A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points

Mehmet Aksit
Software Engineering Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
aksit@ewi.utwente.nl

Arend Rensink
Formal Methods and Tools Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
rensink@cs.utwente.nl

Tom Staijen
Software Engineering Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
staijen@cs.utwente.nl

ABSTRACT

Aspects that in isolation behave correctly, may interact when being combined. When interaction changes an aspect's behaviour or disables an aspect, we call this interference. One particular type of interference occurs when aspects are applied to shared join points, since then the ordering of the aspects can also influence the behaviour of the composition. We present an approach to detect aspect interference at shared join points. Aspect compositions are modelled by using a graph production system for modelling aspect-language semantics. A graph-based model of a join point is generated from the source-code of the system. This graph is transformed into a runtime-state representation. Combined with the production system (and the correct tooling) the execution of the aspects is simulated. This simulation results in a labelled transition system that can be used to analyse and verify different properties of the system at the join point.
Simulation of the entire system can be computationally expensive. In our approach, we decide to abstract base system execution into non-deterministic valuation and carefully choosing advice semantics, such that simulation of the entire system can be avoided.

Categories and Subject Descriptors

D2.4 [**Software Engineering**]: Software/Program Verification—formal methods, model checking, validation.

General Terms

Verification.

1. INTRODUCTION

Aspect-oriented programming languages allow the modular specification of crosscutting concerns. Behaviour specified separately in aspects is added into a base system[1] – a system without aspects – during compile-time or run-time. This process is called weaving. When multiple aspects are applied to a system, unexpected results can emerge: two or more aspects behaving correctly when applied in isolation, may interact in an undesired matter when applied together. This phenomenon is called *aspect interference.*

Aspects are typically constructed by pointcuts and advice. A pointcut selects a set of points in the execution of a program, so-called *join points.* Advice consists of the units of execution that are inserted at these join points. Interference between aspects occurs when one aspect disables or changes the behaviour or applicability (i.e. the composition with the base system) of another aspect. There are different causes for aspect inference:

- At weave-time, the set of join points (pointcut matches) of one aspect can be changed by another aspect;
- At weave-time, aspects that change the static structure of a program (introductions) can cause ambiguous weaving - resulting in different programs - depending on the weaving order [10].
- At run-time, one aspect can modify fields or variables, affecting the behaviour of another aspect;
- At run-time, one aspect can change the control-flow of the system, causing a join point of another aspect to never be reached.

Join points that are selected by more than one pointcut are called *shared* join points. When no fixed order of advice execution is determined by the program directives, but the order of advice execution affects the result, a shared join point can lead to unpredictable and undesired behaviour of the woven system, or even an ambiguous system. Other studies such as [18] have already indicated that special attention must be paid to shared join points.

In this paper we propose a detection mechanism for aspect interference at shared join points. The run-time semantics of an aspect-oriented programming language will be specified as a graph-transformation production rule system. The formal specification allows simulation of the execution of advice, resulting in an execution state space. We will show that this can be used to detect aspect interference.

[1]There are also aspect-oriented languages which do not make distinction between aspect and base system; these languages are not addressed in this paper.

Simulation of a complete system is computationally expensive; it is desirable to only analyse the aspects. In our approach, only the execution of the composed aspects is simulated, not the base system. The source-code of the base system is only used for extraction of aspect compositions - those combinations of aspects that occur on shared join points - and to construct a simplified representation of the base state that allows for advice execution.

We find the following characteristics of an interference analysis approach to be desirable:

- It should be possible to detect aspect interference on shared join points.
- The approach should be generic with respect to AOP languages: it should be applicable to different aspect-oriented languages.
- The approach should be modular: although analysis of aspects can be limited to the occurrence of shared join points in a program, the source code of the base system should not be required for the actual analysis.
- The approach should be *practical*: it should let a developer analyse a program without much effort.

The approach is explained using Composition Filters (CF) [2] as example aspect-oriented language. We will argue that CF is typically but not solely suitable for our approach.

The rest of the paper is organised as follows. In Section 2 the problem of aspect interference at shared join points is discussed in more detail and an example is introduced. In Section 3 we will explain our approach in detail. We evaluate the approach in Section 4 and in Section 5 we elaborate on some possible improvements. We finalise with related work in Section 6 and our conclusions in Section 7.

2. PROBLEM DEFINITION

In this section we will elaborate on the problem of interference among aspects at shared join points.

2.1 Aspect Interference at Shared Join Points

Interference between aspects at shared join points will only occur when aspects depend on the same system state during the execution of the advice. Three different kinds of interaction between aspects can cause interference to occur at shared join points.

1. The first problem is caused by a common feature of aspect-oriented languages; they allow pointcuts to consist of predicates over both the static structure of the program and the run-time state. These conditions allow the aspect to be applied only at a certain run-time state. Where the static predicates are typically resolved during weaving - resulting in a set of so-called *shadow* join points - the dynamic predicates (also referred to as pointcut residues) are typically woven in with the advice like an *if*-statement. The value this pointcut residue depends on can be changed by a preceding aspect. This weaving-interference, already described in the introduction, is only recognised during the execution of the system.
2. An aspect can change or abort the control-flow of the system, for example by aborting the join point action and all succeeding aspects to be executed at that join point. A typical *AspectJ* example of this behaviour is an *around* advice without a *proceed()* statement.
3. Aspect interference occurs if one aspect writes to variables or fields, such that the behaviour of a succeding advice is affected. As this is a general problem, it can also occur on shared join points.

We will illustrate the problem by examples of aspect interference. Consider a system where *String* objects are sent between objects and assume we add the following four aspects to this system:

Logging.

The logging aspect logs method-calls to a file.

Authorisation.

The authorisation aspect will disallow unprivileged users from using certain methods in the system. These methods will be blocked by aborting the method dispatch process.

Profanity Filtering.

The profanity filter aspect removes inappropriate words that are sent.

Encryption.

The encryption aspect encrypts strings for secure transportation in the system.

The combination of Logging and Authorization illustrates an example of interference type 2 presented above: interference caused by a control-flow modification. When the logging advice is executed before the authorisation advice, a method can be logged without being executed. In reversed order, the method may be aborted before the method-call is logged.

The Profanity Filter and Encryption aspects combined present an example of interference type 3: both aspects change the same field. Encrypting the string after filtering inappropriate words is the obviously desired behaviour. In reverse order, the profanity filter will be applied to an encrypted string and will not be able find any profane language.

Logging and Encryption is also an example of type 3, where one advice writes to a field and another advice reads this field. When Logging is executed first, the original string is logged; in reverse order the encrypted string is logged.

We will now explain the Composition Filters model, and then introduce the implementation of the introduced examples in Composition Filters. We will explain that in Composition Filters, interference type 1 is not possible.

2.2 The Composition Filters Model

The Composition Filters model is an extension of the conventional object-based model, where objects are enhanced with *filters* for the manipulation of incoming and outgoing messages. Figure 1 illustrates this mechanism; an object is enhanced into a CF object by adding filters. All incoming and outgoing messages have to pass the corresponding set of filters. Typically, in object-oriented languages, messages are method-calls.

Filters are grouped into components called *filter modules*.

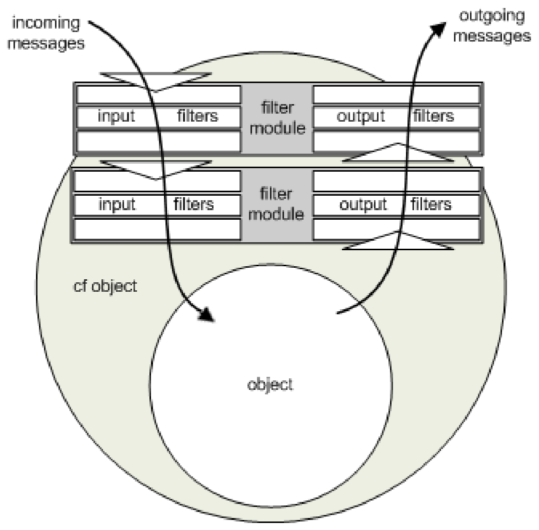

Figure 1: Simplified representation of concern instance with filters

These units of reuse provide execution context for the filters: persistent local and global variables called internals and externals, respectively. Filter modules can be declared separately and superimposed on a selection of classes, allowing the declaration of aspects. We consider this to be the static pointcut matching.

In essence, a filter declaration consists of a filter-type and a matching pattern. The filter-type defines the behaviour of the filter, by an accept action and a reject action. The matching pattern consists of a condition expression and a matching part. The condition expression refers to a condition (boolean) method in the enhanced object or in the filter's context, given by the filter module. The values of conditions can be changed by aspects, but changed values are not used during the evaluation of the filter set. Therefore, interference type 1 of the previous subsection, can not occur. The matching part accepts messages typically by their *selector*, which is the name of the called method.

Composition Filters concepts can be mapped to those of regular AOP-languages. Superimposition specifications are pointcut designators. Selecting classes in this superimposition specification corresponds to static pointcut matching. *Before, after and around* is replaced by an implicit method interface and/or abstract interfaces of objects (i.e. whenever a message enters of leaves an object). Filter module specifications correspond to advice declarations. Conditions in filter declarations are like *if*-pointcuts, testing a dynamic property. Notice that conditions – in this mapping – are part of the advice.

Composition Filters is among the AOP languages where aspects are added to the objects modularly (also called blackbox AOP approach). Languages that introduce aspects as proxies/interceptors can also be put into this category. Such approaches may have the following advantages:

1. The AOP extension *can* be designed in a base-language-independent way;

2. Advice respects the encapsulation of modules; aspects only affect incoming and outgoing calls;

3. Interface-based composition, due to encapsulation and well-defined method interfaces.

Composition Filters is an attractive language for our approach: not only can aspects be analysed in a base-language-independent way, the advice-language itself consists of a fixed set of filter-types; only the semantics of this fixed set of filter-types has to be specified.

2.3 Example Code

We now give an implementation of the aspects described earlier in this section in the Composition Filters language. Listing 1 shows the complete specification of the `Logging` aspect. The aspect consists of a filter module named `LogModule`, which contains one input-filter. This filter is evaluated when a message is received by an object enhanced with this filter module. The input-filter declaration contains the name of the `log` filter, the filter type `Log`, and a matching pattern, which matches the selector `send`. A substitution part (`*.*`) will pass the matched target and selector to the action performed by the filter. The superimposition selects class `Server`, using a Prolog query on the static structure of the base program, and superimposes the `LogModule` filter module on this class. Thus, whenever a method named `send` is called on an instance of class `Server`, the message will be logged.

```
concern Logging {

  filtermodule LogModule {
    inputfilters:
      log: Log = { [*.send] *.* }
  }

  superimposition {
    filtermodules
      classes = { x | ClassByName{ x, 'Server' } };
    superimposition
      classes <- LogModule;
  }
}
```

Listing 1: Composition Filters source code of the Tracing aspect

Listing 2 shows the filter module specification of the other three aspects: Authorisation, Profanity Filtering and Encryption. The superimposition specifications have been left out; except for the name of the filter modules they are identical to the specification of the Logging aspect. The Authorization aspect is implemented by the AuthModule filter module. It uses the Abort filter-type, which will abort evaluation of any other filters and return without executing the requested method. The filter specification matches any incoming message with selector `send`, but only when condition `isAllowed` is false. The value of the condition is requested from an external of type `User`. The Profanity Filtering and EncryptModule are identical to the Logging aspect, except for the filter-type that is used. The FilterProfanity type will change the string argument of the message to not contain any profane language; the EncryptString type will encrypt the string argument of the message.

At run-time, the used filter types *Log*, *Abort*, *FilterProfanity* and *EncryptString* are associated with accept-actions *LogAction*, *AbortAction*, *FilterProfanityAction* and *EncryptAction*, respectively, which are performed when the filter accepts a message. All used filter types have the *Continue-Action* as a reject-action. This action is executed when the filter rejects a message, and will continue the message to the next filter. If the filter is last in line, the message will be dispatched.

```
filtermodule AuthModule {
  externals
    user: User;
  condition
    isAllowed = user.isAllowed();
  inputfilters:
    auth: Abort = { !isAllowed => [*.send] *.* }
}

 filtermodule ProfanityModule {
    inputfilters
      profanity: FilterProfanity = { [*.send] *.* }
 }

 filtermodule EncryptModule {
   inputfilters
     encrypt: EncryptString = { [*.send] *.* }

 }
```

Listing 2: Composition Filters source code of the Authorisation aspect

For pre-defined filter types, the user will not see the implementation of these actions. Rather, a well-defined semantics is given, although in natural language. User-defined filter types are implemented by the user, but are intended to have a reusable nature.

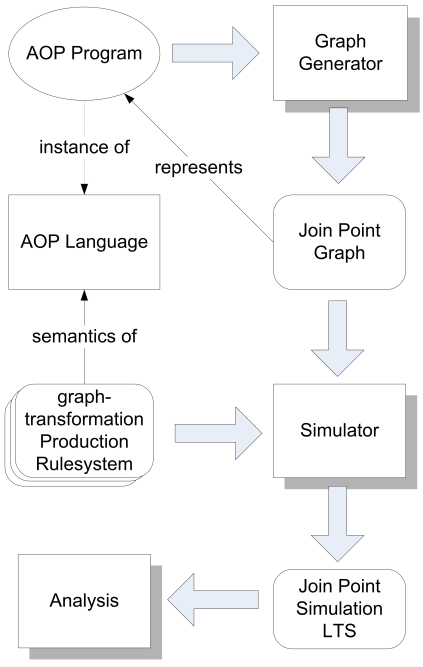

Figure 2: Global Overview of the Approach

3. APPROACH

We propose to compare different advice orderings to detect aspect interference. We will analyse these orderings by means of simulation of shared join points using graph transformations. Interference is detected by comparing the simulation results of the different orderings.

3.1 Process of the Approach

The approach is visualised in Figure 2. From a program specified in a specific AOP language, a graph is generated for each shared join point.

The semantics of the aspect-oriented language is specified using graph production rules. Essentially, this means that each rule specifies the semantics of a single language element, a small-step semantics. A join point graph essentially represents a snapshot of the initial state of the system, but only the part we are interested in.

As mentioned earlier, we do not wish to simulate the entire program. Firstly, because creating a state space of the entire program is computationally expensive. Secondly, simulation of the entire program would require us to model the semantics of the base language with graph production rules.

A join point graph and the run-time semantics production rule system are given to a simulator tool which will generate a state space (represented as a labelled transition system, or LTS) of the execution of the different advices at the specified join point in all possible orders. In this LTS, each state is represented by a graph; the transitions represent rule-applications. Again, the graphs only describe the essential part of the state.

The generated LTS is then used for analysis of the join point. We detect interference by comparing the result of different orders: the advices have to be commutative.

We will explain the concept of graph-transformation first. Then we will explain the structure of the generated graphs, followed by a description of the graph production rules used to simulate the execution of the advice. Then, we will describe the simulation of the shared join points and the analysis method for interference detection.

3.2 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 of an LTS contain the static structure of that part of the program that is involved in the join point simulation, additional control flow information, and a part that represents run-time 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. Given a start-graph and a set of production rules, a state space can be generated by iteratively applying all rules to all graphs, wherever this is possible. Hence, the states in this state space are graphs; the transitions are rule-applications.

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 [23]. In fact, we have used GROOVE [21] 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 [7]. Since the point of this paper is to illustrate an application of graph transformations, we omit the details of the formalism.

Figure 3 shows a rule specification in the GROOVE notation, containing nodes and edges, for simply explaining the different lines styles used in GROOVE. The labels in the nodes are in fact self-edges connected to these nodes. The different line styles all have different meanings: normal

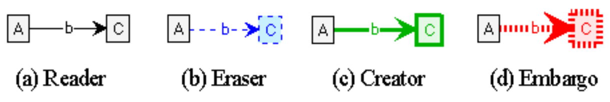

Figure 3: Example rule with different line styles.

nodes and edges (a) are so called *reader* elements, that will be used for matching; dashed elements (b) are *eraser* elements, which will be removed and thus also are required for matching the rule; thick lines (c) represent *creator* elements, which will be added to the graph when the rule is applied[2]. Thick dashed lines (d) represent *embargoes*, or so-called *negative application conditions* (NAC), which — when matched — prevent the rule from being applicable.

3.3 Graphs Representing Shared Join Points

Graphs represent run-time states of shared join points. A join point in CF is a method call to a class that has one or more filter modules — two or more for shared join points. The contents of the graph — representing a single shared join point — can be divided into three parts.

First, the graphs contain the essential part of the static structure of the program to be able to simulate the advices. This is in fact the static structure of the filter modules superimposed on the class of the target of the method call. This is essentially a graph-based representation of the *abstract syntax tree* (AST) of the filter modules, a single node for the class, and a filterset node connecting the class and the filter modules.

Second, control flow information is added to the graph. Specification of the language semantics involves giving both a control flow semantics and a run-time semantics to the Composition Filters language. The control flow semantics of Composition Filters is specified in a dedicated language for this purpose, developed in [25]. This includes an automatic translation to a graph production system. Applying this transition system to the graph will add control flow information to the graph-representation of the filter modules. This means that nodes are enriched with a `flow` edge to the next control flow element, or with a nodes and edges representing the "true" and "false" flow for conditionals. These edges can be used in the run-time semantics to be able to move the program counter to the next flow element.

Last, a run-time state representation of a shared join point is added to the graph. In the composition filters model, this requires a message with a sender, target, selector and arguments. This message — which is in fact a method call to the class with the filter modules of which the static structure is in the graph — is represented by a graph representation of a number of stack frames such as described by the type-graph in Figure 4.

Frames represent the execution context of a specific part of the program. Frames typically have an outgoing `pc` edge — the program counter — to a `FlowElement` (an element of the AST). The `parent` edge connects to the parent frame, the execution context of the method-call. A outgoing program counter indicates the frame is active. Typically, only one `pc` edge exists in the graph when dealing with a single-threaded program. When a frame is finished and deleted, execution continues in the parent frame, by restoring the program counter.

In our model we distinguish between two different kinds of frames: a `MethodFrame` is used for executing a method body; a `FilterFrame` is used for executing a filter set[3].

A method frame has a `target` edge to an `Object` of a certain `type`; this is the target of the message. When the method is dispatched (after executing the filter-modules), this edge is replaces by a `self` edge.

The method frame is connected to a `Signature` with a `name` of type `string:`, and any number of `Arguments`. The frame is connected to `ArgValue` nodes via `arg` edges. These `ArgValue` nodes are connected to `Object` nodes that represent the arguments of the method call.

A method frame can be in three different states before it starts executing a method body (i.e. before it has a `pc` edge): `filtering`, `dispatch`, or `abort`. Filtering occurs before the method is dispatched. When the filter process is finished, the state of the frame is updated to dispatch, causing the dispatch process to start. Filters that abort the message flow will change the state to abort. The actual run-time implementation of such an abort depends on the platform (e.g. an Exception), but since we do not simulate dispatching and returning, we are merely interesting in the fact *that* the frame was aborted, not *how*. Therefore, the `abort` edge will be sufficient. When no filters are present, the frame immediately reaches the dispatch state.

A filter frame is connected to a method frame in the filtering state via a `filters` edge. This method frame represents the message in the CF model. The filter frame is connected via `pending` edges to the filter modules of the that are in the filter set of the message target's class. When a filter module is selected for execution the pending edge is removed. When no more pending edges exist, the filtering process is finished.

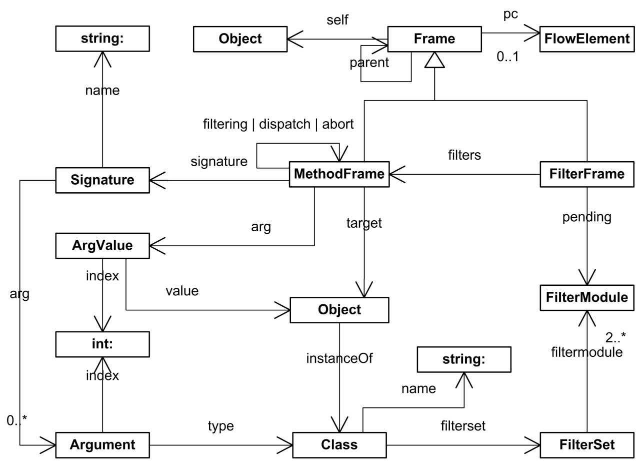

Figure 4: Type-graph of a Shared Join Point

Figure 5 shows the part of a generated graph representing the run-time state of a join point. It reflects a method call, before it is dispatched, to an object of a class on which two

[2]For the more knowledgeable in the field of graph-transformations: the normal and dashed elements represent the left-hand-side (LHS); the normal and thick elements represent the right-hand-side

[3]The labels `MethodFrame` and `FilterFrame` do not appear in the actual graphs; they are added just to distinguish them in this explanation

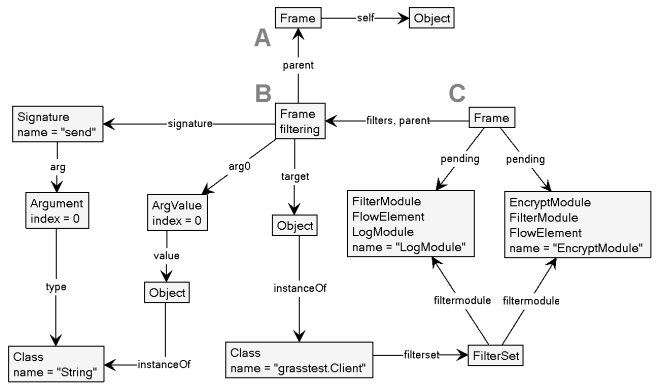

Figure 5: Example run-time part of a join point graph

filter modules are superimposed. The graph contains three frames, that have only the essential parts of the type-graph needed for the simulation, i.e. to reflect the message:

- The *caller* frame (annotated with A in the figure), which is associated with an `Object` via a `self` edge. This is the frame the method-call originated from. The `self` object of the frame corresponds to the *sender* of the message.

- The *target* frame (B), not being dispatched yet, represents the frame that will execute the called method. The frame, which is in a filtering state - is connected to the caller frame via a `parent` edge. The `target` object corresponds to the target of the message. The `name` of the signature corresponds to the *selector* of the message. The arguments of the frame correspond to the *arguments* of the message.

- The *filter* frame (C) represents the frame responsible for the execution of the filters. Thus, when filtering is done, the initial method-call operation can be continued. The filter frame has a `pending` edge to each of the `FilterModule` nodes that are in the filter set of the target's class. It has a `filters` edge to the target frame; this is used for linking to the sender, target, selector, and arguments. The `parent` edge represents the normal frame hierarchy.

3.4 Graph Transformation based Language Semantics

Run-time semantics of imperative languages can be modelled by a rule-based specification language such as graph transformations. In particular with GROOVE, work has been done on specifying the run-time semantics of the .NET IL language [26], a subset of Java [1], and TAAL [13] (a small but realistic "toy" object-oriented language without the more advanced features such as exceptions and threads). In this subsection we will show the specification of the run-time semantics for Composition Filters as a graph-transformation production system.

The run-time semantics has been defined by a single rule for each type of syntax element of the language. These rules are specified by hand but, once fixed, allow the method to be applied to any CF specification.

In fact, these rules can be divided into two categories. One set of rules is used to specify the semantics of the filtering language. When simulated, these rules evaluate whether a filter should execute either its accept or reject action.

The other category consists of a rule for each type of filter action, and describes the actual behaviour of the filter actions: the effect of the action on the state of the system. In this paper, we describe the rules that represent the actions performed by the filter types used in the running example.

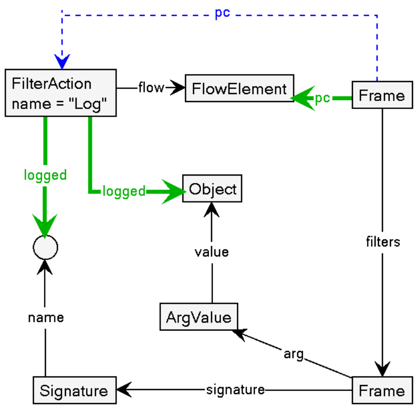

Figure 6: Rule specifying the Log action

The rule specifying the *Log* action is displayed in Figure 6. As a quick reminder: the dashed and normal line styles represent the part that is used for matching the rule. The rule matches a `Frame` node (which in fact represents a filter frame) with a `pc` edge to a `FilterAction` node, where the name of the action is `"Log"`. The Frame has a `filters` edge to the method frame representing the message. This second frame has a signature with a `name` edge to round shaped node: an attribute node. Attribute nodes have an identity related to their value, e.g. two equal string values always use the same string attribute node. We assume that the filter type "Log" should be used for messages with one argument only. The rule matches this argument via the `arg` edge to the `ArgValue` node and the attached value.

The specification mechanism allows for a certain level of abstraction. For example, a rule can specify where the message is logged (e.g. to console or a file), what exactly is logged, and in what order messages are logged. We have chosen to specify only what is logged. When applying the rule, a `logged` edge to the selector and the argument is created. Also, the `pc` is removed and a new `pc` edge is created to the next flow element, the target of the `flow` edge.

This rule illustrates that we can choose a level of abstraction in specifying the behaviour of the filter actions. We merely need an abstract representation of the effect of advice application on the state, such that different states can be identified. In the case of the logging filter, we only care about *what* is logged, and therefore only that information determines the resulting state.

Figure 7 shows the production rule of the *Abort* action. The dashed `Frame` with a `pc` edge to a *FilterAction* with name `"Abort"` indicates that it is applicable when the execution has reached the abort action. The dashed filter frame is removed, ending the filter evaluation; the target frame is tagged with an `abort` edge to prevent it from being dispatched.

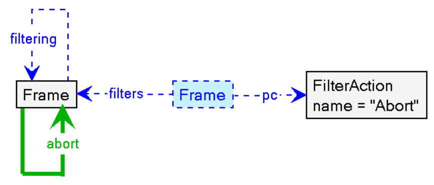

Figure 7: Rule specifying the Abort action

Figures 8 and 9 show a specification of the ProfanityAction and EncryptAction, respectively. Both "increase" the program counter along the `flow` edge. A newly created `Object` node replaces the original argument of the message. Connecting the new argument with the old argument with an advice-specific edge. This allows us to distinguish with which argument the actions are performed.

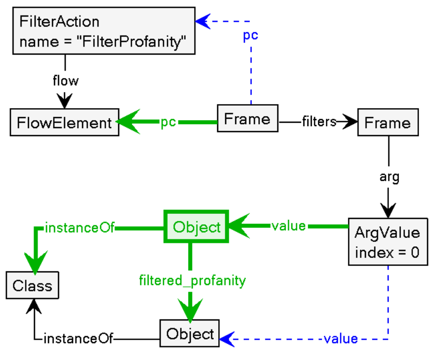

Figure 8: Rule specifying the FilterProfanity action

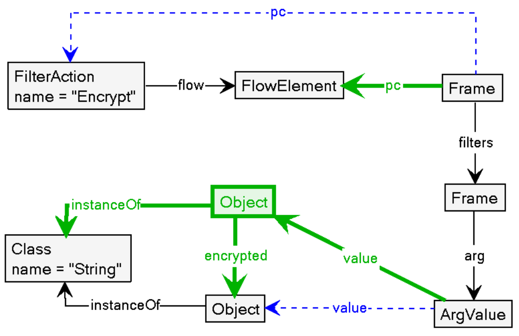

Figure 9: Rule specifying the Encrypt action

3.5 Simulation & Analysis

For simulation of the advice at the join point, there are two issues to be dealt with.

1. Part of the filtering process deals with accepting or rejecting a message based on the value of a condition. Since we do not execute the entire system, we do not know (and can not evaluate) the actual values of conditions at the join point. We deal with conditions by valuating them non-deterministically; the transition system branches into a path where the condition is true and a path where the condition is false.

2. To simulate all possible orders of the advices, two rules are used to non-deterministically choose the next advice to be simulated (one rule for the first filter module, and one rule for any succeeding filter module). As explained, the filter frame was connected to the filter modules with `pending` edges. Both rules move the `pc` edge to *any* filter module connected by a pending edge and remove this pending edge. Thus, a branch is created for every possible choice of a filter module. Once execution of the filter module is finished, the same is done for the remaining filter modules, until no more pending edges remain. The worst-case number of different orders (and paths) is $a!$, where a represents the number of filter modules at the shared join point.

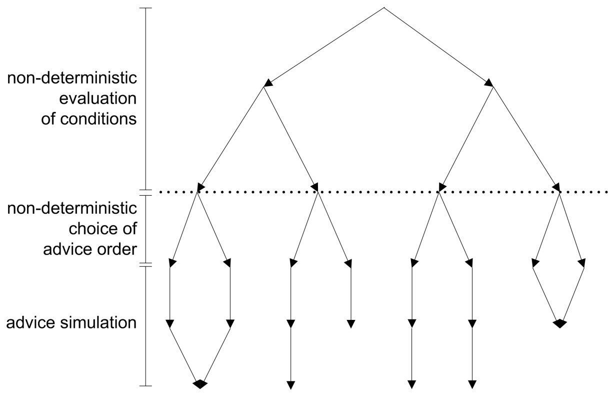

Figure 10: General Shape of the LTS, with two conditions and two advices

A generalised shape of the generated LTS for the specified rule system is shown in figure 10. In AOP terminology, the initial state represents a shadow join point, since it reflects a statically matched join point. Branching occurs when a rule has multiple matchings in one graph or when different rules match in the same graph. The branching above the dotted horizontal line represents assigning values to the conditions that do not have a value. The states on the dotted line represent all actual join points for the shadow join point. The number of paths to these states is 2^c, where c represents the number of conditions.

Below the dotted line, branching occurs when the first filter module is selected, after which all filter modules are executed. The figure illustrates that for different values of condition, different shapes in the LTS can occur.

When all advice orders on a shared join point have been simulated, we can analyse the composition of the advices by looking at the shape of the LTS. Our goal is to use this to conclude whether or not the advices interfere.

When advice actions are commutative — the order of execution does not affect the resulting state — the execution traces of the different orders are confluent, because the same state is automatically represented by the same "box" in the LTS. The equivalence of states is based on an isomorphism check that is based on labels, not on node identities. Confluence is visualised in the left-most and right-most diamond shape in Figure 10.

Although the order of execution of a number of advices does not affect the result, we can not immediately conclude that this is also the desired result. Imagine a logging advice that logs an immutable copy of the (original) argument of a method, and another advice that modifies the value of a mutable variable containing the argument. Both orders of advice execution would log the original argument and make the same change to the argument. In other words, the LTS would be confluent. However, if the behaviour of the logging advice is documented as “logging the value of the argument passed to the method body”, the combined behaviour is not correct. The expectation is only satisfied when the argument is not changed.

Although we cannot be certain that a result from confluent orderings is the “expected” result, at least we are certain that the all orders of the advices yield the same result. In problem in the scenario we just illustrated, is caused by an assumption that holds when the aspect is applied in isolation, but may not hold when composed with other aspects.

When advice units are not commutative, the order of execution affects the resulting state and the execution of different advice orders will result in different final states. This is illustrated in the two middle cases in Figure10. One of these states might be the desired result, or both might be undesirable. In any case, we can conclude that the advices interfere: the changes made to the state by one advice affects the applicability of the other advice or the state change made by the other advice.

This definition of interference helps us understand when confluence can also occur even though the advices interfere. The state change made by the first advice and the effect of the first advice on the state change made by the other advice might “accidentally” add up to an identical resulting state. However, by intuition we believe this to occur only very occasionally. In future work we like to investigate this further.

Figure 11 shows the generated state space for a shared join point with the Logging and Encryption aspects. The two paths are not confluent, since the Log action tags a different string; the advices interfere. Figure 12 shows the generated state space for a shared join point with the Authorisation and ProfanityFilter aspects. In the first branching, different values are assigned to the condition expression of the authorisation advice, resulting in two actual join points. Then, on the left side (where the condition is true), the branches do not merge because the authorisation advice aborts the flow either before or after the profanity filter has executed, resulting in different final states. On the right side (where the condition is false), the condition of the authoriation advice fails, such that it does not abort, in which case the advice do not interfere.

Figure 13 shows the generated state space for all our four aspects, which is a bit harder to analyse visually. The first branching again represents the evaluation of the condition expression in the Authorisation aspect. We can see that more than one final state can be reached via the 24 ($a!$, where $a = 4$) different execution orders for each of the two actual join points. The figure can be used to analyse the possible executions of the join point.

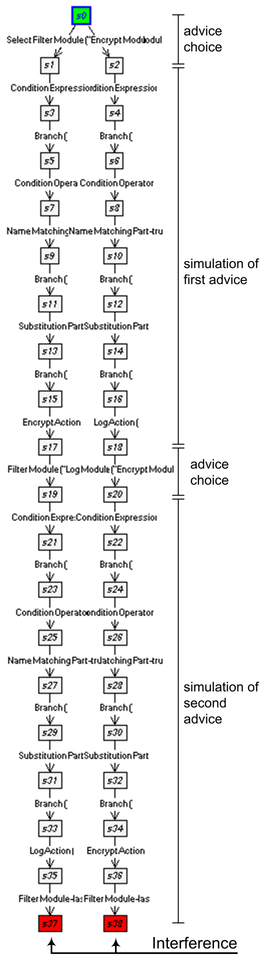

Figure 11: Generated transition system for Logging and Encryption

4. EVALUATION OF THE APPROACH

In this section various aspects of the approach are evaluated.

4.1 Detection of Interference

The approach allows to abstractly specify the behaviour of advice actions, such that only relevant behaviour is incorporated. Of course this also means that the specification can be over-abstracted causing certain problems to be undetectable.

Although we cannot guarantee that a composition of aspects is free of interference, we can warn the user (with certainty) for interference in case of a non-confluent result. When the result is confluent the advices are either free of interference or coincidentally causing the same wrong result with all orders of execution (which we consider very rare). We believe that when advices are commutative for every combination of condition values, the shared join point is highly likely free of interference.

The visual nature of the result — the LTS — can help in understanding the composition of advice, even as simply as seeing different shapes under different condition values. This can help in understanding if a result is desirable or help debugging a problem.

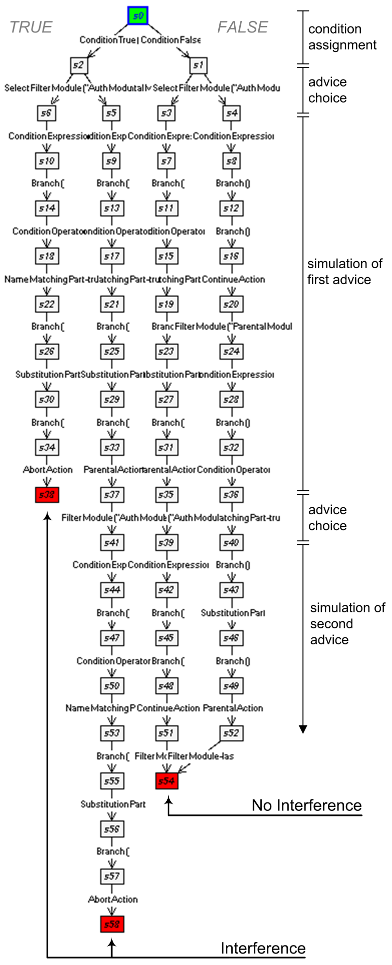

Figure 12: Generated transition system for Authorisation and ProfanityFilter

The approach allows to distinguish results for different run-time states, by tailoring the condition assignments. Some false positives (detection of interference that will never actually happen) can occur when certain combinations of condition values (i.e. run-time states) are never found when the program is executed. Again, the visual nature of the LTS can help in analysing the different scenarios.

4.2 Modularity

Modularity in the context of verification of aspect-oriented software development typically means that it is possible to analyse and verify aspects and aspect compositions independently of a base system. Our approach is modular in such a way that the source-code of the base system is merely used to extract shared join points. We have shown that our approach can detect aspect interference by simulation of advices alone. When the base system's source code is not available, it is possible to simulate every combination of two advices. This can, however, lead to false negatives and false positives for actual base programs, when the advices are never composed at shared join points.

4.3 Practical in Use

Since the base system is not simulated, our approach requires only the advice language to be specified formally using graph-transformation rules. For the Composition Filters language this is a reasonable task, although we have not included *Meta* filters. Such filters call a method specified in the base-language. To incorporate analysis of meta-filters, we would not only have to specify base language semantics, but also have to incorporate a larger subset of the base-system's run-time state. Determining this subset might be a difficult task.

CF comes with a base set of filter-types. These filter-types are argued to be useful for a large number of advice specifications, and therefore elements of reuse. The specification of the language performed for this works may therefore already allow analysis of a large number of programs specified using CF.

However, custom filter types can be added to the Composition Filters language. In this case, either these custom types can be neglected during analysis, or the developer could specify the behaviour as a graph production rule to be able to include the custom filter type in the analysis.

In fact, the example presented in this paper is based on the use of custom filter types (*Log*, *FilterProfanity*, and *EncryptString*). However, this was done just to create a simple and small yet interesting example.

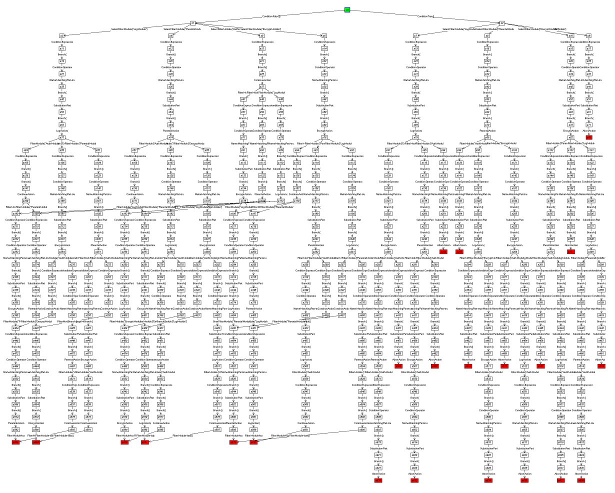

Figure 13: Generated transition system of the example program with four aspects

4.4 Scalability

The complexity of our chosen simulation strategy is a function of the number of different conditions used (c) in the filter modules, and the number of filter modules (a) on the join point. The complexity of the simulation is of order $2^c \times a!$ per analysed join point, which expresses the number of paths. The number of conditions will most likely remain small compared to the number of advice, since conditions in pointcuts are commonly known to be a run-time performance bottle-neck.

Simulation of a single filter takes approximately ten rule applications. In a (bad-case) scenario where every filter module contains one filter and one condition expression, the size of the state space approaches $10 \times 2^a \times a!$. GROOVE is able to generate state spaces with size up to an order of 10^6 in a timescale of seconds. This allows us to simulate up to around six advices in the assumed scenario (one condition per filter module). Although the occurrence of shared join points is not rare, finding shared join points of six or more advices will not be a common case.

4.5 Tool support

The graph generator has been implemented as a *Compose** compiler module, which is a compile-time and run-time implementation of the Composition Filters language. Compose* is available for both the Java and .NET platform. Automatic graph generating involves generating graphs for all shadow join points, adding control flow information (by simulating the graph-transforation rules of the control flow semantics) and adding stack frames representing messages based on the selectors that are used in the filter specifications.

After graphs have been generated, run-time simulation is started. The resulting LTS can either be written to a file or opened in a GROOVE viewer. Analysis of the state space to give understandable feedback to the user has not yet been automated and currently can only be done by visually analysing the state space. Given the generalised structure of the state spaces, however, implementing this is straightforward.

The tool has been integrated into the Common Aspect Proof Environment (CAPE) [6], a framework for aspect verification and analysis tools and modules over various aspect languages.

5. DISCUSSION

In this section we discuss some possible improvements of the approach.

5.1 Applicability to other Aspect Languages

Composition Filters has proved itself to be a suitable language for our approach, due to the nature of the advice language. Two issues have to be dealt with, when trying to apply the approach to other languages.

First, many other aspect-oriented languages have an advice language that is a variant of the base language. Specifying such an advice language is possible, but a much larger task then specifying the small CF language. The entire base-language needs to be specified using graph transformation rules. However, applying the approach to a simplified base-language with *proceed* may already be interesting. As a matter of fact, a case study has been performed, where the approach was successfully applied to the Common Aspect Semantic Base (CASB) [5], a formal model of Featherweight Java with assignments and Featherweight AspectJ. The work resulted in a graph production system for simulation of entire programs written in the language, not just join points.

Second, languages like AspectJ employ more comprehensive join point models that allow for interception of assignments, constructor calls, static initialization, throwing exceptions, etc. In our approach, the interception mechanism is modelled as a change of the method dispatch mechanism, which is based on method frames. This is feasible because, in the Composition Filters model, the only kind of join points are messages (or method calls) between objects. For more fine-grained join point models, a different graph representation is needed, as not only a run-time process is changed, but in essence any instruction can be intercepted. Automatic join point graph generation becomes a lot more complicated. However, proxy/interceptor based languages such as Spring AOP [12] are becoming more and more popular. These languages typically only intercept messages between objects. Therefore, it is likely that join points can be represented by graphs as presented in this paper — using frames for those messages — since the join point models of these languages are similar to that of CF.

Currently, we are implementing a graph transformation-based semantics of AspectJ. The work includes specification of base language semantics. For applying the analysis presented in this paper, we foresee that it does not require a full program simulation; we can integrate existing work for the detection of potential interference problems in AspectJ, and generate join point graphs based on the signature of the involved pointcuts.

5.2 Using Classifications for Optimisation

Classification of aspects helps in understanding the behaviour of aspects. In [24], a classification of aspects was proposed as spectative, regulative and invasive types. In [15] these categories of aspects are extended and specified in a more precise way. Also, similar aspect classifications are in [22] and [3]. The classification presented in [24] can be summarized as follows:

- Spectative aspects produce a side-effect w.r.t. the base system. They do not change the base system.
- Regulatory aspects change or abort the control flow of the system.
- Invasive aspects change variables in the system.

These classifications can be easily mapped to the causes for interference at shared join points that were introduced in Section 2. Interference between aspects can occur with different combinations of aspect types; interaction of a regulatory or invasive advice and any other type can result in interference. Using this knowledge, we could optimise our approach by skipping join points that are shared by only spectative aspects.

5.3 Simulation Strategies

Currently, our approach employs a simulation strategy where unknown fields – needed by an advice – are evaluated before simulating the advice. The advantage of this is that it allows to visually detect interference by looking at confluence of traces that have branched after these fields are evaluated (i.e. shadow join points versus actual join points). The disadvantage is that the advice that uses a condition might not be reached due to a preceding advice aborting the message, such that unnecessary states are created. Traces which involve variables whose values are not going to be read or evaluated (because message abortion occurs before the respective aspect are reached) are essentially equivalent — independent of the values of those variables. An optimization w.r.t. the size of the state-space (and thus the simulation time) would be to lazily assign values to these variables (i.e. when the program counter has reached the actual use of a variable). This however, makes the visual detection of interference hardly feasible and requires detection of interference to be automated to provide understandable analysis results.

In future work, we would like to prove that commutativity is also compositional: commutative aspects $a_1 + a_2, a_2 + a_3, a_1 + a_3$ would ensure that any order of a_1, a_2, a_3 would also be commutative, and thus most likely free of interference. Instead of simulating all possible advice orders on the shared join points that occur in a given program, this allows to verify just every couple of aspects and will greatly reduce the complexity, especially since this verification can be done *once and for all* for a given set of aspects (and any base system). One could verify a set of aspects to be free of interference by merely analysing every combination of two aspects.

6. RELATED WORK

A lot of work has been done in the area that investigates the problems that can occur when aspects are composed. We try to discuss the ones closely related to our approach.

Pawlak et al. [19] 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 join point. The difference with our approach is that the aspects need to be specified in another language in order to be analysed whereas our approach uses the aspect specification directly (once the aspect language has been specified once).

Lagaisse et al. [17] stress the need to express which modules may use and affect each other in the module composition process. Artefacts can be equipped with contracts that specify the provided functionality and dependencies on other components. For aspects, however, the accepted notion of a contract is no longer sufficient. They propose that aspects require to obey the contractual obligations of the components, such as not allowing breaking scope qualifiers (public/private/protected). They call breaking a contract *uncontrolled semantic interference*. So-called aspect integration contracts are introduced, which specify the permitted interference between an aspect and a base component. Essentially, the work focuses on aspects interfering with the composition of the base system, where our approach focusses on interference among aspects.

Rinard et al. [22] propose a classification for aspects interacting with methods. The work also mentions that the same classification can be used between aspects. This classification proposes aspects to have interfering scopes when both aspects write to the same field. The classification system is supported by analysis tools that identify classes of interactions and hence help developers to detect potentially undesired interactions. It is left to the user to decide what is problematic and what is not.

Störzer at al [27] also identify the problem of non-commutative advices at shared join points, the so-called advice precedence problem. A mechanism is proposed to detect relevant undefined advice precedence, by detecting common fields used in read and write operations for advice that share join points.
In our approach, we also recognise the problem of interference based on read and write operations on fields. However, this does not imply that the advice are interfering. It is possible that two advices make an orthogonal change to the same field. Simulation will indicate whether the operations are orthogonal and commutative or not.

Goldman et al. [8] present a modular approach to verify correctness of an aspect relative to a formal specification. The approach is based on model checking using linear temporal logic. We only look at the actual state change of aspects to detect interference, and cannot reason about the intended behaviour.

Kniesel [16] presents a analysis method for weaving interaction and interference. It is based on a logical model of aspects, which specifies conditions and operations on program elements. There is some overlap between their approach and ours, but their approach can not detect run-time data interference between aspects.

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 [20]) to graph-based semantics for actor languages (see Janssens [11]) and visual languages (e.g., [9]). For object-oriented languages the first approach of this kind is by Corradini et al. [4]; the approach of this paper is inspired by [14].

7. CONCLUSIONS

In this paper we present a novel approach to detect aspect interference at shared join points. We employ a graph-transformation-based formalism to specify the semantics of aspect-oriented languages in a precise manner. This involves giving a control-flow semantics and a run-time semantics of the aspect language, and a run-time semantics for advices. By modelling the specification of aspects and a join point as a graph, we can simulate the execution of the aspects, resulting in a state space of the execution of the join point.

Simulating different advice orders allows us to detect aspect interference by analysing whether or not aspects are commutative – whether the order of advice execution at a shared join point affects the result – by analysing confluence of execution paths for different orders in the state space.

The analysis of aspects is done independently of the base system, making the approach more scalable. Besides the specification of the language-semantics, no additional specification is required, making the approach very practical. The approach has been implemented for the Composition Filters language but the approach in general is applicable also to other aspect-oriented languages.

8. REFERENCES

[1] Mark Arends. A simulation of the java virtual machine using graph grammars. In *Master thesis*. 2003.

[2] Lodewijk Bergmans and Mehmet Akşit. Principles and design rationale of composition filters. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 63–95. Addison-Wesley, Boston, 2005.

[3] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Ron Cytron and Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, March 2002.

[4] Andrea Corradini, Fernando L. Dotti, Luciana Foss, and Leila Ribeiro. Translating java into graph transformation systems. In Hartmut Ehrig, Gregor

Engels, Fransesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Second International Conference on Graph Transformation (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 383–389. Springer-Verlag, 2004.

[5] Rémi Douence, Simplice Djoko Djoko, Pascal Fradet, and Didier Le Botlan. Towards a common aspect semantic base (casb). In *Deliverable 54, AOSD-Europe, EU Network of Excellence in AOSD*, August 2006.

[6] Eyal Dror and Shmuel Katz. The revised architecture of the cape. In *Deliverable 42, AOSD-Europe, EU Network of Excellence in AOSD*, August 2006.

[7] Hartmut Ehrig, R. Heckel, Martin Korff, M. Löwe, L. Ribeiro, A. Wagner, and Andrea Corradini. Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [23].

[8] Max Goldman and Shmuel Katz. Maven: Modular aspect verification. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2007.

[9] Jan Hendrik Hausmann. *Dynamic Meta Modelling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2006.

[10] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, Vancouver, Canada*, pages 85–95, New York, NY, USA, 2007. ACM Press.

[11] D. Janssens. Actor grammars and local actions. In Grzegorz Rozenberg, Hartmut Ehrig, et al., editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume III: Parallelism, Concurrency and Distribution, chapter 2, pages 57–106. World Scientific, Signapore, 1999.

[12] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, and Rick Evans. Aspect oriented programming with spring. In *The Spring Framework - Reference Documentation.*

[13] H. Kastenberg, A. G. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. In R. Gorrieri and H. Wehrheim, editors, *Proceedings of the 8th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems, Bologna, Italy*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201, London, June 2006. Springer Verlag.

[14] Harmen Kastenberg, Anneke Kleppe, and Arend 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)*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201. Springer-Verlag, 2006.

[15] Shmuel Katz. Aspect categories and classes of temporal properties. In Awais Rashid and Mehmet Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2006.

[16] Günter Kniesel. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Programming*, LNCS(submitted), 2007. Special issue on Aspect Dependencies and Interactions, edited by Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, Arend Rensink.

[17] Bert Lagaisse, Wouter Joosen, and Bart De Win. Managing semantic interference with aspect integration contracts. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect*, March 2004.

[18] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In Andreas Polze Robert Hirschfeld, Ryszard Kowalczyk and Mathias Weske, editors, *Proceedings of International Conference NetObjectDays (NODe)*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[19] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier. CompAr: Ensuring safe around advice composition. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 3535 of *Lecture Notes in Computer Science*, pages 163–178, 2005.

[20] D. Plump. Term graph rewriting. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume II: Applications, Languages and Tools. World Scientific, Singapore, 1999.

[21] Arend Rensink. 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)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[22] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FOSE)*. ACM, October 2004.

[23] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, Singapore, 1997.

[24] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[25] Ruben Smelik, Arend Rensink, and Harmen Kastenberg. Specification and construction of control flow semantics. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006.

[26] Niek Sombekke. Graph-based semantics of the .net intermediate language. In *Master thesis*. May 2003.

[27] Maximilian Storzer and Florian Forster. Detecting precedence-related advice interference. In *ASE*, pages 317–322. IEEE Computer Society, 2006.