Static and Dynamic Detection of Behavioral Conflicts Between Aspects

Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit

University of Twente, The Netherlands
{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 a *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. This approach 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. This supports automatic conflict detection. The presented approach employs a set of conflict detection rules, which can be used to detect generic, domain specific and application specific conflicts. The approach is implemented in Compose*, which is an implementation of Composition Filters. This application shows that a declarative advice language can be exploited for aiding automated conflict detection.

This paper discusses the need for a runtime extension to the described static approach. It also presents a possible implementation approach of such an extension in Compose*. This allows us to reason efficiently about the behavior of aspects. It also enables us to detect these conflicts with minimal overhead at runtime.

1 An Example Conflict: Security vs. Logging

We first briefly present an example of a behavioral conflict. Assume that there is a base system that uses a Protocol to interact with other systems. Class Protocol has two methods: one for transmitting, sendData(String) and one 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. Imagine a second aspect that traces all the methods and possible arguments. The implementation of aspect Tracing uses a condition to dynamically determine if the current method should be traced, as tracing all the methods is not very efficient. Aspect Tracing 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 that 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.

2 Approach Outlined

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 running example is an example of a conflict that is ordering dependent. And as such can also be resolved by changing the order. However there are conflicts, like synchronization and real-time behavior, which are independent of the chosen order. Order dependent conflicts can be resolved by changing the order either statically or dynamically. This kind of automatic resolution is beyond the scope of this paper. We have implemented such a user-defined option is our Compose* toolset.

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

A resource operation model is defined as follows:

Resources is the set of all resources in the system, e.g. target, sender, selector, arguments;

[1] In this paper we only focus on join point Protocol.sendData, but a similar situation presents itself for join point Protocol.receiveData.

Operations is the set of all possible operations in the system, e.g. read, destructive write and non-destructive write;

Alphabet(resource) is the set of operations which can be carried out on a specific resource, such that $\forall resource \in Resources \bullet Alphabet(resource) \subseteq ValidOperations$;

ResourceOperations is the set of all valid resource operations tuples on a specific resource, such that $ResourceOperations = \{(rsrc, op) \bullet rsrc \in Resources \land op \in Alphabet(rsrc)\}$;

ConflictRules(resource) is the set of conflict rules for resource *resource*;

AssertionRules(resource) is the set of assertion rules for resource *resource.*

2.1 Conflict Model

The previous described conflict detection model has been used to model the behavior of advice. In [2] we provide more detailed information about this generic model and show how this model is derived from two AOP approaches, namely AspectJ and Composition Filters. In this model, we distinguish two types of conflict, control and data related conflicts. Where the first models the effect of advice on the control flow and the latter captures conflicts that occur due to shared data. It is out of the scope of this paper to discuss all the details of this generic model, please consult [2] for this. However, we will present an overview of this generic model.

Data Conflicts. The presented resources are commonly used program elements, which can be inspected or manipulated by advice. These are usually bound via explicit context bindings or via pseudo variables, like *thisJoinPoint* in AspectJ. These resources are: caller, target, selector, arguments, returnvalue and variables.

On these resources we can execute the following operations:

*read***:** queries the state of the resource on which it operates;

$write_n$**:** A nondestructive write will update the state of the resource on which it operates, e.g. compressing (lossless) or encrypting the arguments, in a reversible manner (i.e. without loss of information);

$write_d$**:** A destructive write will override the state of the resource on which it operates, this is normally irreversible;

*unknown***:** Can be either a *read*, $write_n$ and $write_d$, but not known precisely.

Now that the resources and operations are defined we present the conflict rules to detect behavioral conflicts on data elements. In general, these conflict rules can be expressed in any matching language. Here, we use extended regular expressions as defined by IEEE standard 1003.1[3] to specify the conflict patterns.

- Conflict(data): $write_d\ write_d$: The effect of the first destructive write is lost.
- Conflict(data): $read\ write_d$: The first advice may become invalid if the data resource is changed afterwards (at the same join point).

– Conflict(data): *read* $write_n$: The first advice may become invalid if the data resource is changed afterwards (at the same join point).
– Conflict(data): $write_n$ $write_d$: The effect of the first nondestructive write is lost.
– Conflict(data): *unknown*: Using an unknown data manipulation operation can be potentially dangerous.

The presented conflict rules have been defined on the basis of pairs, however matching any of the rules as a sub-pattern is also considered a conflict.

Control Flow Conflicts. To capture control flow related behavioral conflicts we also instantiate the conflict detection model to capture the effects of advice on the control flow. We model control flow behavior as operations on the abstract controlflow resource. On this single resource controlflow, advice can operate using the following operations:

*continue***:** The advice does not change the control flow, it simply passes control to the next advice, if any.
*return***:** The advice returns immediately to either after advice or to the caller, and as such the original join point is no longer executed,
*exit***:** The advice terminates the entire control flow, e.g. a exception is thrown or an exit call is made.

We will now show which combinations of operations on the $controlflow$ resource (may) yield a conflict. Again, we assume here that the (conflicting) operations are derived from two different advices.

– $Conflict(controlflow)$: *return* $.^+$: If one advice returns, another advice which should be executed after this advice, is never executed, hence if there are one or more other operations after a *return*, this will be signaled as a conflict.
– $Conflict(controlflow)$: *exit* $.^+$: Similarly, if one advice terminates the execution, the advice which should be executed after this advice is never executed. hence if an *exit* operation is followed by one or more other operations, this will be signaled as a conflict.

Note that especially these generic rules are typically conservative: i.e. they aim at detecting *potential* conflicts, and will also point out situations that are in reality not conflicting. It is important to see the resulting conflicts as warnings that something might be wrong, rather than absolute errors!

One key observation we have made, is the fact that modeling 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. Our model allows for the extension of both resources and operations to capture also more domain or application specific conflicts.

2.2 Analysis Process

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 1.1. Composed filter sequence example

Filter trace traces all parameters and return value in the beginning and end of a method execution. Filter encrypt subsequently secures the data being send.

To illustrate how we can achieve automated reasoning using the declarative filter language of Composition Filters, we now present an example implementation of a filter which traces all the parameters. See [2] for more detailed information.

$$\overbrace{trace}^{Name} : \overbrace{ParameterTracing}^{Type} = \{\overbrace{ShouldTrace}^{Condition} => [\overbrace{\underbrace{*}_{target} . \underbrace{*}_{selector}}^{Matching}]\overbrace{\underbrace{*}_{target} . \underbrace{*}_{selector}}^{Substitution}\}$$

Name: the name of this filter;
Type: the type of this filter, a filter type can thus be instantiated;
Condition: the condition is evaluated to determine whether to continue to the matching part. If this condition yields false, the filter will reject and execute its corresponding reject action. If it yields a truth value, the matching part is evaluated;
Matching: this allows for selecting a specific message. A matching part can match the *target* and/or the *selector* of a message. If a given message matches, the substitution part is executed, if any, and the filter accepts. This acceptance will result in the execution of the accept action;
Substitution: this allows for simple rewriting the target and selector of a message.

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 action, 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. Filter actions perform specific tasks of a filter type, e.g. filter action *Encrypt* of filter type *Encryption* will encrypt the arguments. Likewise, filter action *Trace* of the filter type *ParameterTracing* will trace the message. Most filter types execute the *Continue* action if the filter rejects. All this domain information is gathered and a so-called message flow graph is generated. A message flow graph G_{mflow} is a directed acyclic graph and is defined as: $< V, E, L >$, where:

V is a set of vertexes representing the composition filters elements that can be evaluated. These can be filter modules, filters, matching parts, condition expressions, filter actions, etc... ;

E: is the set of edges connecting the vertexes, such that $E = \{(u, v) \bullet u, v \in V \wedge u \neq v\}$;

L: is the set of resource-operations labels attached to the edges, such that $L = \{(e, rsrcop) \bullet e \in E \wedge rsrcop \in ResourceOperations\}$.

For each shared join point a message flow graph G_{mflow} is created. This graph is subsequently simulated to detect impossible or dependent paths through the filter set. It is out of the scope of this paper to discuss the full implementation of this simulation, please see [4] for more details. The general idea is that for each message that can be accepted by the filter we determine its effect on the filter set. If we do this for all possible messages, and once for those messages that are not accepted by the filterset, we are able to determine which filter actions can be reached and how they are reached. Impossible paths are removed and dependent paths are marked as such.

The filter sequence presented in listing 1.1 can be translated to the filter execution graph in figure 1.

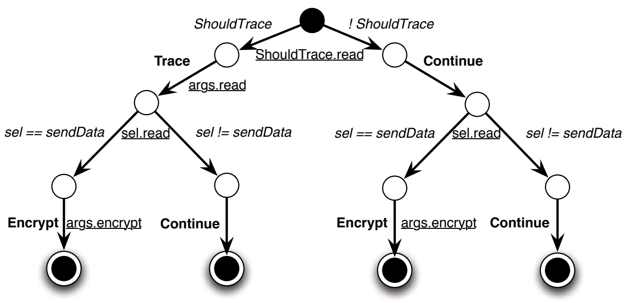

Fig. 1. Filter execution graph example

This graph is a simplified version of the actual graph, for readability purposes. The *italic* labels on the transitions are evaluations 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 underlined labels are resource-operations tuples corresponding to the evaluation of the conditions, matching parts and the filter actions.

Next we transform the conflict and assertion rules to graphs that are matched to the message flow graph. We require all assertion rules to be inverted, as the process for determining whether a rule matches only works for conflict rules. The assertion rules can be inverted, because we use a regular language and the alphabet is known and limited. A conflict rule graph $G_{conflict}$ is a directed acyclic graph and is defined as: $< V, E, L >$, where:

V is a set of vertexes;

E: is the set of edges connecting the vertexes, such that $E = \{(u, v) \bullet u, v \in V \wedge u \neq v\}$;

L: is the set of resource-operations labels attached to the edges, such that $L = \{(e, rsrcop) \bullet e \in E \wedge rsrcop \in ResourceOperations\}$. The label can also be a wildcard to indicate that we are that we are not interested in a certain step.

Once we have these conflict rule graphs we can intersect both graphs and see whether the intersection is empty. If so the conflict rule does not match and as such is conflict free for this rule. If the intersection is not empty, we have encountered a conflict and a trace is asked. To summarize, a shared join point contains a conflict if:

Lemma 1. $\exists g_{rule} \in G_{conflict} \bullet g_{rule} \cap G_{mflow} \neq \emptyset$

For each such conflict we have a corresponding path $P_{conflict}$, or a set of paths if there are more paths leading to the same conflicting situation. $P_{conflict}$ is a sub graph of G_{mflow}.

In this case a conflict rule stating that it is is not allowed for the arguments to be read before they are encrypted. In a regular expression: Conflict(arguments): ˆ*read encrypt*, this states that a conflict occurs if we encounter a situation where the arguments are read and afterwards they are encrypted. From this graph we can see that in the left most path, the *arguments* are *read* before they are *encrypted.* The intersection of the conflict rule with the message flow graph of shared join point *Protocol.sendData* is not empty, and thus the conflicting situation is detected.

Now let us elaborate on this conflict a bit more. In the example we use two filters, one of these filter uses a condition. Condition *ShouldTrace* is used to determine whether to trace this method or not. Whether this condition is true or false depends on some runtime configuration option. Statically we see that there is a possibility of a conflict, as we modeled both true and false values of the condition. This enhances our ability to reason about behavioral conflict but it also introduces possible false positives. The use of such a condition may always yield a false value, i.e. no methods should be traced. This thus requires dynamic monitoring to determine whether such a conflict actually occurs at runtime. The next section will discuss in which situations static checking is not sufficient, when using AOP.

3 Issues with Static Checking in AOP

The previous section outlined our approach to statically determine behavioral conflicts between advice. Although this provides a developer with a list of potential conflicts, not all these conflicts may occur at runtime. The simplest example of such a situation, is when the code in which the conflicting join point resides is never executed. However, there are more complex cases where static checking is not sufficient.

3.1 Dynamic Weaving

There are AOP approaches which employ dynamic weaving or proxy-base techniques to instrument an application. Although this provides some unique features over statically based weaving, it does present difficulties when statically reasoning about behavioral conflicts at shared join points. One such difficulty is that not all shared join points are known statically. As such, it becomes hard to know which advices are imposed at a shared join point. An example of such a construct is conditional superimposition found in Composition Filters. In this case one can assume a worst case situation, where each advice can be composed with any other advice. However, this can lead to large set of orders and possible combinations which have to be checked.

3.2 Dynamic Advice Execution

Most AOP approaches support conditional or dynamic properties in either pointcut or advice language. Examples of such constructs are, the *if(...)* pointcut in AspectJ and conditions in Composition Filters. In this case all shared join points are known. However, not all possible combinations of advice may occur at runtime. This can depend on some runtime state. In the running example we use condition *ShouldTrace* to determine whether to trace or not. At runtime this condition can be true or false. In our static approach, we simulate all possibilities of conditions.

3.3 Concurrency

In this paper we limit ourselves to the detection of concurrency conflicts at a single shared join point. We are aware that concurrency conflicts can also occur between join points. Concurrency conflicts between advice at a single shared join point, are caused by an unanticipated interleaving of the advices. This interleaving can occur because there is a single advice applied to a join point and that join point is concurrently executed. In this case the aspect is conflicting with itself and no resolution can be made. The interleaving can also be caused by a composition of multiple advices. In this case we may be able to resolve the conflict by changing the order. In both cases the problem can be prevented using atomic advice execution.

A single aspect or multiple aspects can cause concurrency conflicts. In either case, it is difficult to statically determine all possible interleavings. To determine the possible interleavings is not only hard, but also simulating all interleavings is very time consuming.

This section presented three situations where static checking is not sufficient. The next section will provide a runtime extension of the approach outlined in section 2. Although we focus on the second situation our approach is equally applicable to the first and third situation.

4 A Runtime Extension

As motivated by the previous section, we would like to extent our work to also capture behavioral conflicts at runtime. A naive application would be to simply instrument all advices and monitor all join points dynamically. This is required for capturing concurrency conflicts, as explained in the previous section. However, for the other two conflicting situations we can reason more efficiently. In section 2 about our approach we stated that for each possible conflict we get a set of conflicting paths called: $P_{conflict}$. This graph is translated into a DFA for checking at runtime. The nodes of this graph are elements that can be evaluated in Composition Filters. The edges represents the control flow between these nodes. Each edge has a set of labels attached to it which represent the corresponding resource operation tuples.

It should be noted that most likely the set of conflicting paths is smaller than the set of all possible paths. We only have to monitor paths that are conflicting for a specific resource and that contain dynamic elements. This will in practice reduce the number of paths to check substantially.

To informally outline our runtime extension we will use the example conflict, as presented earlier. In figure 1, we saw that the left most path was a conflicting path. This full path is: $< ShouldTrace.read >, < args.read >, < selector.read >, < args.encrypt >$. However, only part of this path is conflicting with our requirement. In this case: Conflict(args):^*read encrypt*. This conflict rule only limits the usage of operations for resource *args*. We can thus reduce the conflicting path to: $< args.read >, < args.encrypt >$. Where $< args.read >$ is caused by the execution of filter action Trace, and $< args.encrypt >$ is caused by filter action Encrypt. We only have to monitor the execution of these two filter actions to determine whether the conflict occurs or not. In this case, even the execution of filter action Trace is sufficient, this is however not true in the general case. There are cases where one has to monitor the evaluation of conditions, message matching and message substitutions.

4.1 Instrumentation

To be able to monitor the system while running, we have to inject monitoring code inside the advices. We assume that all code will be passed through the Compose* compiler. In our case this is always the case. However, with other more dynamic approaches this may not be a valid assumption. Our compiler will inject the bookkeeping code in the appropriate places. This ensures that the executing code will emit updates to the monitor. The next section will provide more details about this monitor.

4.2 Analysis Process At Runtime

There are multiple steps involved in checking at runtime for a behavioral conflict. Our runtime extension uses an Abstract Virtual Machine(AVM)[2] to do bookkeeping at runtime. This AVM is defined as:

ConflictingResources is the set of resources which should be monitor, where $ConflictingResources \subseteq Resources$,

[2] Note, that besides the name there are no similarities between the AVM and a runtime virtual machine, e.g. the JVM.

OperationSequence(rsrc) is the sequence of operations carried out on resource $rsrc$, where $\forall rsrc \in ConflictingResources$ $\bullet\, OperationSequence(rsrc) \subseteq Alphabet(rsrc)$,

ConflictRules(rsrc) is the set of conflict rules for resource $rsrc$.

Now that we have defined the monitor we define the three phases that are involved while reasoning about behavioral conflicts at a shared join point.

1. **Initialization:** At the start of the first edge in conflicting path we initialize the AVM. This AVM is responsible for keeping the state of resources during the execution of this join point. It keeps track of all operations that are carried out on resources. If an operation is carried out on a resource which does not exist, this resource is created.
   In our running example the initialization is done before the filter action *Trace* or the first *continue* action is executed.
2. **Execution:** For each edge involved in a conflicting path, we execute the operations on the conflicting resources. These are carried out on the AVM, and this AVM will update its state accordingly. The execution of the operations has to be done immediately and atomically, after the filter actions, conditions and such have been executed or evaluated.
   In the example, the execution step is carried out if the edge with label $< args.read >$ attached is taken. This corresponds to the execution of operation *read* on resource *args*. The result: $OperationSequence(args) = read$. The execution step is also executed for the edge with label $< args.encrypt >$ attached is taken. This corresponds to the execution of operation *encrypt* on resource *args*. Resulting in: $OperationSequence(args) = read\ encrypt$.
3. **Evaluation:** If we reach the end of the execution path, we have to signal the AVM to verify that the rules still hold for the given execution path. We have encountered a conflict if any of the conflict rules match. In such a case we can alert the user, e.g. via a message or an exception. At the end of a join point we verify that: $\forall conf_{rule} \in ConflictRules(rsrc) \bullet rsrc \in ConflictingResources \wedge conf_{rule} \cap$ $OperationSequence(rsrc) \neq \emptyset$.
   In the example case, this will occur after the edge, that is labeled $< args.encrypt >$, is taken. A conflict has been detected if the conflict rule matches.

The above process has to synchronize all conflict paths. Thus, start monitoring at the beginning of the first conflicting path. Similarly, at the end of the execution, the evaluation phase has to be performed at the correct time. To reduce the complexity of this, we could easily initialize the VM at the start of the join point. Similar, we could simply check at the end of the join point execution. However, these simplifications might impose a larger runtime performance hit.

Another option would be to verify the rules continuously. This would provide possibly earlier detection of the conflict. However, the runtime performance might also be decreased, due to the abundance of verifications.

5 Related Work

There is a lot of work on static analysis of AOP languages. Most of these limit themselves to detecting interaction. In some cases even the presence of a shared join point is considered a issue.

One approach to program verification is to utilize traditional model checking techniques. Krishnamurthi et. al. propose one such approach in [5]. The paper considers the base program and aspects separately. The author state that a set of desired properties, given a pointcut descriptor, can be verified by checking the advice in isolation, thus providing modular reasoning. The paper focuses on ensuring that the desired properties are preserved in the presence of aspects, in other words, the situation where applying aspects causes the desired properties of the base system to be invalidated. The paper only considers aspect-base conflicts and not conflicts between aspects.

In [6], Katz et. al. propose an approach to use model checking to verify aspects modularly. The authors create a generic state machine of the assumptions of an aspect. If the augmented system, the generic system machine with the aspect applied, satisfies certain desired properties, then all base systems satisfying the assumptions of the aspect will satisfy the desired properties. The proposed technique has several limitations, for example the restriction to a single aspect and pointcut designator, and thus can only detect base-aspect conflicts, and not conflicts between aspects at shared join points.

Another aspect verification approach is based on graph transformations. In [7], Staijen and Rensink model, part of, the Composition Filters behavior with graph based semantics. The result is a state space representation of the execution of the composed filter sequence at a shared join point. The paper proposes an interference detection approach based on the ordering of filter modules on this resulting state space. If the different orderings of the filter modules result in different state spaces, the program is considered to have a filter module (advice) composition conflict. This approach also detects aspect-aspect conflicts, but only detect an interaction. There is no way to state whether such an interaction desirable or undesirable.

In several papers (e.g. [8] and [9]), Südholt et. al. present a technique to detect shared join points, based on similarities in the crosscut specification of the aspects involved. If there is no conflict the aspects can be woven without modification, else the user has to specify the order in which the aspects should be composed. The approach does not consider the semantics of the advice on *inserts*, it just considers the presence of a shared join point to be an interaction.

There is also a lot of work about runtime verification of systems. However, these techniques are not immediately suitable for AOP languages, as these languages implement new constructs and can alter the base system even during runtime. This makes it harder to statically instrument or verify the base system and to know the exact composition of all elements. Nonetheless, especially for dynamic AOP approaches, providing runtime verification of advice and the composition of advice is important.

The notion of using resources and operations on these resources to model dependencies and conflicts has already been applied in many different fields in software engineering, e.g. for synchronization constraints [10] and for transaction systems[11].

6 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 that 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. Durr, P., Bergmans, L., Aksit, M.: Reasoning about semantic conflicts between aspects. In: Chitchyan, R., Fabry, J., Bergmans, L., Nedos, A., Rensink, A. (eds.) Proceeding of ADI 2006 Aspect, Dependencies, and interactions Workshop, Lancaster University, pp. 10–18 (July 2006)
2. Durr, P.E.A., Bergmans, L.M.J., Aksit, M.: Reasoning about behavioral conflicts between aspects. Technical Report TR-CTIT-07-15, Enschede (February 2007)
3. Group, T.O.: IEEE: Regular expressions. The Open Group Base Specifications, IEEE Std 1003.1 (6) (2004)
4. de Roo, A.: Towards more robust advice: Message flow analysis for composition filters and its application. Master's thesis, University of Twente (March 2007)
5. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: SIGSOFT 2004/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pp. 137–146. ACM Press, New York, USA (2004)
6. Goldman, M., Katz, S.: Modular generic verification of ltl properties for aspects. In: Clifton, C., Lämmel, R., Leavens, G.T. (eds.) FOAL: Foundations Of Aspect-Oriented Languages, pp. 11–19 (March 2006)
7. Staijen, T., Rensink, A.: A graph-transformation-based semantics for analysing aspect interference. In: Workshop on Graph Computation Models, Natal, Brazil (2006)

8. Douence, R., Fradet, P.: Trace-based aspects. In: Filman, R.E., Elrad, T., Clarke, S. (eds.) Aspect-Oriented Software Development, pp. 201–217. Addison-Wesley, Boston (2005)
9. Ségura-Devillechaise, M., Menaud, J.M., Fritz, T., Loriant, N., Douence, R., Südholt, M.: An expressive aspect language for system applications with arachne. Transactions on AOSD I 1(1), 174–213 (2006)
10. Bernstein, A.J.: Program analysis for parallel processing. In: IEEE Trans. on Electronic Computers. EC-15, pp. 757–762 (1966)
11. Lynch, N.A., Merritt, M., Weihl, W.E., Fekete, A. (eds.): Atomic Transactions: In Concurrent and Distributed Systems. Morgan Kaufmann, San Francisco (1993)