A Graph-Transformation-Based Semantics for Analysing Aspect Interference

Tom Staijen* and Arend Rensink**

Software Engineering Group/Formal Methods and Tools Group***
Department of Computer Science,
University of Twente,
The Netherlands
{*staijen,rensink*}*@cs.utwente.nl*

Abstract

AOP is widely accepted as a language concept to improve separation of concerns. However, the separate development of aspects may introduce semantic problems in the composition of the aspects and the base system. We propose a modular and graph-based verification approach. An aspect-oriented program is represented by a graph. A graph production system specifying the semantics of the language allows us to generate a transition system of the execution of the program. This can be used to analyse and verify different properties of the system. We show that, currently, it allows the detection of semantic differences between advice orderings on shared joinpoints, which is one of the semantic problems referred to above.

1 Introduction

Aspect-oriented programming (AOP) is a widely accepted language concept to improve *separation of concerns* on the implementation level. Before or during the execution of the program the behaviour of the aspects is imposed on to the base program (the program to be manipulated, specified in an underlying, object-oriented language). One of the major advantages of this is that is allows separate development of the base program and the aspects. However, since this also allows the base program and the aspects to be implemented by different (teams of) developers, the composed behaviour of the program might not always be understood. When different aspects apply at the same point in the execution of the base-program, they might interfere in their modification of the base program's behaviour. As pointed out in several publications and workshops [7,12,2,5,13], aspect interference is an important issue that should be detected in order to avoid errors.

Preprint submitted to Elsevier Preprint *16 June 2006*

We show in this paper that we can create a transition system of the execution of (part of) an aspect-oriented program — without any (semantic) abstraction — so that we will be able to detect occurrences of aspect interference. Because the verification of an augmented system (the merger of the base-program and the aspects) can be enormous, we take a modular verification approach; we want to analyse the composed aspects only and not the base program as well. Our choice of aspect-oriented language is guided by this requirement: we use Composition Filters [1] rather than the more widespread AspectJ language [11]. We present a graph-transformation-based approach to translating aspect-specifications to a state transition system of the execution of the aspects. We will show how this transition system can be used to find aspect interference. Section 2 will explain the problem in more detail, and introduce a running example. In Section 3 we will explain our solution on the basis of the running example and a brief more general discussion, followed in Section 4 by related work and in Section 5 by our conclusions and future work.

2 The Problem: Aspect Inteference

In aspect-oriented programming crosscutting-concerns can be specified in separate modules, so-called aspects. The behaviour of the aspects is often referred to as an *advice*. This advice will be executed on a certain pointcut. A pointcut is a set of joinpoints: points in the execution of the base-program where the advice is executed. The pointcut is specified by a pointcut specification, which is essentially a predicate over the potential pointcuts. In most languages the pointcut specification is part of the aspect specification.

2.1 Running Example

We will illustrate the problem of aspect interference by an example. In this example the base-system is a Database Management System. Two features of the system are implemented with aspects. The first is a *tracing* aspect that will log all database actions to be used for rolling back transactions. The second is a *authorisation* aspect that will abort any actions that are not allowed. Implicitly, the tracing aspect requires that, when an action has been logged, it will have to be executed afterwards. The authorisation aspect requires that, if a user is not allowed to perform an action, the action will be aborted.

As mentioned above, in our approach we use Composition Filters (CF). Listing 1 and Listing 2 show the CF source code of the Tracing and Authorisation aspect, respectively. We will explain the code, and by doing that also partially explain the language, below.

```
1 concern Tracing {
2
3         filtermodule Trace {
```

```
4            inputfilters:
5                    trace: Trace = { [*.execute] }
6        }
7
8        superimposition {
9                filtermodules
10                       traceTargets = { Classes | ClassByName{ Classes, '
                             Query' } };
11               superimposition
12                       traceTargets <- Trace;
13       }
14 }
```

Listing 1: Composition Filters sourcecode of the Tracing aspect

```
1  concern Authorisation {
2
3          filtermodule Authorise {
4                  inputfilters:
5                          auth: Abort = { !isAllowed => [*.execute] }
6          }
7
8          superimposition {
9                  filtermodules
10                         authTargets = { Classes | ClassByName{ Classes, '
                               Query' } };
11                 superimposition
12                         authTargets <- Authorise;
13         }
14
15 }
```

Listing 2: Composition Filters sourcecode of the Authorisation aspect

2.2 Composition Filters

In Composition Filters the declaration of aspects is independent of the language of the base-system. Communication between elements in the base-system is viewed as the sending of messages. Messages have a sender, a target, and a selector (the name of the message). Typically, for object-oriented base-languages, messages are method-calls.

To explain the working of Composition Filters we will go over the source code of the Tracing concerns in Listing 1. In Composition Filters both (base-language) classes and aspects are called *concerns*. A concern consists of zero or more *filter modules*. A filter module specifies the behaviour of the concern. Filter module *Trace* has one input filter. Such a filter will be evaluated when a message arrives at an object. The filter specification starts with the name of the filter *trace* followed by the filtertype, *Trace*. The rest of the filter is the MatchingPattern, which specifies the messages that the filter will accept. The *trace* filter will only accept message with selector *execute*. Depending on the type of the filter and whether the filter accepts or rejects, a corresponding action will be executed. For the *Trace* filtertype the accept-action is a TraceAction, which will add the method a log. The reject-action is a Con-

tinueAction, which will continue to the next filter, if any, or continues the dispatching of the method, otherwise.

Optionally a concern might also contain a superimposition specification, as the listings show. The *filtermodules* part specifies sets of classes by using prolog expressions. The *traceTargets* set in the example selects only class *Query*. In the second part the superimposition of filter modules on these sets of classes is specified. Thus, the Trace filter module is superimposed on class Query.

The source code of the Authorisation concern in Listing 2 does not differ much from the Tracing concern. The specified filter has filtertype *Abort*. This filter will execute an AbortAction (causing an exception to be thrown) when the filter accepts, a ContinueAction when the filter rejects. The MatchingPattern is preceded by a ConditionExpression, which refers to a boolean base-system method *isAllowed* that is evaluated at runtime. Thus, this filter will cause an exception to be thrown when *isAllowed* is false and the selector of the incoming message is *execute*.

2.3 The Example Problem Revisited

Depending on the implementation and the ordering of the aspects, the requirement of the tracing aspect can be invalidated by the authorisation aspect. In case of both aspects being implemented as so-called before-advice, and the logging aspect being the first to be executed, the action might be logged by the tracing aspect and then aborted by the authorisation aspect, thus invalidating the requirement of the tracing-aspect.

3 The Solution: Simulation-Based Analysis

Now that we have recognised the problem of aspect interference, we will show in this section that we can take the a program with aspects and generate a transition system of the execution of the (composed) aspects, using a graph transformation based operational semantics. We will then show that we can identify the occurrence of aspect interference from this transition system.

The process is shown in Figure 1. Given a Composition Filters program we generate a graph of the Abstract Syntax Tree. This graph will be referred to as the Abstract Syntax Graph. To this graph, additional control-flow information will be added. Taking the resulting graph as the start state, we will then simulate the execution of aspects specified through the filters by means of a small-step semantics expressed in *graph production rules*. This results in the transition system of the execution, another graph, where every node is a state but also corresponds to a graph by itself, and the transitions are the transitions specified by the production rules.

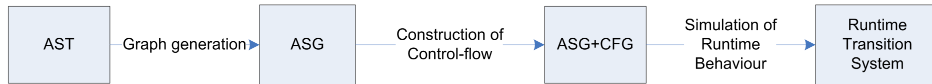

Fig. 1. Global description of the approach

3.1 Graphs and Graph Production Rules

Graphs are mathematical models with an intuitive and attractive visual representation. Graphs essentially consist of boxes — called nodes — connected by labelled arrows — called edges — possibly with labels on the nodes. In the context of this paper, the states contain the static structure, additional control flow information, and a part that represents runtime information. A graph production rule, in general, is a directive for changing graphs. It specifies a pattern of transformations between two graphs. A set of production rules is called a graph production system. We use graph production systems to specify the second and third transformation steps in Fig. 1.

For the purpose of the rules used in this paper, it actually does not make an essential difference what precise graph transformation formalism is used, since the rules can be formulated in either algebraic or algorithmic formalisms (cf. [17]. In point of fact, we have used GROOVE [16] as a tool to carry out the transformations and generate the state spaces; this means that the actual rules have been defined in the Single-Pushout approach (cf. [6]). Since the point of this paper is to illustrate an application, we omit the details of the formalism.

3.2 Generating the Abstract Syntax Graph

From the AST generated by the compiler, we generate an *Abstract Syntax Graph* (ASG), which is a kind of graph as explained before. By the time the graph is generated, the compiler has already resolved the superimposition part (establishing a relation between a class and a filter module) and the filtertype (which is replaced by the accept- and reject-action).

Figure 2 shows the ASG of the authorisation aspect of the example; to improve readability the Tracing filter module has been left out. A FilterSet node connects class *Query* to only one filter module, the *Authorise* FilterModule (again, the Tracing filter module is not shown in the picture). This FilterModule consists of one Filter of type *Trace*. The filter has one FilterElement consisting of a ConditionExpression, a ConditionOperator and a MatchingPattern. The MatchingPattern has a MatchingPart that will match all messages named *execute*.

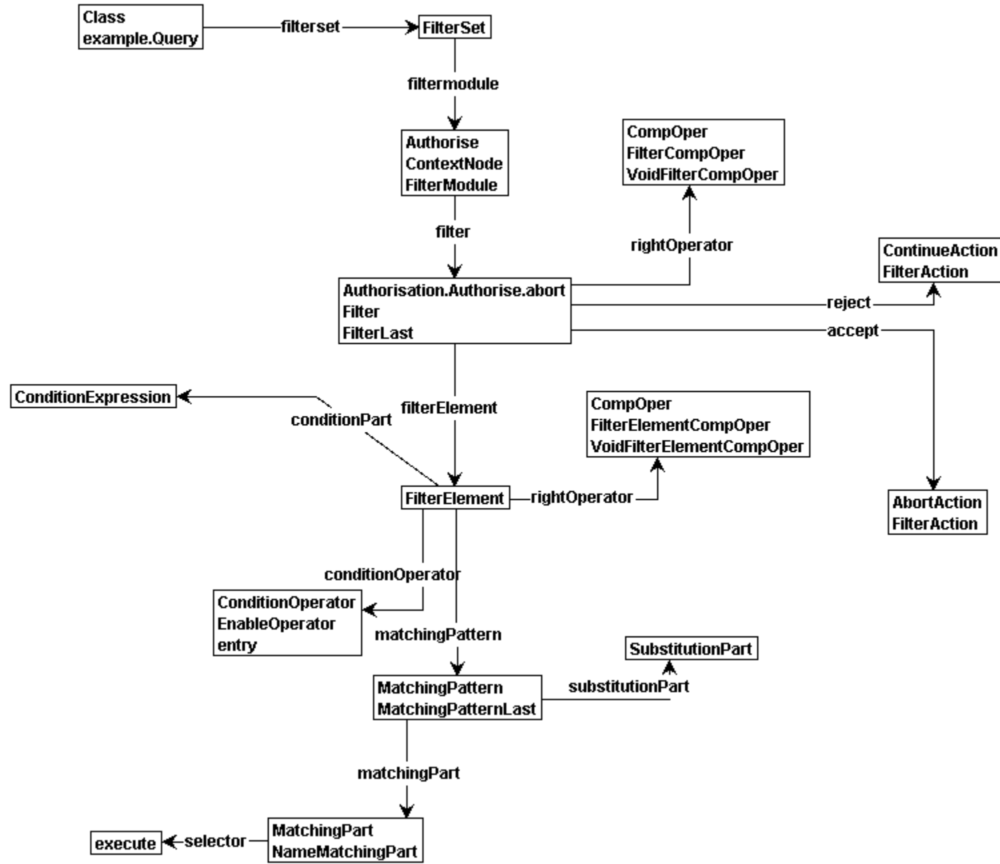

Fig. 2. Abstract Syntax Graph of the Authorisation concern

3.3 Control Flow Semantics

The next step is to add control flow information. The control flow semantics of Composition Filters is specified in a dedicated language developed in [18]; this includes an automatic translation to a graph production system that implements the second step of Figure 1. Applying this syst em to the ASG in figure 2 results in the graph shown in figure 3. The gray nodes and edges already existed in the source graph; the black nodes and edges have been added. These nodes and edges, together with the connected black nodes that are part of the control flow, constitute the Control Flow Graph. It consists of flow and branch edges; the latter lead to dedicated Branch nodes, which in turn identify the value under which a particular control flow branch is taken. The flow starts at a ContextNode — in our case the FilterModule node — and ends at the FlowConnector node that is the exit of the ContextNode.

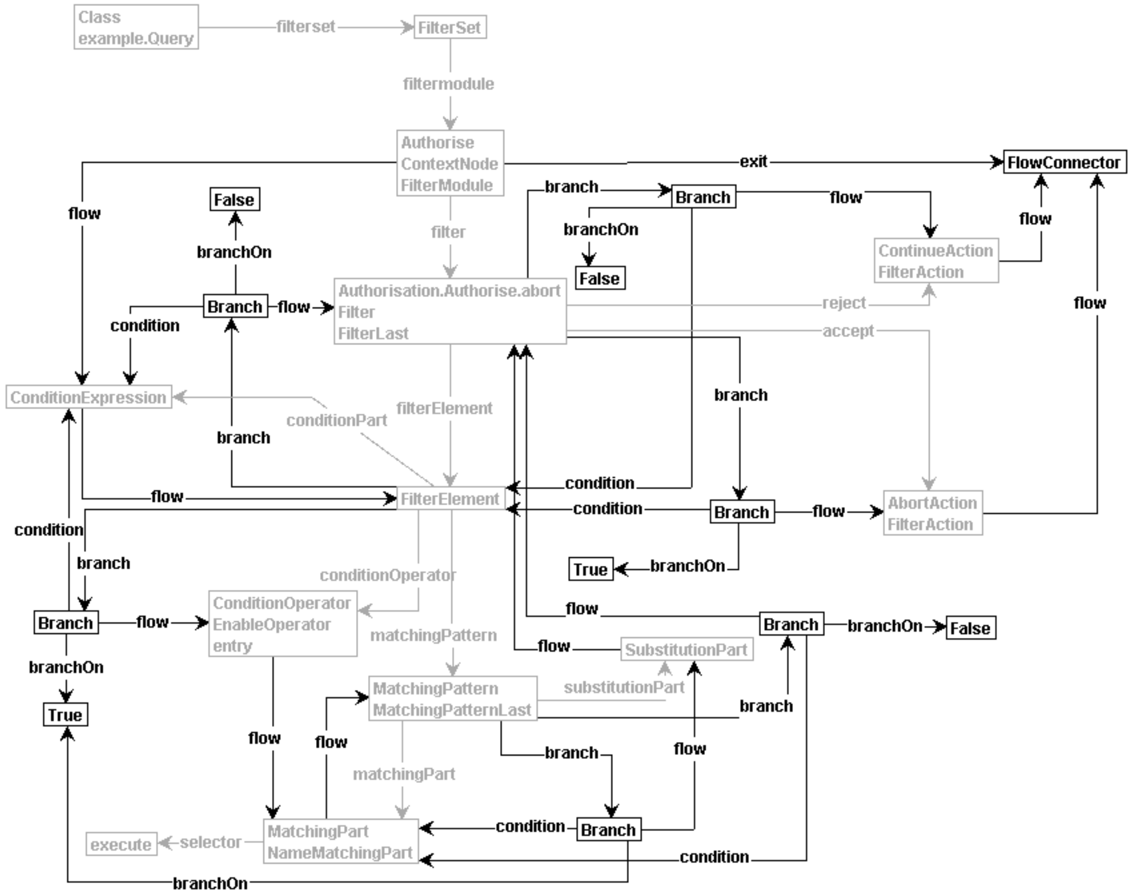

Fig. 3. Control Flow Graph (black) added to Abstract Syntax Graph (gray)

3.4 Runtime Semantics

For the simulation of the execution we use a production system where ever rule specified the runtime semantics of a single flow element. In other words, the production system is the graph-transformation-based small-step semantics. The result is another graph, where the nodes are states, and in fact graphs by itself, and the edges are transformations given by the rules of the production system. Figure 4 shows the resulting transformation system of the example program, with both aspects.

The enlarged part is the part in the transition system enclosed by a rectangle. This part shows a state *s34* with an outgoing transition *AbortAction*. State s34 is shown in Figure 5. The grey sub-graph is graph shown in figure 3. The black parts are part of the runtime structure. Notice that two *Frame* nodes are added. These frames relate to what we know as *stack frames* or *activation records*. The lower frame has a pc edge to the AbortAction. This frame is the stack frame responsible for the execution of the filter set. It also has edges for target and the selector of the parent frame: the stack frame

Fig. 4. Generated transition system of the example program

responsible for the method that is called. This method frame contains a self-edge to record its state. Currently, the frame is in a *filtering* state. The state can either be changed to *dispatch* to start the dynamic dispatch process, or to *abort* to indicate an exception will be thrown.

The state shown in Figure 5 matches the *AbortAction* production rule. This rule is displayed in Figure 6. Such a rule matches when the normal and dotted nodes and edges are present in the graph. During the application of the production rule, the dotted parts are removed and the bold parts are added. This rule matches when a pc edge connects a Frame node with the AbortAction node. During the transformation the Frame node — and all connected edges — is removed. The parent Frame, which is the Frame that would be responsible for the execution of the called method has its *filtering* edge removed and an *abort* edge added, indicating that the method will not be dispatched but an exception will be thrown.

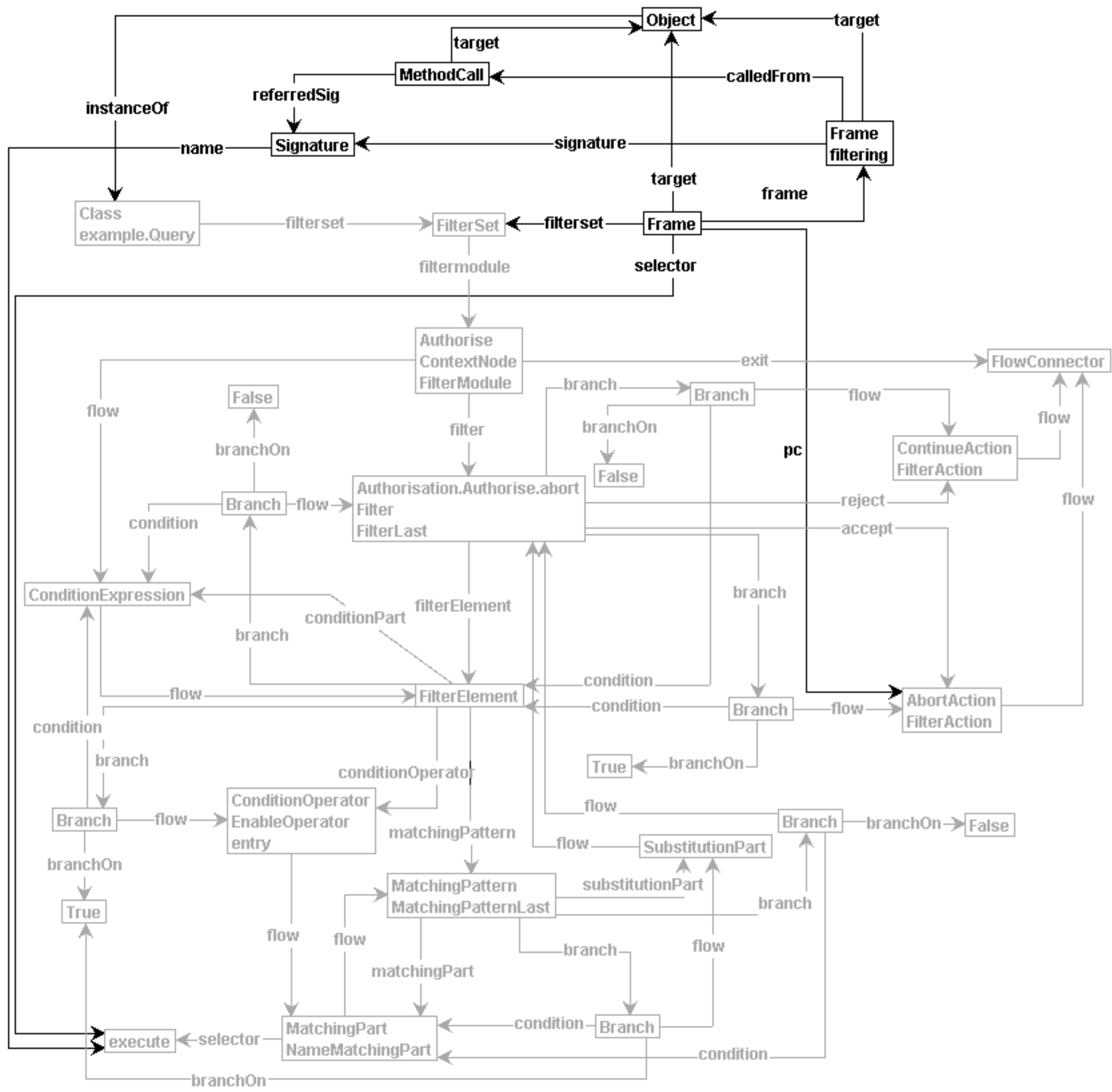

Fig. 5. State Graph before AbortAction (*s34*)

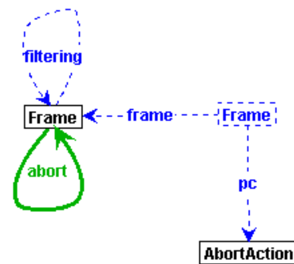

Fig. 6. AbortAction transformation specification

3.5 Analysis

We can identify two kinds of non-determinism in our simulation approach. The first is the resolution of unknown conditions, such as *isAtomic* in our example. The can influence the end-state of the simulation, since the matching of a filter might depend on such a condition. The second is the choice of imposition order. This *should* not have any influence on the resulting end-state: once the context of the filters is known, the filters combined should have an unambiguous behaviour. If the order turns out to make a difference, this signals aspect interference, which is a semantic conflict of the type we want to detect using our approach.

We choose to resolve the first kind of non-determinism — the outcome of unknown conditions — to occur first. The non-determinism caused by the choice of imposition order, on the other hand, should still result in a single end-state. If this is not the case, we can conclude the occurrence of aspect interference. In general we can say that, after the resolution of unknown conditions, the production system should be confluent.

In Figure 4 this analysis can be performed quite directly. The first branching is caused by the resolution of the *!isAllowed* condition, False on the left and True on the right. Notice that the value of the condition is attached to the entire expression, including the negation operator. The second branch is caused by the choice which filter module to execute first. In the False-branch, where the authorisation filter rejects the message, the paths corresponding to the different filter compositions join at the end. In the True-branch, however, the AbortAction will be executed, and the two branches corresponding to the different filter compositions do not join, but instead result in different states. From this result we can conclude that aspect interference occurs between the filter modules in the example. We can also say that this occurs when the condition — *!isAllowed* — is True (or *isAllowed* is False).

3.6 Method

Above, we have discussed our approach largely on the basis of the running example. The actual method involves giving both a control flow semantics and a run-time semantics to a language, in this case Composition Filters. As mentioned above, the first was done using a dedicated language for this purpose, developed in [18]; the run-time semantics, on the other hand, has been defined on a more *ad hoc* basis for the particular language at hand — Composition Filters — essentially by defining a single rule for each type of syntax element of the language. These rules, once fixed, allow the method to be applied to any CF specification; so, we can claim to have a working verification method for aspect interference, based on a graph computation model for aspect orientation.

4 Related Work

There has not been that much work on interference analysis between aspects. In [4], Douence, Fradet and Sudholt present a framework to identify overlapping joinpoints and detect possible aspect interference. The abstract formalism can be a basis for future analysis tools, but these have yet to be implemented. In [5] we propose a semantic conflict detection model for Composition Filters. This model translates the semantics of filter action to operations on resources. The desired behaviour can be specified by means of patterns of operations on a resource, either being a conflict or a requirement. To be able to detect aspect interference, a pattern must exist that can identify the faulty execution. However, the abstraction level of the resource-operation model from the exact behaviour is quite high. Depending on the kind of the interference, it might not be possible to describe the behaviour of the filters on this level such that it is still possible to detect the problem. Pawlak, Duchien and Seinturier [14] present a language called *CompAr*, which allows the programmer to abstractly define an execution domain, the advice semantics and the execution constraints of around advices in order to check if the execution constraints are fulfilled when the aspects share a joinpoint. The difference with our approach is that the aspects need to be specified in another language in order to be analysed where our approach uses the aspect specification directly.

The work reported in this paper is based on a graph transformation-based operational semantics of Composition Filters, an aspect-oriented language. The basic idea of using graph transformations for operational semantics is far from new: it ranges from a term graph-based semantics for functional languages (see Plump [15] to graph-based semantics for actor languages (see Janssens [9] and visual languages (e.g., [8]). For object-oriented languages the first approach of this kind is by Corradini et al. [3]; the approach of this paper is inspired by [10].

5 Conclusions and Future Research

Our approach generates a model of the execution from an aspect-oriented program specification. This is done by expressing the program as a graph and the semantics of the language as a production. An analysis of the model allows us to identify any occurring aspect interference on shared joinpoints. Composition Filters suits very well for our method since its aspect-specifications are independent of the base-language.

Future Work.

Our goal is to construct a framework for the behavioural modelling and verification of aspect oriented software. Currently, only a simple verification

technique is used to detect semantic differences caused by aspect orderings. Future work should result in allowing other kinds of conflicts to be detected. Model checking techniques can be used to validate the behaviour introduced by the aspect to make sure the intended behaviour of the base program is not invalidated by any aspect.

Acknowledgement.

This work has been carried out in the context of the European Network of Excellence on Aspect-Oriented Software Development, IST-2- 004349-NoE.

References

[1] Bergmans, L. and M. Akşit, *Principles and design rationale of composition filters*, in: R. E. Filman, T. Elrad, S. Clarke and M. Akşit, editors, *Aspect-Oriented Software Development*, Addison-Wesley, Boston, 2005 pp. 63–95.

[2] Clifton, C., R. Lämmel and G. T. Leavens, editors, "FOAL: Foundations Of Aspect-Oriented Languages," 2004.
URL http://www.cs.iastate.edu/FOAL/

[3] Corradini, A., F. L. Dotti, L. Foss and L. Ribeiro, *Translating java into graph transformation systems*, in: H. Ehrig, G. Engels, F. Parisi-Presicce and G. Rozenberg, editors, *Second International Conference on Graph Transformation (ICGT)*, Lecture Notes in Computer Science **3256** (2004), pp. 383–389.

[4] Douence, R., P. Fradet and M. Südholt, *A framework for the detection and resolution of aspect interactions*, in: *1st Conf. Generative Programming and Component Engineering*, Lecture Notes in Computer Science **2487** (2002), pp. 173–188.

[5] Durr, P., T. Staijen, L. Bergmans and M. Aksit, *Reasoning about semantic conflicts between aspects*, in: *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[6] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach*, in: Rozenberg [17] pp. 247–312.

[7] Hannemann, J., R. Chitchyan and A. Rashid, editors, "Analysis of Aspect-Oriented Software (ECOOP 2003)," 2003.
URL http://www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/AAOS_Home.php

[8] Hausmann, J. H., “Dynamic Meta Modelling: A Semantics Description Technique for Visual Modeling Languages,” Ph.D. thesis, University of Paderborn (2006).

[9] Janssens, D., *Actor grammars and local actions*, , **III: Parallelism, Concurrency and Distribution**, World Scientific, Signapore, 1999 pp. 57–106.

[10] Kastenberg, H., A. Kleppe and A. Rensink, *Defining object-oriented execution semantics using graph transformations*, in: R. Gorrieri and H. Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science **4037** (2006), pp. 186–201.

[11] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, *Getting started with AspectJ*, Commun. ACM **44** (2001), pp. 59–65.

[12] Leavens, G. T. and C. Clifton, editors, “FOAL: Foundations of Aspect-Oriented Languages,” 2003.
URL http://www.cs.iastate.edu/~leavens/FOAL/cfp-2003.shtml

[13] Nagy, I., L. Bergmans and M. Aksit, *Composing aspects at shared join points*, in: A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays (NODe)*, Lecture Notes in Informatics **P-69**, Gesellschaft für Informatik (GI), Erfurt, Germany, 2005.

[14] Pawlak, R., L. Duchien and L. Seinturier, *CompAr: Ensuring safe around advice composition*, in: M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science **3535**, 2005, pp. 163–178.

[15] Plump, D., *Term graph rewriting*, , **II: Applications, Languages and Tools**, World Scientific, Singapore, 1999 .

[16] Rensink, A., *The GROOVE simulator: A tool for state space generation*, in: J. Pfalz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science **3062** (2004), pp. 479–485.

[17] Rozenberg, G., editor, **I: Foundations**, World Scientific, Singapore, 1997.

[18] Smelik, R., A. Rensink and H. Kastenberg, *Specification and construction of control flow semantics*, in: *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006.