The base program is not prepared for `null`-values and crashes. So our aspect for limiting costs was a little naive here, but it demonstrates how the traces allows us to relate the crash to the applied aspect.

2.3 Identifying Failure Inducing Aspects

Without additional explanation, we modified our approach in the last section not to compare the base with the woven system, but compared two woven systems with different sets of applied aspects. In principle, we have relaxed the traces we compare. More formally, the comparison is done on traces for two system S_1 and S_2, where S_2 is produced by application of a set of aspects: $S_2 = S_1 \oplus A_1 \oplus \ldots \oplus A_n$. S_1 is allowed to be composed from a base system S with a set of applied aspects: $S_1 = S \oplus A_{n+1} \oplus \ldots \oplus A_m$. The traces to compare are then $T_1 = T(S_1, I)$ and $T_2 = T(S_2, I)$.

In the last example, the failure could have also been induced while comparing the base system with the woven system where all three aspects `Timing`, `Billing` and `CostLimit` have been applied. In that case it would not be obvious which of the three aspect has induced the failure. For large systems with a large set of aspects, identifying the failure inducing aspect is a cumbersome task. We use *Delta Debugging* [10] in that case: When aspects are seen as the deltas, this will automatically compute a minimal set of aspects responsible for the failure.

2.4 A Note on Multi-threaded Programs

Trace analysis can also be applied to multi-threaded programs, as long as the execution inside threads is still deterministic. As trace output generated by different threads is arbitrary interleaved, it must be able to identify trace messages from each thread which can be easily achieved in Java. Analysis can then be performed by comparing traces for each single thread.

3 Validating Unchanged Behavior

Besides the development of new systems, AOP can also be used to refactor pre-AOP software. For example, we can improve the program structure by encapsulating scattered code or crosscutting concerns into aspects. However, this may change the program behavior. If

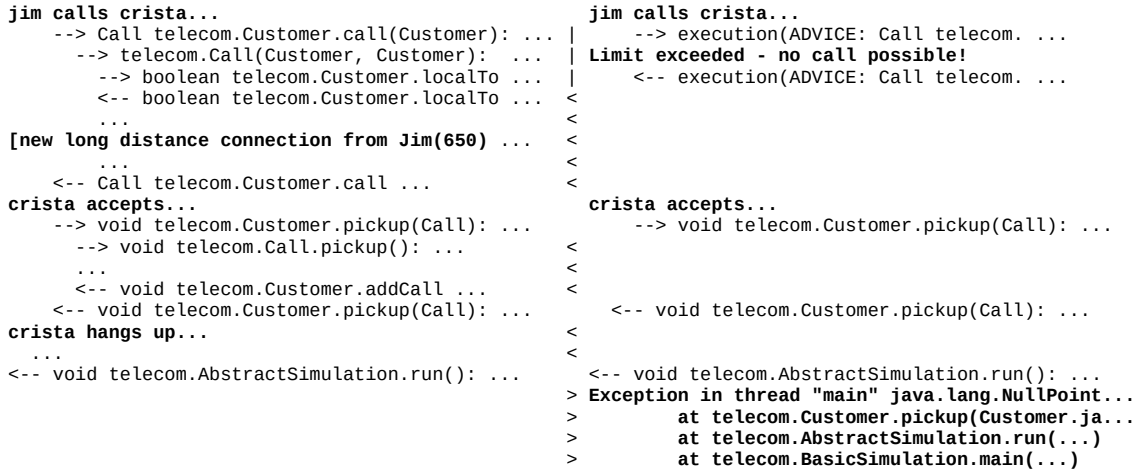

Figure 3: Primary effect—execution of advice.

these changes result in wrong program executions, this must be avoided.

The presented approach can be used to validate application of refactorings: Changes in program behavior become apparent if traces diverge. The approach now used is very similar to 2.2: We start with the old system S and refactor it into a new base system $S^\star$ together with the refactored aspect A. We are now comparing the old system S with the new system $S' = S^\star \oplus A$. Again, this is done by comparing the traces $T_S = T(S,I)$ and $T_A = T(S',I)$ for identical inputs $I \in \mathcal{T}$.

If refactoring has not changed program semantics, both traces should be identical. If the traces have differences, we know that a modified control flow is most likely and the encapsulation of crosscutting concerns into an aspect changed system behavior.

3.1 Acceptable Changes in Traces

In contrast to pure refactorings that should not change program behavior, changes are acceptable if they add additional functionality to the system, e.g. method calls for debugging purposes. Other sound alterations are syntactic: Depending on how the refactoring is implemented, the class from where a method is called or even method names can change. For example, imagine the following design decision: Shall just the method calls concerning a specific aspect be encapsulated or the class(es) implementing those as well? However, all these behavioral changes are acceptable as they do not alter the base system's behavior and its control flow; they just may add further functionality.

Such syntactic differences can easily be removed by applying a filter f to the traces before comparison: $T_S = f(T(S,I))$ and $T_A = f(T(S^\star \oplus A, I))$.

3.2 Example: AnChoVis

As a second example, AnChoVis, a small visualization tool for chopping and slicing, has been examined. Analyzing the produced program traces for standard aspects revealed a well known pattern: logging. By looking into the code we discovered that this functionality is scattered throughout the entire software.

Therefore, we have tried to encapsulate logging into an aspect in two different ways. First, as a simple aspect which uses the existing `Log` class of the system. It defines a pointcut which calls the `entering` method before each method execution, and after execution the `exiting` method of the logger. The original implementation concerning logging in the base system has been removed. The traces for program runs with the scattered and the aspect version have been compared and the (filtered) results are completely identical traces in case of equal program runs. As can be seen, the refactoring process didn't cause any changed or unwanted program behavior.

In the second aspect version, the aspect does not only include the logging calls but also the implementing functionality which was formerly in the `Log` class of the original system. Besides slightly different constructor calls of the logging functionality, the filtered traces are again identical.

4 Related Work

The so called *development aspects*, like tracing or profiling are the well known examples for the benefit of aspects [4]. However, we simply use these aspects for program instrumentation to get the traces which are the basis for our analysis.

Our approach is very similar to *relative debugging* [1], where a specialized debugger is used to automatically compare an old program version with a new one. This is achieved by inserting breakpoints and assertions to compare the values of fields at the same execution point in both programs. Instead of tracing method calls, we could use a different tracing aspect which traces values of fields. Thus, our approach can also be used for relative debugging.

In [8], profiling is used for software maintenance. In that work, *path spectra* generated by path profiling are compared for one program with two different inputs. Detected differences are evidence for diverging behavior. In contrast, our approach use the *same input* but *two versions* of a program. Comparing two versions with path spectra was mentioned but rejected in [8] because of difficulties establishing a correspondence between paths in two versions. Our approach has no problem identifying correspondence in tracing and explicitly identifies and uses not corresponding parts.

Traces are often used for software maintenance tasks, an exemplary tool is JInsight [6]. Traces can be too large and compaction techniques like in [7] can be used.

5 Conclusions

We have presented a low-cost low-tech dynamic analysis approach to analyze aspect application. This approach is based on comparison of traces for the base and woven system with identical inputs. Our approach is able to distinguish primary effects, which are due to aspect application directly, and secondary effects, which are most likely due to behavioral changes.

As traces capture an actual program run, not only obvious aspect influences from advice application are captured, but also more subtle behavioral changes due to introduction or (in general) secondary effects are revealed.

Our approach can also be used to validate that refactoring a pre-AOP system into an AOP-system does not change system behavior, thus supporting future reengineering tasks. Two examples demonstrated the effectiveness of our approach.

In this paper we only presented the application of our approach to small examples. However, the typical problems of large traces apply. On the other hand, trace analysis can be used to examine only parts of the system and thus is useful nonetheless.

As long as static program analysis cannot be used for this purpose because data flow analysis of aspect oriented programs has not yet evolved enough, our approach can be used instead. Moreover, our approach can be used best together with static analysis: It will not show false positives, only false negatives, and static analysis will show false positives and not false negatives instead.

References

[1] D. Abramson, I. Foster, J. Michalakes, and R. Socič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, 1996.

[2] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proc. FOAL Workshop*, 2002.

[3] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[4] G. Kiczales, E. Hilsdale, J. Jugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[6] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In S. Diehl, editor, *Software Visualization*, pages 151–162. 2002.

[7] S. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering*, pages 221–230, 2001.

[8] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. 6th European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449, 1997.

[9] M. Störzer and J. Krinke. Interference analysis for AspectJ. In *Proc. FOAL Workshop*, 2003.

[10] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proc. 7th European Software Engineering Conference (ESEC/FSE 99)*, pages 253–267, 1999.