Static and Dynamic Detection of Behavioral Conflicts between Aspects

Pascal Durr, Lodewijk Bergmans, Mehmet Aksit
University of Twente
{durr,bergmans,aksit}@ewi.utwente.nl

ABSTRACT

Aspects have been successfully promoted as a means to improve the modularization of software in the presence of crosscutting concerns. The so-called *aspect interference problem* is considered to be one of the remaining challenges of aspect-oriented software development: aspects may interfere with the behavior of the base code or other aspects. Especially interference between aspects is difficult to prevent, as this may be caused solely by the composition of aspects that behave correctly in isolation. A typical situation where this may occur is when multiple advices are applied at the same, or *shared*, join point.

In [1] we explained the problem of behavioral conflicts between aspects at shared join points. We presented an approach for the detection of behavioral conflicts that is based on a novel abstraction model for representing the behavior of advice. This model allows the expression of both primitive and complex behavior in a simple manner that is suitable for automated conflict detection. The presented approach employs a set of conflict detection rules, which can be used to detect both generic conflicts, as well as, domain- or application specific conflicts. The application of the approach to Compose*, which is an implementation of Composition Filters, demonstrates how the use of a declarative advice language can be exploited for aiding automated conflict detection.

This paper presents the need for and a possible approach to a runtime extension to the described static approach. The approach uses the declarative language of Composition Fillers. This allows us to reason efficiently about the behavior of aspects. It also enables us to detect these conflicts with minimal overhead at runtime.

An example conflict: Security vs. Logging.

We first briefly present an example of a behavioral conflict. Assume that there is a base system which uses a *Protocol* to interact with other systems. Class *Protocol* has two methods: one for transmitting, *sendData(String)* and for receiving, *receiveData(String)*. Now image that we would like to secure this protocol. To achieve this, we encrypt all outgoing messages and decrypt all incoming messages. We implement this as an encryption advice on the execution of method *sendData*. Likewise, we superimpose a decryption advice on method *receiveData*. Now imagine a second aspect which traces all the methods and possible arguments. The implementation of the tracing aspect uses a condition to dynamically determine if the current method should be traced, as tracing all the methods is not very efficient. The tracing aspect can, for instance, be used to create a stack trace of the execution within a certain package.

These two advices are superimposed on the same join point, in this case *Protocol.sendData*[1]. As the advices have to be sequentially executed, there are two possible execution orders here. Now assume that we want to ensure that no one accesses the data before it is encrypted. This constraint is violated, if the two advices are ordered in such a way that advice *tracing* is executed before advice *encryption*. We may end up with a log file which contains "sensitive" information. The resulting situation is what we call a behavioral conflict. We can make two observations, the first is that there is an ordering dependency between the aspects. If advice *trace* is executed before advice *encryption*, we might expose sensitive data. The second observation is that, although this order can be statically determined, we are unsure whether the conflicting situation will even occur at runtime, as advice *trace* is conditionally executed.

Approach.

An approach for detecting such behavioral conflicts at shared join points has been detailed in [1]. A shared join point has multiple advices superimposed on it. These are, in most AOP systems, executed sequentially. This implies an ordering between the advices, which can be (partially) specified by the aspect programmer. This ordering may or may not cause the behavioral conflict. The conflict in the example, is the case where the ordering causes the conflict. However there are conflicts, like synchronization and real-time behavior, which are independent of the chosen order.

One key observation we have made, is the fact that modelling the entire system, is not only extremely complex but it also does not model the conflict at the appropriate level of abstraction. With this we mean, that during the transformation, of behavior to read and write operations on a set of variables, we might loose important information. In our example we *encrypt* the arguments of a message to provide some level of security. Modelling this as a write on the arguments can work in some cases, however this makes expressing application specific conflict patterns hard. i.e. we do not want to consider all changes of all arguments of all messages conflicting. Also semantically, the *encrypt* operation does not change the value of the arguments, it only presents the data in a different form.

Our approach revolves around abstracting the behavior of an advice into a resource operation model. Here the resources present common or shared interactions (e.g. a semaphore). These resources are thus potential conflicting "areas". Advices interact with resources using operations. As the advices are sequentially composed at a shared join point, we can also sequentially compose the operations for each (shared) resource. After this composition, we verify whether a set of rules accepts the resulting sequence of operations

[1]Here, we only focus on join point *Protocol.sendData*, but a simular situation presents itself for join point *Protocol.receiveData*.

for that specific resource. These rules can either be conflict rules, i.e. a pattern which is not allowed to occur, or an assertion rule, i.e. a pattern which must always occur. These rules can be expressed as a regular expression or a temporal logic formula.

In [1], an instantiation of the presented model for the Composition Filters approach is shown. We adopted this approach, as the filter language is to a large extent declarative, and the compositional semantics are well-defined. This improves reasoning about the combination of multiple advice at the same join point. In addition, the filters provide encapsulation of the behavior through the use of filter types, which can be reused. However, there are elements which are filter instance specific and must be analyzed for each instance of a filter, such as the condition and matching parts of the filter. The conflict detection model can be enriched with domain or application specific information to capture more application or domain specific conflicts.

There are many steps involved in processing and analyzing a sequence of filters on a specific join point. One such step is to analyze the effects of each of the composed filters. A filter can either execute an accept action or a reject reject, given a set of conditions and a message. Next we have to determine which filter actions can be reached and whether, for example, the *target* has been read in the matching part. These actions perform the specific tasks of the filter type, e.g. the *Encrypt* action of filter type *Encryption* will encrypt the arguments. Likewise, the *Trace* action of the filter type *ParameterTracing* will trace the message. Most filter types execute the *Continue* action if the filter rejects. Imagine the following composed filter sequence on method *Protocol.sendData* in our example. The result is the following composed filter sequence:

```
trace : ParameterTracing = { ShouldTrace => [*.*] };
encrypt : Encryption = { [*.sendData] }
```

Listing 1: Composed filter sequence example.

This filter sequence can be translated to the filter execution graph in figure 1. The *italic* labels on the transitions are evalutions of the conditions (e.g. *ShouldTrace*), and the message matching, e.g. *message.sel(ector) == sendData*. The **bold** labels on the transitions show the filter actions. The <u>underlined</u> labels are resource-operations tuples corresponding to the evaluation of the conditions, matching parts and the filter actions.

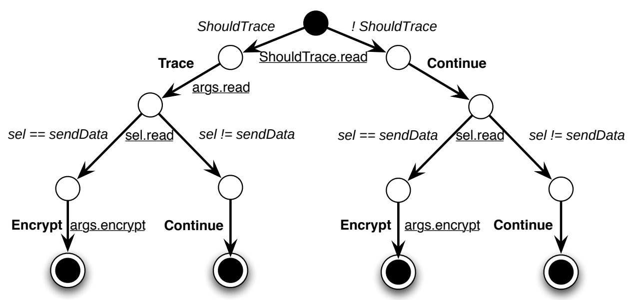

Figure 1: Filter execution graph example

From this graph we can see that in the left most path, the *arguments* are *read* before they are *encrypted*. This path thus violates the encryption requirement described in the example.

In Compose* we analyze the conflicts statically. However, it is not always possible to determine statically whether certain conflicts occur. There are three situations where dynamic verification is relevant:

1. If a program uses dynamic or conditional superimposition, and we detect a conflict in the program, we can only issue a warning at compile time. Only at runtime can we be sure that the conflict occurs.
2. Similarly, if the program uses conditional or dynamic advice execution. Here we also have to monitor the system at runtime to detect the conflict.
3. Concurrency can cause a wide variation of interleavings, including potentially conflicting sequences. This requires full monitoring of the advice at shared join points.

For the dynamic and conditional superimposition or advice execution, we can only issue a warning at compile time but we have to monitor the execution to detect the conflict at runtime. However, we can use the analysis results from the compile time to determine which paths of the composed program at a shared join point may potentially lead to a conflict.

As illustrated in figure 1, we have an internal representation of the sequence of filters at a shared join point. This representation is an execution graph, in which all the possible messages are simulated. Each end state of this graph corresponds to an unique path through the filter sequence. The graph branches if a condition is used within the filters. It also accounts for the various ways a message can flow through the filter sequence.

For each such path we know which conflict rule matches and which assertion rule rule does not match. We also know the transitions required to reach the erroneous end state. This information can be used to inject bookkeeping information at the transitions with are part of the path leading to the erroneous end state. This bookkeeping information performs the operations on the specific resources. If a, possible erroneous, final state is reached, we verify whether the conflict rules match or whether assertion rules do not match. If so, we can throw an exception, which can be handled by the user.

Conclusion.

The presented approach does not only provide feedback in an early stage of software development, i.e. while writing and compiling the aspect, it also provides an optimized way of checking whether certain conditional or dynamic conflicts actually occur at runtime. We only monitor those cases where it is known that a conflict could occur, but can not be completely statically determined. The declarative language of Composition Filters enables us to only verify those combinations which may lead to a conflict. It also enables us to reason about aspects without detailed knowledge of the base code, i.e. we only need to know the join points of the system, thus providing some form of isolated reasoning. Currently, only static verification has been implemented, in Compose*. However, we do plan to implement the proposed runtime extension in the near future.

This work has been partially carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program. This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

References

[1] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In R.Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Lancaster University, Jul 2006.