Reasoning About Semantic Conflicts Between Aspects

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

ABSTRACT

The AOP community has successfully promoted and illustrated the power and elegance of aspect-oriented programming. One of the main problems of Aspect-oriented programming is, however, the aspect interference problem. When multiple aspects are superimposed on the same join point, undesired or incorrect behavior may emerge due to the side effects of behavior of the aspects at the join point. In this paper we present a technique and a tool to detect and correct the semantic conflicts among aspects that are superimposed on the same join point.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) aims at improving the modularity of software in the presence of crosscutting concerns. In component-based programming, each component explicitly specifies its dependencies to its environment, for example, through *import* and *export* declarations. Separating interface declarations from implementation has provided sufficient information to predict the emerging behavior of components in a given application context.

In aspect-oriented programming, superimposing aspects on software modules may cause side effects that cannot be encapsulated in the implementation of these modules. The reasoning techniques on components, therefore, cannot be directly applied to aspect-oriented programs. In fact, as pointed out in several workshops and publications[11, 12, 10], reasoning about the correctness of a system after superimposing multiple aspects on the same join point has been considered an important issue to be addressed.

The application of aspects also brings a more subtle, but a prominent problem: different aspects possibly developed by different software engineers at different time, are superimposed on the same join point may semantically interfere with each other and with other program modules in undesired manner.

Although the semantic interference problem can also be observed in other programming paradigms, due to the specifics of aspect-oriented superimposition mechanisms, new techniques are necessary to deal with this problem within the AOP context.

In this paper we present an approach to analyze and detect semantics conflicts at shared join points[14]. The paper is structured as follows; first we will provide an example of a semantic conflict, based on a Jukebox system, discuss the origin and types of semantic conflicts. Section three provides a discussion about Composition Filters, and the example is revised. In section 4 our conflict detection model is presented. We also show some implementation details of the Compose*[1] tool, which uses this model. Finally we conclude and provide an overview of related work.

2. EXAMPLE

Consider the system shown in figure 1 representing a Jukebox system.

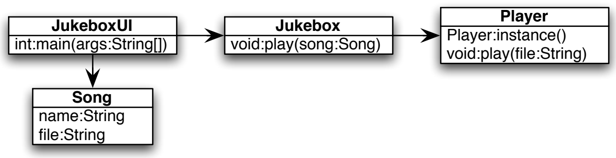

Figure 1: Jukebox system

If a song is selected on the interface of the *JukeboxUI*, the method *play(Song)* is called on *Jukebox*, passing the song as an argument. This method calls the *play(String)* method on the player, which is the interface to the audio-subsystem. We will now add two new requirements to the Jukebox system. The first requirement states that we have to check if the user has enough credits. If the user has enough credits, we have to withdraw one credit, each time the play method is called. The second requirement states that we should create a playlist which queues all songs, thus releasing the caller. We use aspect-orientation to implement both requirements.

The first aspect checks the number of credits before withdrawing one credit and proceeding with the execution. An implementation is AspectJ[7] is given in listing 1.

```
aspect CreditsAspect {
   void around(): call(public void Jukebox.play(Song)) {
      if(Credits.instance().enoughCredits()) {
         Credits.instance().withdraw();
         proceed();
      } else {
         throw new NotEnoughCreditsException();
      }
   }
}
```

Listing 1: Credits aspect

A playlist aspect queues songs and immediately returns to the caller. A possible implementation in AspectJ is shown in listing 2. In the advice, the *enqueue(Song)* method is called on the *Playlist* instance, which is a singleton. This method puts the *Song* object in a queue and, if the player is idle, starts a thread that plays songs until the queue is empty.

```
aspect PlaylistAspect {
   void around(Song song): call(public void Jukebox.
        play(Song)) && args(song) {
      Playlist.instance().enqueue(song);
      return;
   }
}
```

Listing 2: Playlist aspect

If we examine the pointcuts of both *Credits* and *Playlist*, we can see that they both select the *Jukebox.play(Song)* call. Both advices are, thus superimposed on the same join point. This is also referred to as a shared join point. If no precedence between the aspects has been declared, the advices can be ordered in two ways. In the first order the *Credits* aspect is applied first and subsequently the *Playlist* aspect. In the other order first the *Playlist* aspect is applied and then the *Credits* aspect. However, if the playlist advice is executed first, the credits advice is not reached anymore. As a results songs are played without checking for sufficient credits and, if so, withdrawing one credit. This is due to the *return* statement in the *Playlist* aspect. It should be noted that this conflict only occurs if the second aspect order is chosen by the weaver. However, without specifying any precedence relationships between aspects this problem may occur. This is a simple example where the semantics of one advice affects the behavior of another advice and/or the base. We use the term “semantic conflict” to designate such problems.

In AspectJ, these kinds of problems should be recognized by the programmer, and can easily be fixed using *declare precedence*[1]. However our goal in this paper is to provide an automatic detection mechanism for such conflicts.

2.1 Origins of Semantic Interference

Aspect-orientation provides a mechanism to modularize cross-cutting concerns. These concerns can be separately and independently engineered and maintained by different developers at different times. Especially in large systems with numerous crosscutting concerns like logging, tracing, debugging, contract enforcement and error handling, compositions of all these aspects occur which are unintended and even undesired. Whether the composition may be desired or undesired, is a matter of requirements. Several reasons can be given why such unintended compositions are created, see [4].

Firstly, the use of open-ended pointcut specifications limits the applicability of the pointcut. The use of patterns like; *set** and *return type int*, are useful but the full impact may not be known at the time the pointcut is written. One may not know what future functionality is added to the base system, that also matches with the pointcut specification.

Secondly, the use of multiple domains can decrease the uniform application of the pointcut. Imagine an aspect which does error handling on the basis of an integer return value, indicating success or failure, specifying a pointcut like: *call(int *.*(..))*. This would however also cover methods which perform mathematical arithmetic, for example *int add(int rhs)*.

Finally, the use of dynamic information in a pointcut expression can also limit the ability to reason about and predictability of the pointcut. In such situations one is not sure whether the composition even occurs, and if so, whether the introduced behavior is actually executed, e.g. the use of the if-construct in AspectJ. The usage of such constructions limits the possibility to reason about applicability of the aspects severely as one can not determine the impact of the aspect at compile-time.

All of the above stated issues contribute to the possible unintended composition of multiple aspects at the same point in the source code or execution. In fully expressive pointcut languages like AspectJ's, detecting these shared join points is not easy, but doable.

2.2 Semantic Conflicts

We do not detect syntactical conflicts that can occur at shared join points, e.g. changing the superclass while another class depends on the original inheritance tree. These kinds of issues are usually captured by the typing system of the underlying language.

The problems we address, are related to semantic interference between aspects. These kind of issues are extremely hard to detect, as these are syntactically sound and, thus compile without any problems. Only when the composed application executes, these problems exhibit themselves.

We distinguish two kinds of semantic conflicts. The first are those conflicts that occur in the Composition Filters[5] domain. For instance, conflicts with respect to the message execution, message property manipulation or synchronization issues. An example of such a conflict is that some filter ends the evaluation of the filters and, as a result, a precondition check is never carried out. The second kind of conflicts are those that conflict in the application or domain requirements. These cannot be detected without extra information about the specific application requirements. An example of the last conflict is the play without paying conflict presented earlier.

3. COMPOSITION FILTERS

The example described in section 2 showed a possible ordering conflict, namely the precondition is never verified. In order to reason about these kinds of conflicts we need to be able to reason about the separate advices and about the composition of these advices. Reasoning about full programming languages like Java or C# is hard. In Composition Filters(CFs) this can be done much more elegantly because they provide a declarative specification of advice. Advices in CFs are sets of filters, called a filtermodule. The filters in this set are declarative and composed out of several elements:

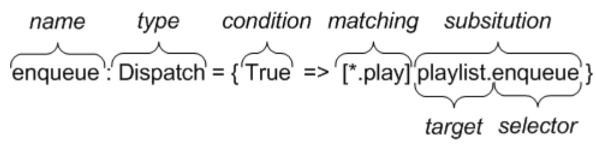

Figure 2: Filter structure

name : The identifier of the filter.

1 This is however on an aspect level and not on an advice level, thus a fine grained ordering scheme is not possible.

type : The type of the filter, e.g. Dispatch, Meta or user-defined.

condition : A boolean expression representing some state, under which this filter is applied.

matching : The matching expression indicates, in which set of messages this filter is interested.

substitution : The substitution part specifies which properties are replaced if both condition part and matching part yield a true value. This is currently limited to the target and selector of the message.

target : The object part of a message.

selector : The method part of a message.

A filter, in the CFs, has the following properties:

- Domain or application knowledge is incorporated in the type of the filter. A dispatch filter encodes knowledge about the message execution manipulation domain. Similar, a wait filter uses knowledge about the synchronization domain. We can use this knowledge to reason more effectively about the filters, without requiring any user defined specification.
- The filter's behavior is deterministic, as they can only accept or reject but not both at the same time. This makes it easier to reason about the impact of the filters.
- The applicability and acceptance of a filter is determined by a condition expression and a message matching expression, which will be evaluated both.
  - The condition expression allows one to access the system state and derive applicability constraints. The condition can be composed of multiple boolean expression linked to each other via boolean operators.
  - The message matching expression specifies which messages are considered while evaluating a filter. An incoming message is matched to both the target and selector of the matching expression.
- If both the condition and matching expressions yield a true value, the filter accepts and the associate substitution(s) and filter action(s) are executed. In all other cases the filter will reject and will execute all the reject actions.

All of the above stated properties enable us to reason about the filters. We know under which conditions and with which set of messages the filter will accept and reject. We also know the impact of the accept and reject substitutions and actions. Filters are composed with each other into a superimposition unit called a filter module. This composition operator is a simple sequence operator. The result of superimposing several filter modules on the shared join point is a new sequence of filters. The order within the filtermodule is fixed, however this is not the case when composing multiple filter modules where the order is arbitrary. This simple composition operator enables to, also reason about the composition of multiple filtermodules.

3.1 Jukebox revisited

The previous section provide an overview about the reasonability of Composition Filters. Before we continue with the actual conflict detection model we first present a Composition Filters implementation of the Jukebox system. Listing 3 shows a possible implementation of the *Credits* concern.

```
concern CreditConcern {
  filtermodule TakeCredits {
    externals
      credits: Jukebox.Credits = Jukebox.Credits.
           instance();
    conditions
      enoughCredits: credits.enoughCredits();
    inputfilters
         check : Error = { enoughCredits => [*.*],
              True ~> [*.play] };
         withdraw : Meta = { True => [*.play]credits.
              withdraw }
  }
  superimposition {
      selectors
         classes = { Class | isClassWithName(Class,'
              Jukebox.Jukebox') };
      filtermodules
         classes <- TakeCredits;
  }
}
```

Listing 3: Credits concern

In the *superimposition* part, the filtermodule *TakeCredits* is superimposed on class *Jukebox*. The filtermodule uses an external *credits* of type *Jukebox.Credits*, which can be seen as a global variable. The first input filter of this filtermodule, *check*, is an error-filter, which throws an exception if the filter does not accept a message. Without going into the details of the Compose* syntax, this filter will only reject *play(Song)* methods whenever the *enoughCredits()* method of external *credits* returns *false*. The second filter is a meta-filter, *withdraw*. If a meta-filter accepts, it reifies the message - it creates an object representing the message - and sends it to a so-called *AdviCe Type* (ACT), where user-defined advice can be used. In this case the message is sent to the *withdraw(ReifiedMessage)* method of the *credits* external. This method withdraws one credit and then continues the message through the filters.

Similar, the Playlist concern can be constructed as shown in listing 4.

```
concern PlaylistConcern {
  filtermodule Enqueue {
    externals
         playlist: Jukebox.Playlist = Jukebox.
              Playlist.instance();
      inputfilters
         enqueue : Dispatch = { True => [*.play]
              playlist.enqueue }
  }
  superimposition {
      selectors
         song = { Song | isClassWithName(Song, '
              Jukebox.Jukebox') };
      filtermodules
         song <- Enqueue;
   }
}
```

Listing 4: Playlist concern

The filtermodule *Enqueue* is also superimposed on Jukebox objects. It uses an external of type Playlist where queued Song objects are stored. The dispatch filter, which matches all *play(..)* methods, redirects the message to the *enqueue(Song)* method in the *playlist* external.

4. CONFLICT DETECTION MODEL

As stated, the Composition Filters approach enables us to reason more easily about advice, without complete program analysis and domain knowledge of the specific application. These characteristics are exploited in our conflict detection model.

4.1 Resource Model

We need a canonical model to reason about the filters and the composition of these filters. This model should be used to model both filter specific and application specific information. We have chosen to adopt a resource-operation model, as this is an easy to use model that can represent both very concrete, low-level, semantics and very high-level, abstract behavior. For more detailed information about the model and its usage we refer to [16]. Our approach of conflict detection resembles the Bernstein[6] read-write sets for detecting possible deadlocks. A similar approach is used for the detection and resolution of conflicts in transaction systems, like databases[13].

We define a resource as being either a:

1. a (composed) concrete property of the system or message, e.g. the sender of a message, or
2. an abstract or application specific concept that encapsulates or indicates the problem area in the best way.

Examples of resources we have defined are;

target : The target of a message

selector : The selector of a message

⟨condition⟩ : All conditions used in the filters, are mapped to resource with the corresponding names.

message : The actual message execution

In order to detect a conflict in our example case, we introduce a new application specific resource, *song*. The resources are used to capture the conflicting areas between aspects.

4.2 Resource Usage

Once we have defined the resources we can map the filters to operations on these resources. It should be noted that all the filters will evaluate the conditions and the matching expressions. This is true for all filters and these operations are carried out first. The evaluation of the conditions is considered a read on that specific condition resource. The matching expression *[*.play]* will result in a read of the selector but not the target as we do not care about its value. Furthermore, a dispatch action will write the target and dispatch the message, thus: *target.write*. The error action results in an exception in the caller, we interpret this as a *exception* message to the caller, thus changing the target of the message to the sender and substituting the selector. We model this in the following way: *sender.read*, *target.write* and *selector.write*. We can also express application specific information in this manner. Consider the dispatch filter in listing 4: $enqueue : Dispatch = \{True => [*.play]playlist.enqueue\}$. If it accepts it will play the song. We can interpret this as a *play* operation on the *song* resource.

In section 3 we provided an implementation of the Jukebox system in Composition Filters. The problematic filter-module order is repeated here:

```
enqueue : Dispatch = { True => [*.play]playlist.
    enqueue }
check : Error = { enoughCredits => [*.*], True ~> [*.
    play] }
withdraw : Meta = { True => [*.play]credits.withdraw
    }
```

Listing 5: The problematic filter-module order.

We can now transform these filters into accept and reject actions. For each action we describe its resource usage. This usage is presented in table 1.

Filter	Resource-Operation tuples	
enqueue	*dispatchAction*	*continueAction*
	selector.read	selector.read
	target.write	
	selector.write	
	song.play	
check	*errorAction*	*continueAction*
	enoughCredits.read	enoughCredits.read
	sender.read	
	target.write	
	selector.write	
withdraw	*metaAction*	*continueAction*
	selector.read	selector.read
	song.pay	

Table 1: Filters mapped to resources

4.3 Execution-Trace Derivation

We can represent a set of filters as a binary tree of filter actions. In this tree we can incorporate knowledge about filters; e.g. after a Dispatch or Error, the rest of the filters are no longer evaluated. For our Jukebox example the tree is shown in figure 3. One should note that although the path $enqueue_{reject}$, $check_{accept}$, $withdraw_{accept}$ looks valid, this path cannot possibly occur as the matching expressions are identical. Thus, if the *enqueue* filter rejects, the *withdraw* cannot possibly accept. The image shows each filter as a node with two edges, one for the accept action(A) and one for the reject (R) action. The gray nodes and dashed edges indicate states and action which can never be reached. For example, if the dispatch action of the *enqueue* filter is executed, no more filters are evaluated.

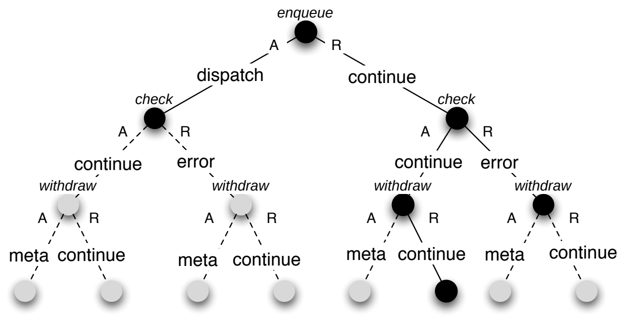

Figure 3: Filter action tree

From this filter tree we can derive all possible traces of filter actions. An action trace is a unique path from the root node to an end node. The traces for the Jukebox are as follows:

1. $enqueue \xrightarrow{dispatch}$
2. $enqueue \xrightarrow{continue} check \xrightarrow{error}$
3. $enqueue \xrightarrow{continue} check \xrightarrow{continue} withdraw \xrightarrow{continue}$

It should be noted that the number of possible traces are considerably less than the number of possible paths through the filterset, $2^3 = 8$ versus the current 3. For each of these traces we generate a new set of resources-usages. After all filters have been evaluated we end up with, for each trace, a sequence of operations for each resource, see table 2, which shows the result of trace 1.

action	target	selector	song
enqueue:continue	write	read, write	play
check:continue			
withdraw:meta			

Table 2: Actions mapped to operation traces

Only the *enqueue* filter contributes to the resource-operation model, as the *check* and *withdraw* filters are never evaluated in this trace.

With these sequences of operations we can determine whether there are conflicts present.

4.4 Conflict Detection

We express, for each resource, a conflict specification as a required or disallowed pattern of operations. For example if we want to ensure that a certain precondition check is executed, we constrain the allowed operations on that specific condition to have at least one read. The exact conflict specification is discussed in section 5. There are various ways to determine whether a sequence is allowed or not. These include; regular expression, temporal logic and trace theory. We recognize two kinds of conflict specifications, assertions and exceptions. These can all be applied to the resulting traces. Assertions specify patterns that must occur. Exceptions are patterns that may not occur.

In order to detect the presented play-without-paying conflict we require that all traces for the song resource must either include *pay-play* tuples or *play-pay* tuples, meaning that a song that is played should always be paid. This specification will not match the trace for the *song* resource that was presented in table 2. There is only one *play* operation and no *pay* operation before or after it, thus it violates the specification.

The next section will provide some more details on the implementation of the described model.

5. CONFLICT DETECTION & COMPOSE*

Compose* is an implementation of Composition Filters (CF) [5] for the Microsoft .NET platform[2]. The Compose* toolset consists of many modules, including a parser, static analysis tools, and a weaver. The tool that implements the previously described conflict detection model and mechanism is the SEmantiC REasoning Tool (SECRET). In order to implement the model, we use an XML-file which specifies the resource-operations for all filtertypes. This file also contains the specification of conflicts, consisting of a pattern and a corresponding warning-message. An example of such a conflict specification is presented in listing 6.

```
<constraints>
   ...
   <assert
      pattern="^[((pay)(play))((play)(pay))]?$"
      resource="song"
      message="A played song is never payed for or a
          paid song is never played!" />
</constraints>
```

Listing 6: Example conflict specification

The specified resource-operations are used to translate a given filter specification to our model. However, for meta-filters, the behavior is not determined by the filter-type but given by the method of the ACT. To be able to include the semantics of ACT methods, we allow this to be specified in an annotation attached to the method, see [17] for more details. Listing 7 shows an example of such an ACT method. This method is called by the *withdraw* filter of the running example. The tight coupling between the implementation and the semantic specification will prevent mismatch when for instance the implementation is modified.

```
class Credits
{
   private int credits = 0;
   ..

   [Semantics("song.play")]
   public void withdraw(ReifiedMessage m) {
      credits--;
   }
}
```

Listing 7: ACT with semantic annotation

In our running example, we want to ensure that one always pays for a song. This requirement cannot be expressed and ensured with the basic filter know-how, without the extra knowledge from the Jukebox domain. In order to incorporate these application requirements we have chosen to annotate the filters with this specification. This is currently not implemented, and should be specified separately.

Before SECRET starts, all superimpositions are resolved and the corresponding filtermodules are attached to the target concerns. Subsequently, all orderings of filtermodules at the same concern are calculated. Then, according to an optional precedence-specification[14], only the allowed filtermodule orders remain as possible orderings. If at this point there are more than one orderings, one order is arbitrary selected. Then SECRET starts to analyze one concern at a time. Analysis can be performed in three ways:

- Only the selected filtermodule-order is analyzed.
- All filtermodule-orders are analyzed.
- All filtermodule-orders are analyzed and - if possible - one without conflicts is selected.

When a filtermodule-order is checked the filters are combined in a new filterset. Then all executions of the filterset are generated. Every execution is then transformed into a resource-operation sequence. At this point, conflicts are detected by matching the sequence of operations with all specified conflict- and assertionpatterns, these are currently regular expressions.

6. DISCUSSION

In the model we generate the possible tree of executions and analyze all traces. In the implementation we generate all traces and, if any conflicts are found, verify if the trace if possible to occur. This allows us even to detect assertion patterns that occur, but can never be reached. For example, we now that a certain precondition is present but never executed. We can provide the user with this information.

If possible conflicts are detected in an execution, but caused by filters within a single filtermodule, one could argue that possibly conflicting behavior is desired. Image for instance that the error- and meta-filter in our CreditsConcern were located in different filtermodules. If we would detect this (by adding a conflict-specification accordingly) the error-filter avoiding the meta-filter to be executed could be a conflict. In the actual example the programmer probably intended the behavior because the filters are in a single filtermodule.

Our tool provides an auto-correcting facility, as it is able to choose a correct advice application order. If after analysis of all possible orderings one order is found to be clean of conflicts this one is chosen. If there are no conflict free orderings, no change is made to the selected order.

We are currently working on an automatic resource derivation tool for our meta-filters. As stated earlier, these user-defined advice types are written in a full programming language, which makes reasoning about them not easy. However, the ACT gets a *ReifiedMessage* object as an argument. This *ReifiedMessage* class has a well defined interface. This allows us to relatively easy extract the operations of the resources. For example, the *ReifiedMessage* has a *setSelector(String sel)* method. We can translate this to a write operation on the selector resource.

The conflicts we detect are treated by the Compose* compiler as warnings, they will never terminate the compilation process. Although for some conflicts we are able to say that this is an error for sure, based on user provided requirements. There are still a number of false positives, these are largely caused by two reasons. As stated, we currently do not distinguish between conflicts between elements in one advice or between advices. This results in probably a number of false positives, as these might be intentional. Secondly, we are currently do not take implicit dependencies between filters into account. These dependencies are similar matching expressions and/or inverted conditions. In the first case, the matching expressions matches on the same or different messages. This means that if one filter with *[*.play]* accepts that another filter with *[*.play]* must also accept. In the second case, if one filter only accepts if condition $enoughCredits$ is true, than another filter depending on a $\neg enoughCredits$ condition can only reject..

Finding the right resources and operations is far from trivial. One requires deep knowledge about the domain and has to throughly analyze conflicts to determine the exact interaction area. This interaction area must subsequently be translated to a resource and the conflicting parties should perform a meaningful, and preferably reusable, operations on this resource. Finally the exact conflict has to specified in terms of required and/or disallowed patterns. The resources, operations and conflicts specifications can not only be derived from the advice specification. They can also be used to convey and enforce design restrictions or requirements.

7. RELATED WORK

There has not been that much work on the detecting and interference analysis of semantic conflicts between aspects. In [8], Douence, Fradet and Sudholt present a framework for the formal definition and interaction analysis of stateful aspects. In [15], Pawlak, Duchien and Seinturier present a language called *CompAr*, which allows the programmer to define an execution domain, the semantics and the execution constraints of an aspect in order to check if the execution constraints are fulfilled when aspects share a join point. Balzarotti, Casteldo and Monga[3], propose an approach for slicing AspectJ woven code. They are also able to detect interference between aspects by checking whether the nodes of one aspect appear in the slice of another aspect; if this is the case, there may be interference between the aspects.

8. CONCLUSION

This paper presents a novel approach for detecting semantic conflicts between aspects. Our approach defines the semantics of advice in terms of operations on an abstract resource model. After analyzing all advices at a shared join point, we are able to detect conflicts based on required and disallowed sequences of operations on these resources. The resource-operation model allows us to express knowledge about the behavior of filters and provides a convenient way to specify application-specific behavior.

The presented approach is generic; it can be applied to any AOP language. it would require annotating all advices in an AspectJ-like language, with a resource-operation specification. If one, subsequently, is able to detect shared join points in such a language, we can also reason about the behavior of the composition of multiple advices. In the Composition Filters approach we can already derive—parts of—the semantics from the filter type specification, which is provided only once, and reusable in many places. We only require explicit input specifications for domain-specific or application-specific information. We believe this approach offers a powerful and practical means of establishing semantic conflict detection with a minimal amount of explicit behavior specifications from the programmer.

9. ACKNOWLEDGMENTS

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.

10. REFERENCES

[1] COMPOSE*. http://composestar.sourceforge.net.

[2] MICROSOFT .NET FRAMEWORK. http://www.microsoft.com/net/.

[3] D. Balzarotti, A. Castaldo, and M. Monga. Slicing aspectj woven code. In *FOAL '05: The 4th Workshop on Foundations of Aspect-Oriented Languages*, Chicago, USA, March, 14 2005.

[4] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In *ECOOP:AAOS '03: The first workshop on Analysis of Aspect-Oriented Software*, Darmstadt, Germany, July, 21 2003.

[5] L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In Filman et al. [9], pages 63–95.

[6] A. J. Bernstein. Program analysis for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15:757–762, 1966.

[7] A. Colyer. AspectJ. In Filman et al. [9], pages 123–143.

[8] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, Mar. 2004.

[9] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[10] J. Hannemann, R. Chitchyan, and A. Rashid. Analysis of aspect-oriented software, workshop report. In *ECOOP 2003 Workshop Reader*, Darmstadt, Germany, July, 21 2003.

[11] G. T. Leavens and C. Clifton. In *Foundations of Aspect-Oriented Langauges Workshop at AOSD 2004*, volume 3rd. AOSD, 2004.

[12] G. T. Leavens and C. Clifton. In *Foundations of Aspect-Oriented Langauges Workshop at AOSD 2005*, volume 4th. AOSD, 2005.

[13] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions : In Concurrent and Distributed Systems*. Morgan Kaufmann, 1993.

[14] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *Proceedings of International Conference NetObjectDays, NODe2005*, Lecture Notes in Computer Science, Erfurt, Gergmany, 2005. Springer-Verglag. to be published.

[15] R. Pawlak, L. Duchien, and L. Seinturier. Compar: Ensuring safe around advice composition. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems*, Athens, Greece, June 2005.

[16] P.E.A. Durr. Detecting Semantic Conflicts Between Aspects, 2004. http://www.cs.utwente.nl/˜durr/papers/Master_Thesis_Pascal_Durr.pdf.

[17] T. Staijen. Towards Safe Advice: semantic reasoning about advice types in Compose*, 2005. http://www.cs.utwente.nl/˜staijen/papers/MasterThesisTomStaijen.pdf.