

Towards Detection of Semantic Conflicts between Crosscutting Concerns

Lodewijk M.J. Bergmans
lbergmans@acm.org

TRESE group, Dept. of Computer Science, University of Twente
<http://trese.cs.utwente.nl>

1. Introduction

Aspect-oriented programming introduces new composition operators that offer enhanced capabilities for constructing well-modularized programs. One of its cornerstones is the introduction of 'reverse' dependency relations; instead of explicitly 'importing' composed behavior (as is e.g. the case with inheritance), the composed behavior is 'exported'. These reverse composition relations are more error-prone than conventional composition relations, in particular when multiple composition relations affect the same part of the program.

This paper should be read as a true 'position statement': we describe our current position and envisioned approach towards addressing this problem. The paper is hence limited to a global explanation of the approach, with little technical detail and no concrete results to be shown.

After the introduction of a motivating example, the paper first highlights some relevant problems, and states a few goals to be addressed by this research. Four key properties are proposed that have an important influence on the ability to reason about compositions of aspects.

Throughout this paper, we will use the following –simplified– example, first to explain the problems we will address, and later to illustrate the approach we propose. We will visually distinguish the example from the surrounding text by horizontal lines:

example: Assume a module (e.g. a class) *TextDocument* that serves as an abstraction to create and modify text documents. It has the following methods for manipulating and retrieving its contents: *gotoLine(int)*, *int currentLineNr()*, *String currentLine()*, *insertLine(String)*, *deleteLine()*, *int searchStr(String)*, *save(String)*, *load(String)*. These

methods are accessible from all parts of the system. The system is divided into –at least– the following layers: the application layer and the system layer. Class *TextDocument* has also the following system-level methods:

- *compact()* : compacts the internal data structure of the document.
- *Boolean isDirty()* : returns a Boolean flag that is true when the document has been modified since the last time it was saved.
- *Vector flattened()* : returns a flattened (marshalled) version of the document in the form of a Vector of binary words.

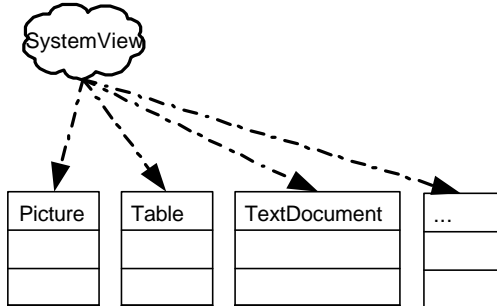
One of the architectural constraints is that these last three methods should be invoked only by the *System* layer, not by the *Application* layer. This avoids inadvertent breaking of system integrity by application-level code.

Aspect-orientation is orthogonal to object-orientation, even though most proposed aspect-oriented programming (AOP) languages and models extend the object-oriented (OO) model, rather than replacing it. The OO model offers several composition techniques, such as aggregation and inheritance, aspect-orientation introduces an additional composition operator, which we refer to as *super-imposition*¹. An essential characteristic of super-imposition is that it *exports* behavior to other modules, instead of letting the other modules *import* the behavior explicitly. Consider for instance the following example case:

example: previously, we explained that class *TextDocument* contains a number of methods (*compact()*, *isDirty()* and *flattened()*) that are intended to be invoked only by modules that belong to the system layer, but not by the application layer. The same

¹ Please note that this bears resemblance to, but is truly distinct from the technique of superimposition as proposed in e.g. [Bougé 88][Katz 93].

constraint is relevant for several (methods in) other classes in the system, such as *Table*, *Picture*, and *Authentication*. Aspect-Oriented allows for expressing the application of this and related constraints (so-called *crosscutting concerns*) on multiple methods and multiple classes, in a single module, say *SystemView*. The following figure shows these modules and the *superimposition* dependencies between them:



Typically, the cloud *SystemView* is referred to as (playing the role of) the *aspect*, and the other modules are (playing the role of) *base*² classes. The important feature illustrated by the above figure is that the aspect *SystemView* affects the behavior of the base classes, without any code within these base classes referring back to the *SystemView* aspect.

2. Problem statement

From the point of view of modularity, adaptability and hence evolvability, the transparency of superimposed aspects to the base classes (or 'obliviousness' [Filman 01]) is considered a blessing, since it reduces the dependencies between modules. However, understanding the exact behavior of a module, and analyzing or verifying correctness, require a global overview and understanding of all modules and aspects that might affect the module under consideration.

The particular predictability problems that we are focusing on, are those that are caused by interference between the modules that are composed. This is also referred to as the aspect interaction problem, and is a special case of the so-called feature interaction problems.

example: the *SystemView* aspect can be expressed by the following constraint (in pseudo-code):

```
SystemView : if callerIsSystem()
  then accept {compact, isDirty, flattened}
  else accept {gotoLine, currentLineNr, currentLine, insertLine,
              deleteLine, searchStr, save, load }
```

Here 'accept' is a keyword that means that any of the subsequent messages can be executed, but no others; therefore in this example, if the condition *callerIsSystem()* holds, *only* the *compact()*, *isDirty()* and *flattened()* messages can be accepted, otherwise all the other messages of *TextDocument* (as written in

the else branch). The condition *callerIsSystem()* is determined by testing whether the sender of the message belongs to the group of system classes; we assume that this condition is implemented by a method.

There are several ways how this aspect may not behave as expected or desired: for example, it makes certain assumptions about the availability of the *callerIsSystem()* method and the *compact()*, *isDirty()* and *flattened()* methods. If this assumption does not hold, the program might not execute. Note that this category of problems is traditionally addressed by typing systems. Another problem could be that the assumption about which methods should only be executed by system classes is incorrect or incomplete. One more example to illustrate what may go wrong in this example is the effect of the *accept* keyword for methods that do not apply; this effect (e.g. raising an exception) must match the desired semantics of the application, and perhaps requires special attention from the base module, such as catching the exception.

In this paper we focus on the occasions where multiple aspects affect the same base modules or join points: as we pointed out above, this may be unnoticed by the developers, but it is not unlikely that the combination will result in conflicts:

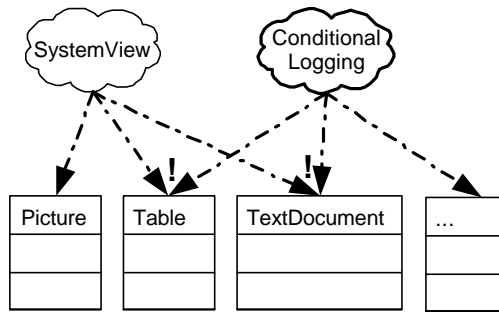
example: Assume that we include an additional aspect to the system, which introduces logging on a selection of the classes in the system; for each object, logging can be turned on or off individually through the methods *startLogging()* and *stopLogging()*, which need to be added to ('introduced' in widely adopted AOSD terminology) the logged classes. This aspect then adds conditionally executed logging code to all selected locations.

The code that causes something to be written in the log can be described as follows in pseudo-code:

```
conditionalLogging:
  if loggingOn()
  then logger.log(<context information>);
```

This pseudocode is assumed to be inserted e.g. before or after individual method bodies. The set of locations is defined by a *crosscut specification*. The following diagram illustrates the new situation with two crosscutting aspects which are superimposed upon overlapping sets of base classes (in the diagram this is shown for classes *Table* and *TextDocument*).

² Many AOP languages offer distinct language constructs for expressing aspects respectively base classes, in some these concepts have been unified.



In principle, using an appropriate aspect-oriented programming language, this extension of the example will work fine. However, problems may occur due to conflicts between the view code and the logging code that are superimposed upon the same base classes: the methods *startLogging()* and *stopLogging()* will not be allowed to execute, due to the (too restricted) constraints imposed by the *SystemView* specification.

The above example illustrates the core motivation for this work: the introduction of new aspects may cause certain modules to cease functioning as expected, due to conflicts, or interference, between the superimposed aspects. This is particularly inconvenient, since the occurrence of the problems may be unpredictable to the responsible developer(s): these aspects can be introduced independently, also at different times, and the programmer of each aspect may be unaware of the existence or future introduction of the other aspect(s).

Also note that the potential conflicts that may occur when combining the aspects, may depend on the adopted model and/or specification style for expressing the aspects; it is perfectly possible to implement the above example in your favorite AOSD language without encountering these problems. However, it is important to realize that there are no general approaches that can avoid all conflicts: it is determined by the requirements of a particular application, whether a certain conflict is appropriate or inappropriate. To illustrate this with the above example: one could imagine that the inability to execute the methods introduced by the *ConditionalLogging* aspect if the *SystemView* aspect is applied, is in fact a correct and desired effect of the *SystemView* aspect.

In the remainder of this section we will continue the problem statement by identifying four general properties that affect the ability to reason about AOSD composition.

2.1 Property 1: obliviousness

In AOSD software, modules are oblivious to superimposition, hence without proper tool support it is not feasible to know all the possible compositions, and reason about the impact of these compositions. One of the issues is that large teams are involved in the development of serious software, which may cause the

superimposition of a certain aspect by one programmer to go unnoticed by the programmers of the base module. This can be the case because the aspect was added after the base module had been written and tested. This is even a bigger issue for programmers of other aspects that happen to be superimposed at the same joinpoint (i.e. location or execution point within a base module).

2.2 Property 2: open-ended crosscuts

The set of locations (within base level modules) that a crosscutting aspect affects is not necessarily known or fixed when writing the aspect. This is due to the ability (depending on the particular crosscutting language) to write open-ended specifications of the *join points* (i.e. the points in the execution of a program that will be enhanced with additional behavior). In the example above, after writing the aspect *SystemView*, additional base level classes may be introduced in the application which belong to the set of system classes, and hence will be subject to superimposition of the *SystemView* aspect.

2.3 Property 3: semantic analyzability of aspect compositions

Limited forms of conflict detection can be done at a syntactic level, or through signature based typing systems. More general conflict detection requires reasoning about the semantics of composed modules (aspects); is a certain composition of behavior semantically correct or conflicting? In general, the ability to reason about the behavior or correctness of programs is very limited, for two reasons mainly:

- Certain characteristics of programs in a general-purpose (Turing equivalent) programming language are impossible to determine through a general algorithm. For example whether the program will actually end (the Halting Problem). As a result, reasoning whether a piece of a program has, or has not, certain semantics is extremely difficult, if possible at all, to realize in a general way.
- To reason about behavior being *correct*, a complete formal specification of the desired behavior (for example in terms of pre- and post-conditions) must be provided by the developers. With the current state of practice, this is rarely the case.

For these reasons, we explicitly do *not* pursue reasoning about correctness or semantic conflicts of *full programs*, nor will we do so about program fragments expressed in general purpose programming languages: instead, we reduce the scope to the composition and interaction of aspects, where the specifications of these aspects are such, that they do allow sufficient reasoning.

2.4 Property 4: Composition of multiple superimposed aspects

In the case where multiple aspects are superimposed upon the same base module, there may be a semantic interference caused by composition of the aspects. In the general case, the composition of aspect specifications matters due to the side effects that they cause (e.g. the

ordering of logging and testing constraints can make a – subtle but visible– difference). Only in those cases where the behavior of one aspect can never interfere with the behavior of another aspect, their composition relation is irrelevant³. Semantic analysis of the behavior of each of the aspects is required to be able to draw conclusions about the allowed compositions.

3. Vision & Requirements

This section is organized as follows: first we define our goals in more detail. Then we define the context and scope more precisely by explaining two key requirements that are needed to achieve our goals. The *composition filters model* is then introduced and selected as a modelling language that meets these requirements. Subsequently we explain how we plan to detect semantic conflicts between aspects.

3.1 Goals

We envision a ‘conflict checker’ tool that will take a complete (aspect-oriented) program as input, and search for semantic conflicts between superimposed aspects, distinguishing at least the following –strongly related– kinds:

- *Error conflict*: conflict that prohibits execution of the program; typical examples are due to dependencies within one aspect that are broken by another aspect, causing the execution to halt or raise an exception.
- *Composition interaction problems*: analyze the possible compositions (including different orderings) that make a (no) semantic difference and offer feedback about alternatives and their consequences.
- *Potential semantic conflict*: depending on the actual application requirements a particular interference between aspects may be a problem or a required feature. We discussed an example of such a conflict when composing the view constraints and the logging functionality in section 1. In such cases, the tool should offer warnings and perhaps concrete suggestions for removing the ambiguity.

Generally, the tool may not be able to find all possible semantic conflicts, but all the conflicts that it detects should be correct.

As should be clear from this discussion, the proposed tool is not intended to be a (type) checker that verifies the adherence to strict rules (only), but more similar to a tool like *lint* [Darwin 88], which analyzes a multitude of properties of C and C++ programs, sometimes resulting in error messages, but mostly presented as warnings.

3.2 Language Requirements

The most crucial requirement that we need to fulfill to achieve our goal successfully is the ability to express the behavior of aspects in a form that supports semantic

analysis. Formulated differently, we need to express the aspects using one or more Domain Specific Languages whose designs are a compromise between expressibility of (domain specific) behavior and analyzability. In the early proposals for aspect-oriented programming [Kiczales 97], the notion of “Aspect Description Languages” was coined, which are typically close to this requirement. Examples are AML [Irwin 97], COOL and RIDL [Lopes 97]. However, since that period of time, all proposed AOP languages, including AspectJ [Kiczales 01] adopt full programming languages (usually Java) for expressing the semantics of aspects.

The second requirement is that we must be able to retrieve the exact definitions of all superimposition relations and interpret these to obtain a complete overview of which aspects are superimposed at which locations (i.e. the *join points*) within the base modules. This means that the specifications of the crosscuts (i.e. sets of join points in the program) are expressed in a language that can be interpreted and analyzed. This requirement is satisfied by most aspect-oriented programming languages such as AspectJ [Kiczales 01] and HyperJ [Ossher 01], because the so-called *aspect weaver* must be able to reason about the specifications statically as well. But for example in fully dynamic languages such as AspectS [Hirschfeld 01], the crosscut must be constructed at run-time by a Smalltalk program, which does not meet our requirement. At this stage, we require that the superimposition relations cannot be modified at run-time, so that we can perform static analysis of the superimposition relations.

4. Adopting the Composition Filters Model

We have chosen the Composition Filters model [Aksit 92] [Bergmans 94, 01] as the input for our analysis process. We will first briefly describe its most relevant characteristics, and summarize how it meets the requirements stated in the previous section.

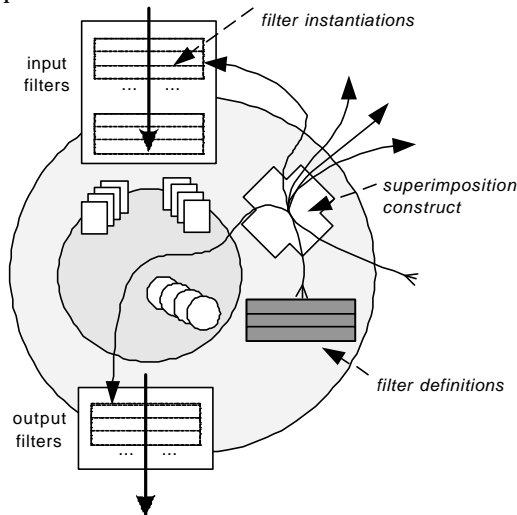
The composition filters (CF) model is a modular extension to the ‘conventional’ object model as adopted e.g. by Java, C++ and Smalltalk. The behavior of an object can be substantially affected and enhanced through the manipulation of incoming and outgoing messages only. To do so, in the CF model, respectively *input filters* and *output filters* can be defined that manipulate messages in a well-defined way. The specification of these filters consists of the declaration of a *filter type* that defines the semantics of the filter, and a simple pattern matching/substitution expression that determines *acceptance* or *rejection* of messages that attempt to pass the filter. The filter type defines the semantics associated with respectively *acceptance* and *rejection* of messages. The pattern matching expressions are used to express application specifics, whereas the filter types encapsulate well-defined domain-specific semantics. The set of filter types is open-ended, some

³ Note that this can never be concluded by a fully automated analysis of the program, since the interference may be due purely to requirements.

commonly used filter types are shown in the following table:

Filter Type	Semantics (possible application)
<i>Dispatch</i>	Message dispatching (conditional, single/multiple inheritance & delegation)
<i>Error</i>	Exception throwing (assertions, multiple views)
<i>Wait</i>	Blocking & queuing messages (synchronization constraints)
<i>Meta</i>	Message reification (abstracting object interactions, user-defined semantics)
<i>RealTime</i>	Affect Real-Time properties (set real-time constraints)

An important part of the composition filter model is the *superimposition* construct: this is a part of a concern (c.f. class) definition that maps the filters (from this object or others) onto a set of instances (possibly including instances of the current class). This means that the filters are added to (superimposed upon) the existing filters. The mapping is expressed as a selection upon the set of objects or classes in a system using OCL predicates. For more details on the superimposition mechanism we refer to [Bergmans 01] [Salinas 01]. The picture below offers an overview of the important elements in the composition filters model as described above:



We assume the Composition Filters model is suitable for meeting the requirements that were set in the previous section because:

- 1) The behavior of the superimposed aspects is defined by one or more filters. The specification of the filters consists of two parts: the pattern matching part can be easily analyzed, the filter types have specific, well-defined semantics, which can serve as input for a reasoning process.
- 2) The superimposition specifications define the crosscuts using OCL predicates over sets of (predictable) objects and classes. These predicates are also well-understood and suitable to analyze and interpret.

5. Outline of Approach

In this section we explain how we plan to achieve the goals that we set in section 2

The semantic analysis requires the following steps:

- 1) All filter types (both predefined and programmer-defined) will be enhanced with semantic annotations that describe the semantics of respectively the accept and reject behavior of a filter.

These semantic annotations will be used to reason about the possible semantic conflicts. Hence, no complete formal semantics of the filter behavior are required, but an abstract model that allows us to detect and reason about overlap, interference and conflicts between multiple filter specifications will suffice:

- 2) The semantic annotations will be expressed in terms of operations offered by a common *Abstract Virtual Machine* (AVM). The AVM offers a set of primitives in which all kinds of semantics can be expressed. The scope of these primitives is determined by the scope of the available and expected filter types. An important part of the AVM is an abstract message model: it is expected that many of the AVM primitives can be expressed as operations on a message model.

It is expected there will be primitives for synchronization & scheduling, real-time scheduling, dispatching, exception throwing/handling, reification of messages, and so forth. Also primitives should be available for manipulating the explicit and implicit arguments of messages.

- 3) The next step is to resolve all crosscut specifications; each crosscut specification applies to a set of join points.
 - We first create 'instance categories' for each of the above sets.
 - Secondly, we distinguish all -partially-overlapping sets, and create new instance categories for each different overlap. The existing instance categories are, at the same time, cleaned by removing all join points in the overlapping subset.
 - Thirdly, we associate all superimposed behavior with the appropriate instance categories; this will yield categories with one or multiple pieces of behavior to be superimposed—at the same joinpoint. These categories are the primary interest of our tools.
- 4) Now for each instance category, given that it has multiple superimposed aspects, a sanity check can be performed, globally as follows:
 - Using the semantic annotations of the filter types, and the specification of the matching expression, translate all filters into AVM instructions.
 - Each AVM instruction is modelled as a set of read and write operations upon a particular resource.

- Using read/write conflict detection techniques, potential semantic conflicts between multiple filter instances can be identified and reported.

6. Summary

In this paper, we have addressed a subcategory of the general analysis of aspect-oriented systems; the composition of multiple aspects (advices) at a single location (joinpoint). We have explained some of the issues that arise in this area in section 2. In particular, we addressed the issue of semantic analysis of interacting aspects. In section 3 we outlined our vision and some of the necessary ingredients of the programming language to achieve that vision. Section 4 explains the composition filters model and motivates its adoption for this research: in particular, composition filters introduce certain abstractions which can be exploited by the proposed approach. Finally, section 5 outlines how we intend to achieve our goals by explaining in detail a set of steps that, when implemented, will yield a tool for analyzing semantic interaction between superimposed aspects.

Finally, we would like to emphasize that this paper does not claim to make detailed technical or scientifically sound contributions, rather it is intended to motivate and explain our approach towards semantic analysis of aspect interaction. Its key contribution is in explaining this approach and stating the necessary constraints on the programming language that is subject to the analysis. Although we have not presented a comparison with the related work, we cautiously suggest that the presented approach is novel, in particular we are not aware of related work that is able to reason about the detailed semantics of aspects

7. References

[Aksit 92] M. Aksit, L. Bergmans & S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, Proc. of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395

[Bergmans 01] L. Bergmans & M. Aksit, *Composing Crosscutting Concerns using Composition Filters*, Communications of the ACM, October 2001/Vol.44 No. 10, pp. 51-57, 2001

[Bergmans 94] L. Bergmans. *Composing Concurrent Objects*, Ph.D. thesis, University of Twente, The Netherlands, 1994

[Bougé 88] L. Bougé and N. Francez. A compositional approach to superimposition. In ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pages 240-249, Jan 1988.

[Darwin 88] Ian F. Darwin, "Checking C Programs with lint", O'Reilly, October 1988

[Filman 01] CACM intro

[Hirschfeld 01] R. Hirschfeld, "AspectS: AOP with Squeak", OOPSLA'01 Workshop on Advanced Separation of Concerns, Tampa FL, 2001

[Irwin 97] J. Irwin, J.M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar & T. Shpeisman, *Aspect-Oriented Programming of Sparse Matrix Code*, In Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Marina del Rey, CA. December 1997. Springer-Verlag LNCS 1343.

[Katz 93] S. Katz. A superimposition control construct for distributed systems. ACM Trans. on Programming Languages and Systems, 15:337-356, April 1993.

[Kiczales 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997

[Kiczales 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold, *An Overview of AspectJ*, Proceedings of ECOOP 2001, LNCS 2072, Springer Verlag, 2001

[Lopes 97] C.L. Lopes, *D: A Language Framework for Distributed Programming*, PhD Thesis, College of Computer Science, Northeastern University. November 1997.

[OCL 03] *Object Constraint Language Specification*, in UML 1.5, chapter 6, OMG document formal/03-03-13, 2003

[Ossher 01] H. Ossher & P. Tarr, *Multi-Dimensional Separation Of Concerns And The Hyperspace Approach*, in M.Aksit (ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001

[Salinas 01] P. Salinas, *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*, EMOOSE MSc. thesis, Vrije Universiteit Brussel, August 2001