Reasoning about Behavioral Conflicts between Aspects

Pascal Durr, Lodewijk Bergmans, 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 the same, or shared, join point. In this paper we explain the problem of behavioral conflicts between aspects at shared join points, illustrated by a real-world example. We present an approach for the detection of behavioral conflicts that 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 that is suitable for automated conflict detection. The approach employs a set of conflict detection rules which can be used to detect both generic conflicts as well as domain- or application-specific conflicts. This approach is general for AOP languages, and its application to both Compose\* and AspectJ is illustrated in this paper. The application to Compose\* demonstrates how the use of a declarative advice language can be exploited for fully automated conflict detection, without the need to annotate the aspect behavior. The approach has been implemented and tested within the Compose\* and CAPE environments.

1 Introduction

Aspect-Oriented Programming (AOP) aims at improving the modularity of software in the presence of crosscutting concerns. AOP languages allow independently programmed aspects to superimpose behavior at the same join point. Unfortunately, such expression power may cause undesired emerging behavior. This is not necessarily due to a wrong implementation of the individual aspects; the composition of independently programmed aspects may cause emerging conflicts due to unexpected behavioral interactions. The most common situation where this occurs is when multiple advices are superimposed at the same join point; we call this a shared join point. Note that interference between aspects may also occur in other places without shared join points, but in this paper we concentrate on this—most relevant—case. We define a behavioral conflict

---

1 We prefer to use the general term superimposition to designate the composition of aspects with a base program, thus distinguishing it from the term weaving, since the latter is commonly associated with implementation techniques that manipulate code to achieve such a composition.
as emerging behavior that conflicts with the originally intended behavior (cf. requirements) of one or more of the involved components. In the context of this paper: when the emerging behavior of the composition of multiple advices at a shared join point conflicts with the intended behavior of the advices.

In component-based programming, each component explicitly composes its behavior from fine-grained actions and the interfaces of other components. For example, behavior is composed as a sequence of function calls, through the specification of inheritance or through aggregation, e.g. of objects. In all these cases, the programmer is responsible to ensure that the specified composition is sensible. In addition, techniques like type checking support the programmer to avoid certain mistakes, e.g. introducing method with the same name.

The reasoning techniques, such as type-systems, that are developed for components cannot be directly applied to aspects at shared join points, because this kind of behavioral composition is implicit: each aspect is defined independently of the others, potentially at different times and by different people. The composition of their advice happens ‘by coincidence’ at shared join points, certainly the programmers of the individual aspects cannot always be aware that this will happen.

Recently, reasoning about the correctness of a system after superimposing multiple aspects at the same or shared join point, as described in [1], has been considered as an important problem to be addressed [2,3,4]. Our approach focuses on behavioral (‘semantic’) conflicts, not conflicts that are syntactic or structural, such as for example changing the inheritance hierarchy while another aspect depends on the original hierarchy.

This paper presents a language-independent technique to detect emerging behavioral conflicts between aspects that share join points. The paper is structured as follows: in section 2 we explain the problem statement through a simple example of a behavioral conflict, based on a system with an Encryption and a Logging aspect. Subsequently, section 3 discusses a number of design considerations for a conflict-detection approach and discusses the related work in the area of conflicts among aspects. Section 4 provides an informal overview of our approach. Section 5 presents an instantiation of the conflict detection model for Composition Filters. The introduced example is revisited in section 6, while section 7 discusses the tool we developed to automatically detect behavioral conflicts. Section 8 provides a brief discussion about an instantiation of our model for AspectJ. Finally, in section 9, we provide some discussion, present possible future work and conclude the paper.

2 Motivation

In this section we present an example application, which serves two purposes: first, it should offer an understanding about the kinds of conflicts we address in this paper. Second, since the example has been identified in the context of a large industrial application, it is intended to motivate the relevance of the problem. The example has been identified within the Ideals project [5]. The Ideals project aims to develop a software design methodology that realizes the structured composition of software from separate modules, while handling system-wide interacting concerns. Examples of such crosscutting concerns are Tracing, Profiling and Contract Enforcement.
In the Ideals project, ASML wafer scanners are taken as a case study and act as drivers for the project. ASML is the world’s leading provider of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or chips. The control software consists of over 15 million lines of C code and is structured into a layered architecture with over 200 components. The complexity of this software is high, and it contains many crosscutting concerns.

The presented example is taken from this large scale embedded application. We present here two aspects that we have identified, namely Parameter Checking and Error Propagation.

### 2.1 Parameter Checking

The Design by Contract approach to software development is based on the principle that the interface between modules of a software system should be bound by precise specifications. These can be pre-conditions, post-conditions and invariants. One such application of design by contract is to check whether the parameters of a function are valid. ASML adopts this in its wafer scanner software to ensure the validity of the parameters. We call this concern Parameter Checking. There are two checks carried out to ensure the contract. Firstly, the function input parameters, that are pointers and are only read, should not be empty (i.e. not null) at the start of a function. If the input parameter pointer is null, it could yield a fatal error whenever this parameter is accessed. The second check verifies that every output parameter, which is also a pointer, is null at the start of a function. An output parameter is a parameter that is written in the function body. If such a parameter points to an already existing memory location, we could accidentally override that data, which is undesired. We present here a simplified version, the actual implementation contains more checks. An example of this concern is shown in listing 1.1.

```c
static int compare_data(
    const DATA_struct* p1,
    const DATA_struct* p2,
    bool* changed_ptr)
{
    int result = OK;

    /* Check preconditions */
    if (p1 == NULL)
    { result = INVALID_INPUT_PARAMETER_ERROR; }
    if (p2 == NULL)
    { result = INVALID_INPUT_PARAMETER_ERROR; }
    if (changed_ptr != NULL)
    { result = INVALID_OUTPUT_PARAMETER_ERROR; }

    // code that compares the structures and sets the changed_ptr boolean accordingly
```

Please note that the example aspects presented here are slightly altered for reasons of confidentiality. However, this does not affect the essence of the examples.
Listing 1.1. Example of the Parameter Checking code

The function compares the two input parameters \( p1 \) and \( p2 \), and sets the \( \text{changed}_\text{ptr} \) boolean output parameter accordingly. At lines 9 to 20, the checks for the input and output parameters is shown. The parameter checking concern accounts for around 7% of the number of statements in the code, although the exact percentage varies among components.

2.2 Error Propagation

The C programming language does not offer a native exception handling mechanism. The typical way to implement exception handling in C, is to use the return value of a function. The function returns 'OK' in case of success and an error number in case of failure. This means that the caller of the function should always get the return value and verify that the value is still OK, if not it should either handle the error or return the error to its caller.

The error propagation thus consists of (a) passing the error state through the error variable and the return value, (b) ensuring that no other actions are performed in an error state, and (c) if an error state is detected, it is logged. Listing 1.2 shows an example of such an exception handling scheme as employed at ASML.

```
static int compare_data(
    const DATA_struct* p1,
    const DATA_struct* p2,
    bool* changed_ptr)
{
    int result = OK;
    if (result == OK) {
        result = example_action1(...);
        if(result != OK) {
            LogError(result);
        }
    }
    return result;
}
```

Listing 1.2. Example of the Error Propagation code

The code in listing 1.2 first(line 6) initializes a variable, “result”, to hold the current error state, this is referred to as the error variable. To determine whether to continue with normal execution, a check is placed which guards the execution(line 8). In this case this might seem useless as the error variable already contains OK, however these are coding guideline templates and as the code evolves such a check might be required if another operation is executed before this. Next, a call to a regular function is done. If an error is detected(lines 11-14), this error is logged. Finally the error variable is returned at line 17.
It is out the scope of this paper to thoroughly elaborate on the alternatives for exception handling. It is however, obvious to see that the integer return value exception handling contributes substantially to the LOC, in some extreme cases even up to 25% of the code, depending on the component. Although the error handling domain can be divided into three main elements: detection, propagation and handling, we only focus here on the propagation part. Detecting and handling of errors are highly context dependent, thus refactoring these into an aspect is hard. Error propagation on the other hand follows a more common pattern which can be refactored into an aspect more easily.

We refactor the concerns above into an AOP solution, and the two concerns above are implemented using some AOP mechanism. Concern ParameterChecking should check the input and out pointer parameters of each function to ensure the contract of the function. We implement this functionality as an advice, called check. Concern ErrorPropagation implements the following elements: check whether we are not in erroneous state, if so execute the original call. If this call yields an error state we should log this. Likewise to the checking concern, we also implement the functionality of the propagation as an advice, called propagate. If we now apply both concerns to a base system, the resulting system is shown in figure[1]

![Diagram](image)

**Fig. 1.** Parameter Checking and Error Propagation example

On the top of the picture the two concerns and their advices are presented, namely: check and propagate. The figure also shows our example C function, compare_data(…). This function with several others form the base system. The arrows show where each advice is applied. In this example, the advices are superimposed on the same join point, in this case compare_data(…). However, as the concerns implement coding conventions, there are many such shared join points. Now assume we would apply propagate before check, the errors detected by check are never propagated to the caller. Furthermore, the check code is always executed, even if in an error state. If we examine the conflict more carefully we see that the conflict is caused by a dependency between the two advices. The propagate advice reads the error variable to determine the current error state and can subsequently write and log if an error is detected. Advice check verifies that the arguments are valid, and possibly sets the error variable. In this case the
presence of the conflict depends on a specific ordering of advice, later we will present some examples where the ordering does not matter.

2.3 Discussion

Now let us elaborate more on the concerns and the conflict between them. Individually, both aspects are consistent with their requirements and therefore they can be considered sound. From the language compiler point of view, the program with either orderings of advices can be considered as a valid program with no errors, there are no syntactical or structural problems. However, once these aspects are applied at the same join point, an emerging conflict situation appears. Such a behavioral conflict may lead to undesired behavior.

In this particular case, if one is aware of such (potentially) conflicting cases, one can enforce an ordering. For example, it is possible to enforce an ordering in AspectJ with the declare precedence construct. However the ordering constraint models in AOP languages may not be sufficient to specify the intended ordering. In the example we showed that the error propagate advice should be applied after the check advice. In practice, however, detecting emerging conflicts may not be that easy, especially if the conflicting aspects crosscut the entire base application and share many join points. The conflicts are especially hard to detect when the conflicting situation emerges only at specific shared join points. This implies that the conflict presents itself given a specific context, the dependency between the aspects is not directly but rather indirectly, through some properties of the join point. It is therefore necessary to develop techniques and tools that reason about the (potential) behavioral conflicts between aspects.

2.4 Other Conflict Examples

To further exemplify behavioral conflicts, and demonstrate that these may occur in many different domains and applications, we show several other examples. We assume that all these aspects apply to same join point.

Authorization and Persistence : An authorization aspect checks whether an “action” is allowed to occur for the current user. A persistence aspect will write data to some persistent structure. Here a conflict situation arises if we first write the required data to the persistent structure and afterward check whether the user was allowed to carry out this action.

Authorization and Authentication : An authentication aspect verifies that the user is actually who he says he is. Here a conflicting situation arises if the authentication aspect is executed after the authorization aspect, as before authentication it is not possible to authorize.

Compressing and Logging : A data compressing aspect can decrease the traffic on a certain communications protocol. If we compose this aspect with an aspect which logs all traffic, we have to ensure that the logging advice is executed before compressing the data.

Encryption and Tracing : A similar situation to the previous one can be stated for encryption and tracing aspects. However, in this situation the correct order of the advices depends on application specific requirements. If the application resides in
a hostile environment, we would like to ensure that all data is encrypted before any other aspect reads this data, to ensure the safety of the data. However, if the application operates in a safe environment and we would like to verify the data, we would like to see the plain text data and not the encrypted data. This conflict is an example of an application specific conflict, or even deployment context specific. In these cases, the correct ordering of advice is not trivial to determine. See [7] for a more detailed discussion about this conflict.

**Database modification** : Imagine two aspects which both write the same field or table of a database, we assume that this is a conflict as the value written by one aspect is overwritten by another aspect. In this there is no (in)valid ordering, either way a value is overwritten. If one aspect would do an update, first a read and then a write, than there would be valid ordering possible.

**Real time constraints and Concurrency** : In this case we assume that a real time aspect enforces some timing constraints on a certain action. If we now also apply an aspect which synchronizes this action with another action, we can never ensure that the deadline is met, because of blocking. This problem is again a conflict that can not be solved by reordering the aspects, unlike the first four example conflicts.

### 3 Background and Related work on AOP Conflicts

The previous section presented several examples of behavioral conflicts. We now present an overview of possible AOP conflict and discuss the related work in this area.

#### 3.1 An overview of AOP conflicts

In this section we discuss—a part of—the problem space of composition conflicts caused by the use of aspects, and position our approach into this space. We discuss the problem space along the following dimensions:

1. Type of superimposition (structural or behavioral)
2. Type of interaction (control-flow based or state-based)
3. Composition type (advice-base or advice-advice)
4. Type of Join Point (shared or distinct)
5. Order dependency (order-dependent or order-independent)
6. Domain of conflict (generic, domain-specific or application-specific)
7. Advice specification form (imperative/declarative, Turing-completeness)

Note that these dimensions are not fully orthogonal, but in the presented order these allow us to scope the context and to indicate the category of problems this paper is focusing on.

**Type of Superimposition** In general, a distinction is made between behavioral and structural superimposition: behavioral superimposition refers to the adornment of a program (at join points) with behavior (expressed as advice). Structural superimposition refers to various -disciplined- forms of transformation of the program, typically through the addition of program elements such as methods and fields. This is also called
introductions or inter-type declarations. Note that one aspect module may consist of a combination of behavioral and structural superimposition.

In this paper, we focus on the possible behavioral conflicts between composed pieces of behavior. Although this applies in principle to any form of behavior composition, to restrict the scope of the discussion we focus on behavioral superimposition, which is also the most common of the two.

Type of interaction Conflicts between aspects are caused by some form of interaction. In general, we can distinguish between control-flow based interaction and state based interaction. Control-flow based interaction means that the composition affects the control-flow of program. State-based interaction means that the composition brings the program into a state that would not occur (at that location) in the program without that particular composition. It should be stressed that an interaction is not necessarily bad, or should be considered a conflict; many interactions are in fact desirable, and compositions may be specified exactly to achieve such an effect on the control flow or state. One of the goals of this paper is to present a means to distinguish among desired interactions and undesired interactions (i.e. conflicts).

In [27], Leavens and Clifton propose a classification of advice based on whether or not the advice changes the complete specification that must be satisfied at runtime. Observers do not alter this effective specification, whereas Assistants do.

In [8,9], Katz proposes three categories of aspects: spectative, regulative and invasive. Spectative aspects do not influence the underlying system, they only query the state of fields. Regulative aspects can alter the control flow of the underlying system. Finally, invasive aspects both alter the control flow and the fields of the underlying system. In [8], they further make a distinction between weakly invasive and strongly invasive. The weakly invasive advice always returns a state of the base system, whereas this is not the case for strongly invasive advice.

Rinard et.al. [36] have proposed a classification system for the possible effects of advices on base code. It also distinguishes the dimension of control flow and the dimension of state. For the dimension of control flow they distinguish the following categories: Augmentation, Narrowing, Replacement and Combination. For the dimension of state they define they following types of interaction: Orthogonal, Independent, Observation, Actuation and Interference.

Composition type In the case of behavioral superimposition, we can distinguish at least two types of composition:

– Advice-base composition: the composition of the advice behavior with the behavior of the base system; this determines the actual impact of applying the aspect to the program.
– Advice-advice composition: the composition of the advice behavior with one or more other advice behaviors; this happens when several advices are superimposed at locations where their behavior ‘joins’.

Although advice-base composition is a very relevant category, and in principle our approach could be applied here as well, we will not further address it in this paper;
the main motivation for this is that automated reasoning about the semantics of fully-
fledged base code is notoriously difficult. We consider the annotation of a complete
base program with clues about its behavior to be impractical, in all but very special
situations and hence not very interesting. As we will explain later in this paper, for
(declarative) advice code we believe it is much more feasible to obtain a specification
of its semantics.

**Type of Join Point** We mentioned that advice-advice composition may yield conflicts
when the advices ‘collide’; this only occurs when there is some direct influence or
interaction between advice. This naturally depends on the relative location of the join
points where the respective advices are superimposed; these join points can be either:

- *shared join points*: when multiple advices are applied to the same join point. It is
  obvious that they may influence each other in this case.
- *distinct join points*: when advices are applied at distinct join points, they may still
  influence each other, albeit less likely. For example, assume that one advice affects
  a state that another advice depends on (or changes as well). If the second advice
  is in the control flow of the first advice, the changed state may very well affect the
  second advice.

In this paper we focus on advice-advice composition at shared join points. The primary
reason for this restriction is that detecting the mutual influence of advices at distinct
join points requires extensive analysis of the base code of programs, something that is
not the primary aim of this work (but our approach could -as far as we can envision- be
applied as well to aspect interaction at distinct join points when such an analysis would
be available).

Douence, Fradet and Sudholt[10] state that two aspects do not interact if they are
independent of each other. They continue to state two forms of independence. The first,
*String Independence*, occurs when the crosscut specification never matches for any base
system, whereas in the second form, the independence of the aspects is relative to a
given base system.

**Order Dependency** Conflicts caused by advice-advice composition can be divided
into two categories:

- *order-dependent*: when the conflict only occurs in a specific ordering of advices
  (and thus not in the reverse ordering).
- *order-independent*: when the conflict is independent of the ordering of the advices.

Since, for advice applied at distinct join points—at least for imperative languages— the
ordering cannot be changed without changing the pointcuts or changing the execution
flow in case of conflicts as a result of side-effects, this distinction has only practical
relevance for advice applied at shared join points. Many AOP languages have sup-
port for ordering advice at shared join points, such as aspect precedence declaration in
AspectJ[40] and declarative ordering constraints in Compose*[1].

**Domain of conflict** It is important to understand that most behavioral conflicts are not
cases where the execution of composed advice leads directly to execution problems,
or is otherwise fundamentally wrong or impossible. Typically, behavioral conflicts are
detected through the use of domain knowledge; the advice composition is problematic
because in a specific domain or application context, such a composition does not make
sense. For example, it may lead to deadlock, incorrect scheduling, invalid application
data, advice that is accidently not executed at all, and so forth. We distinguish the fol-
lowing categories:

- **Generic**: conflicts that are general to computing and program execution, and hence
  may occur in any type of program. For example, when two advices write the same
  variable consequently, the value that was written by the first advice may be lost,
  which means the intended behavior of the first advice is not represented in the
  composed application. This conflicts is independent of the behavior of the advice
  and the meaning of the values.

- **Domain-specific**: conflicts that only occur in a specific domain, such as concurre-
cy, persistence, exception handling or security. Each of these domains imposes
  specific rules about correct compositions. For example, the combination of two
  advices may cause a wrong synchronization, which can e.g. lead to deadlock or
  indirectly to corrupt data.

- **Application-specific**: conflicts that only arise due to specific constraints that can be
  traced back to the requirements of the application. For example, when an encryption
  advice is composed with a logging advice, it depends on the application require-
  ments whether the encrypted or the unencrypted information should be logged.

The work in this paper is not restricted to any of the above categories; we will show that
the presented approach is applicable to all three categories. However, for the domain-
specific and especially the application-specific conflicts, the programmer may need to
provide specific information, e.g. in the form of annotations, to be able to identify the
related behavioral conflicts.

**Advice specification form** Finally, we briefly discuss the various forms of advice spec-
icification, and their relationships with advice composition conflicts and the detection of
such conflicts. First, we can distinguish between advices that are specified in an impera-
tive manner and those that are specified in a declarative manner. The first group
is the common approach in most AOP languages, such as AspectJ, where advice is ex-
pressed just like a method body in the base language. The latter group is less common,
and most examples appear in domain specific aspect languages, such as the COOL and
RIDLE [35] aspect languages for distributed programming. There are definite advan-
tages for programmers to be able to express advice in the base language they are fa-
miliar with, and interfacing (such as sharing data structures) between the base language
and the advice languages is also straightforward in such cases. However, a declarative
advice specification may (though not necessarily) avoid certain categories of conflicts,
and in general it is easier to analyze it with the purpose of detecting conflicts.

A second categorization, that can be made, relates to the expressiveness of the ad-
vise language; most advice languages are expressed in an (imperative) Turing-complete
language. Some, usually domain-specific, advice languages however, have restricted ex-
pressiveness and are not Turning-complete. This is a trade-off for the language designer
between expressiveness, and ease of understanding for both programmers and machines (such as conflict detection tools).

This paper is not restricted to any advice specification form. However, we will demonstrate that our approach can leverage the characteristics of a declarative, non Turing-complete, advice language, to allow for better automated conflict detection.

3.2 Discussion of related work

The previous section and the presented taxonomy of aspect conflicts, showed which conflicts we are focusing on in this paper. In short, we want to detect behavioral conflicts between aspects at shared join points. There are several different approaches which verify programs in the presence of aspects.

One approach of program verifiers utilizes traditional model checking techniques. Krishnamurthi et. al. propose one such approach in [34]. The paper consider the base program and aspects separately. The paper states 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 [32], 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 single aspect and pointcut designator support, and thus can only detect base-aspect conflicts, and not conflicts between aspects at shared join points.

Program slicing is another approach to detect state based interactions among aspects. In [11] Balzarotti et. al. present an approach to slice AspectJ woven code. First a slice of the woven program is created, given one aspect, and subsequently the slice of the woven program, using another aspect. If the slices intersect, then the aspects interact. The approach is not only capable of detection interaction between aspects, but also between aspects and a base program. It supports determining conflicts due to side effects of advice. However, the approach is unable to determine whether an interaction is desired or undesired, thus it does not provide interference detection.

Another aspect verification approach is base on graph transformations. In [37], Staijen and Rensink model, part of, the Composition Filters behavior as a 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 filtermodules on this resulting state space. If the different orderings of the filtermodules result in different state spaces, the program is considered to be interacting. This approach also detects aspect-aspect conflicts, but does not provide a way to model conflicts at the appropriate level of abstraction, which is what we require.

In several papers (e.g. [29] and [38]), Södholt 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 considers the presence of a
shared join point to be an interaction.

In [12], Pawlak, Duchien and Seinturier present a language called CompAr, which
allows the programmer to specify a set of execution constraints of the advice. It also
provides an abstraction of the implementation language. This technique also analyzes
interactions of aspects at shared join points. The CompAr compiler verifies whether
the execution constraints hold for that given abstract specification. The work focuses
on determining the correct order of composition given the execution constraints. The
current approach does not take the ordering of the constraints into account. This order
is most of the times vital for correct conflict detection.

Some conflicts among aspects are caused by interactions among aspects during
weaving. This category of problems has been presented in [13] and [14]. Havinga et.al.
[13] explain the problem that interdependent (interacting) introductions may cause
ambiguities during weaving. They present an approach to achieve a correct weaving whenever
possible and signal a conflict if not. Kniesel and Bardey [14] introduce the notion of weaving interactions in a more general sense. They present techniques for detecting
weaving interactions, for automatically resolving a large class of interactions, and for
giving precise diagnostics about conflicts and interactions that cannot be resolved automatically. Their approach is implemented in the Condor tool [15]. Mens, Kniesel and
Runge [16] compare Condor to a detection tool based on graph rewriting and show that
Condor has only slight advantages with respect to its functionality but is several orders
of magnitude more efficient. In this paper we focus on the behavioral interactions of
advice at run-time, rather than the interactions during weaving.

3.3 Contributions of this paper

The contributions of this paper are as follows: the paper explains the problem of behavioral
conflicts with an example from an industrial context, accompanied by additional
examples to illustrate the generality of the problem. The paper examines the space of behav-
ioral conflicts in the context of AOSD (and selects a focus for this paper within this
space: behavioral conflicts between advices at shared join points). The paper presents
an approach to the detection of behavioral conflicts that is generally applicable for most,
if not all, AOP languages. This is illustrated by concrete examples of how to apply the
approach to both composition filters (as implemented in the Compose* language and
tools) and AspectJ.

The key idea of the approach is to introduce an abstraction of program behavior in
terms of resources and resource operations. Such an abstraction has several advantages:
  – It allows for expressing behavior (and conflicts) without dealing with (needless)
    implementation-level details.
  – It allows to express not only generic, or universal, conflicts, but also domain- and
    application-specific ones.
  – It strongly reduces the computational complexity of conflict detection analysis. The
    approach is scalable to large applications; the analysis requires polynomial time for
each join point (this is repeated linear with the number of join points).
4 Approach

To reason about the behavior of advices and detect behavioral conflicts between them, we need to introduce a formalization that enables us to express behavior and conflict detection rules. Clearly, a formalization of the complete behavior of advice in general would be too complicated to reason with. Therefore, an appropriate abstraction must be designed that can both represent the essential behavior of advice, and be used to detect behavioral conflicts between advices.

Our approach is based on a resource operation model, to represent the relevant semantics of advice, and detect conflicts. We have chosen to adopt a resource model, since this is an easy to use model that can represent both concrete, low-level, behavior as well as abstract high-level behavior. Our approach of conflict detection resembles the Bernstein conditions\[17\] for stating concurrency requirements. A similar approach is also used for detecting and resolving (concurrency) conflicts in transaction systems, such as databases\[18\].

A conflict can only occur if there is an interaction, which can always be modeled as one or more shared resources. A conflict can then be modeled as the occurrence of a certain pattern of operations on a shared resource. In this section we will explain the model intuitively, based on the previously presented example. \[19\] offers a formal description of the resource model. To summarize: our behavioral specification language is expressed in terms of operations on resources, which is attached to an advice.

Figure 2 presents the semantic analysis process and the relationships to the base system and advice. We use this image as a guideline through section 4.1 and 4.3.

4.1 Pointcut designator analysis

At the top of figure 2, one can see a set of aspects (Aspect1...AspectN). These aspects contain Advices(Adv) and Pointcut Designators (PCD). There is also a base system with a set of classes (ClassA...ClassZ). The aspects and base system specifications are inputs of the Pointcut Designator Analysis phase. During this phase all PCDs are evaluated with respect to the base program. This results in a set of join points with advice(s) superimposed on them.

4.2 Abstraction

After the PCD analysis phase, we retain a sequence of advices per shared join point\[3\]. This sequence is used in the next phase, the Advice Behavior Abstraction phase. The other input for this phase is the resource operation specification. During the abstraction phase, the sequence of advices is transformed into a sequence of resource-operation tuples per shared join point. We now discuss the notion of Resources and Operations and show instantiations of these notions for the running example.

\[3\] In the case that the ordering is only partially known, we iterate over the Advice Behavior Abstraction and Conflict Detection phase for each valid ordering.
Resources A resource is an Abstract Data Type\citep{20}: its identity is determined by the operations that can be carried out on it. A resource can represent either a concrete property of the system, such as a field or the arguments of a method, or an abstract property which may, or may not have a one to one mapping with elements in the system. Such elements can be domain specific or even application specific elements. One such resource is the error variable in the running example. Advice check checks the arguments and alters this error variable, if it detects a bad input or output parameter. As explained in the motivation section\citep{2}, propagate has to ensure that if an error is set, it is logged and should be propagated to the caller. There is thus a clear dependency between these advices, w.r.t. the error variable. We capture this dependency through multiple operations on a shared resource, called errorvariable.

Operations As explained previously, both check and propagate, access the errorvariable. Check reads the arguments and possibly writes errorvariable and propagate reads\citep{line 8 of listing 1.2}, writes\citep{line 10 of listing 1.2} and reads\citep{line 13 of listing 1.2} this variable to determine whether an error has occurred. We model these as read and write operations on the errorvariable resource.
Although the most primitive actions on shared resources are basically read and write operations, if desired by the programmer, we allow such actions to be modeled at a higher level of abstraction. These more abstract operations can be derived from a specific domain, e.g. the “P” and “V” operations on a semaphore, or can even be application specific. For example, in [7], we discuss an example conflict between Logging and Encryption, there we modeled both encryption and decryption advices as respectively encrypt and decrypt operations instead of read-write operations.

4.3 Conflict detection

The operation sequences per resource per shared join point are, in combination with the conflict detection rules, the inputs for the Conflict Detection phase. The phase passes a verdict, i.e. conflict present or not, for each shared join point and for each combined sequence of operation per resource.

Conflict detection rules A conflict detection rule is a constraint on possible the operation on a resource, which is specified as a matching expression on the sequences of operations per resource. This rule can either be an assertion pattern, a combination of operations that must occur on a resource, or as a conflict pattern, a combination of operations that are not allowed.

In the example used in this paper, the conflicting situation is specified as: “if the sequence of operations of resource errorvariable ends with a write or has two consecutive writes, then it is considered as a conflict”. The conflict detection rules can be expressed in any suitable matching language, such as temporal logic, regular expressions or predicate based. For instance, we can formulate these two requirements, on the errorvariable resource, as the following conflict rule: \((\text{if \ ends \ with \ write}||\text{write \ followed \ by \ write})\).

Conflict analysis For each shared join point, there is a sequence of operations on the resource errorvariable. Now assume that operations read, write and read (caused by the error propagation concern) are carried out on the errorvariable resource at a shared join point. And that subsequently a write operation (caused by the parameter checking concern) occurs on that errorvariable resource. This would match the conflict detection rule: \((\text{if \ ends \ with \ write}||\text{write \ followed \ by \ write})\), in which case the verdict of the conflict detection process is: “conflict”. In case of detecting an error, several actions can be carried out, such as reporting the conflict to the programmer.

5 Composition Filters

In this section we explain the essential principles of composition filters, and discuss some of its characteristics that are relevant in the context of this paper. The composition filters model has evolved from the first (published) version of the Sina language in the second half of the 1980s [21], to a version that supports language independent composition of cross-cutting concerns [22,26]. In this paper we use the language Compose* as
an implementation of the composition filters model. A key design goal of the composition filters model has always been to contribute to object-oriented and aspect-oriented programming languages by improving their composability.

The composition filters model aims at offering abstractions for software units at the ‘right’ abstraction level; this means an abstraction level that is sufficiently expressive to address a wide range of requirements, while at the same time attending to certain software engineering characteristics such as adaptability, robustness, comprehensibility, and (especially) composability. In this paper, we intend to demonstrate that a careful design of aspect-oriented language abstractions can help to support automated reasoning about the correctness of a program. In this case with respect to the detection of certain—potentially—conflicting specifications.

5.1 The Parameter Checking Concern in Compose *

We explain the composition filters model with the use of the example of the parameter checking concern presented earlier in this paper. Listing 1.3 shows the implementation of the parameter checking concern in Compose*. In Compose*, the basic abstraction is a concern; this is a generalization of both (object-oriented) classes and aspects. In this example, the concern ParameterChecking corresponds to an aspect that implements a crosscutting concern, i.e. the contract enforcement for the parameters of a function. In the context of the composition filters model, we adopt the term superimposition to denote the—potentially crosscutting—composition of “aspect” program units with ‘base’ program units, since we believe it is more neutral and has less implementation-connotations than terms such as ‘weaving’.

In general, composition filters concerns consist of three main parts, all of them optional:

filtermodule : the unit of superimposition, corresponds to the general notion of advice. A filtermodule defines certain behavior that is to be superimposed at specific locations in the program. In the ParameterChecking example, there is one filtermodule, called check. We discuss the filtermodule specification in some more detail shortly.

superimposition : this is the specification of the actual crosscutting, defining the pointcuts (called selectors), and what behavior (expressed by filtermodules) is to be superimposed at the selected join points. In the example, the filtermodule check is superimposed at the locations indicated by the selector sel that designates all relevant classes, in this case the class with the name CC.CX.FS. The superimposition will later be discussed in more detail.

implementation : the implementation part contains the definition of the object behavior of a concern: this can be expressed in an arbitrary object-based language (assuming it is supported by the implementation): the Compose* language can integrate with any object-based implementation language. In the example, there is no implementation part.

Listing 1.3

```plaintext
concern ParameterChecking

filtermodule check

superimposition

implementation
```

Note that we use the terms aspect and base only as relative roles of program units, not fixed, since we assume a symmetrical model of AOP.
filtermodule check
{
  internals
  checker : ParameterChecker;
  conditions
  inputwrong : checker.inputParametersAreInvalid();
  outputwrong : checker.outputParametersAreInvalid();
  inputfilters
  paramcheckfilter : ParameterChecking = {
    inputwrong || outputwrong => [*.compare_data] *.* }
}

superimposition
{
  selectors
  sel = {Class | isClassWithName(Class, 'CC.CX.FS')};
  filtermodules
  sel <- check;
}

Listing 1.3. Source of the ParameterChecking concern

In listing 1.3 after the declaration of the concern ParameterChecking (line 1), a filtermodule, check (lines 3-13) is defined. This filtermodule consists, in this case, of an internal declaration, two conditions and one input filter declaration. An input filter intercepts, inspects and—possibly—manipulates messages that are received by the instance where the filter has been superimposed. Line 6 defines an internal, called checker, this checker is of type ParameterChecker and is instantiated for each class where this filtermodule is superimposed. Subsequently, two conditions are defined, inputwrong and outputwrong. Inputwrong checks whether any of the input parameters is not null, in case one of the input parameters is null, the condition yields a truth value. For outputwrong, a similar story holds, only now the output parameters are checked. If any of the output parameters is not null, the condition yields a true value. These conditions are used in the (input) filter declared at lines 11-12. Line 11 defines a filter with identity paramcheckfilter and filter type ParameterChecking. The expression in curly brackets at the right hand side of the filter, after the ‘=’ symbol, is the filter pattern. The filter type encapsulates the behavior of the advice, whereas the filter pattern can be considered as the parameterization of the behavior.

An implementation of composition filters comes with a number of predefined filter types; section 5.3 discusses the common filter types in more detail. These filter types can be generic and domain or application specific. The filter type ParameterChecking exemplifies a more generic functionality. We could have chosen to use a more generic Error filter here. However, if we examine the semantics of the Error filter more closely we see that the Error filter halts the execution of the remaining filters, as this is not similar to the intended behavior of our concern ParameterChecking, we chose to implement it as a separate filter type. Filter types can be considered as parameterized by a filter pattern; depending on the result of matching the incoming message against the filter pattern, the filter type defines what action to perform in the case this matching succeeds resp. fails.

Now let us examine the elements of a filter more closely. To illustrate the elements of a filter, we use the filter presented in listing 1.3.
**Name**: This is the identity of the filter, it can be used to refer to a specific filter, and must be unique for a given filter set.

**Type**: The type of the filter determines its semantics of the filter. An filter can either accept or reject, which action is carried out is determined by the filter type, for more detail we refer to section [5.3](#).

**Condition**: Refers to a boolean expression that represents the state (of the instance where the filter is superimposed). This state must be available on the interface of that instance, usually implemented by methods of that instance. In our example (lines 7-9), the conditions `inputwrong` and `outputwrong` are used. `Inputwrong` yields a true value in case any the input parameters are null. We have omitted the exact implementation of the `inputParametersAreInvalid`, this boolean method iterates over all the input parameters, if any of these input parameters are null it will return true. `Condition outputwrong` checks whether any of the output parameters are not null. In this case we use a logic or operator to compose the two conditions to achieve our desired condition.

**Enable operator ‘=>’**: this indicates that, if the condition expression on the left hand side evaluates to `true`, all the messages that can match at the right hand side of this operator can be accepted by the filter element. Compose* also supports an disable operator, ‘˜>’, which accepts all messages except those stated in the matching part.

**Matching part (between '[' and ']’)**: the incoming message is matched with this expression, which consist of a `target` part on the left side of the dot; if the target object of the message matches with the one specified (by identity). On the right side of the dot is the `selector` part, which tries to match the selector of the message (cf. name of the method that is called) with the specified selector (again by name). In the example we use “compare_data” as a selector. For both the target and the selector, a wildcard ‘*’ can be used instead of an identifier. Compose* also offers signature based matching, using ‘<’ and ‘>’, this means there is only a match if defined selector is in the signature of the target object. In the example, we only use name based matching.

**Substitution part**: The substitution part specifies a replacement value for the target and selector, if both condition part and matching part yield a `true` value. Again a wildcard can be used, which indicates that no replacement is to be made for that specific element.

Filter elements can be composed using a composition operator. We consider only the ‘,’ operator, which corresponds to a conditional-OR; if the first element matches, the whole filter matches, if not, the next filter element is tried to match. If the match—eventually—succeeds, the message is said to be accepted, if not it is said to be rejected. The filter type defines the behavior to be performed in either the case of an accept or a reject.

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramcheckfilter</td>
<td>ParameterChecking</td>
<td>{inputwrong}</td>
</tr>
<tr>
<td></td>
<td>Matching</td>
<td>Matching</td>
</tr>
<tr>
<td></td>
<td>Substitution</td>
<td>Substitution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>match</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target</td>
</tr>
<tr>
<td></td>
<td></td>
<td>selector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compare_data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>
The behavior of the filter module `check` is then as follows: for each object where this filter module is superimposed, internal `checker` is instantiated. For every incoming message, it first verifies whether the conditions `inputwrong` and `outputwrong` are true. If both conditions are false, the filter will reject and will hand off the message to the subsequent filter, if any. In case either of the two condition are true, the filter tries to match the selector of that message to `compare_data`. In the example the target of the message is ignored. If the selector matches, no substitution has to take place in this case, due to the wildcards, and the message is accepted by the filter as a whole. This triggers the accept action of the `ParameterChecking` filter type, this accept action reports the erroneous situation and writes the `errorvariable`.

The superimposition part of the concern definition, in line 15-21 of listing 1.3 starts with a section labeled `selectors`. A selector corresponds to a pointcut designator; it selects a number of join points within the program. This is expressed using Prolog predicates, making use of a number of primitive predicates that express properties of the program. For example, line 18 defines a selector with identifier `sel`; its intention is to select all relevant classes. This concern implements a coding guideline, as such we would select all classes in the system. We mention here the specific class “CC.CX.FS” by name, in order to demonstrate the selection mechanism and to have a single concrete shared join point. The first part of the selector defines the unbound variable that refers to all the join points; in this case `Class`. The selector consists of a predicate, `isClass-WithName` which narrows down the possible values for `Class` to those classes with the name “CC.CX.FS”. After the declaration of the selectors, the superimposition of filtermodules is defined by associating selectors and filtermodules; a filtermodule is then superimposed on each join point identified by the selector.

As a second aspect, we show how the error propagation concern can be constructed, this is shown in listing 1.4. The concern `ErrorPropagation` defines one filtermodule `propagate`, which consists of an input filter named `errorpropagationfilter` of type `ErrorPropagation`. The filter, defined on line 6, matches all messages, and thus will always execute the accept action of the filter. The accept action ensures that all calls are only executed in a non erroneous state and that if an error is detected that it will be logged and properly propagated.

```plaintext
concern ErrorPropagation
{
    filtermodule propagate
    {
        inputfilters
        errorpropagationfilter : ErrorPropagation = { [*] }
    }
    superimposition
    {
        selectors
        sel = (Class | isClass(Class));
        filtermodules
        sel <- propagate;
    }
}
```

**Listing 1.4.** Source of the `ErrorPropagation` concern

The filtermodule `propagate` is to be superimposed on all classes in the system, see lines 12 and 14.
5.2 Reasoning about filters

There are a number of characteristics of the composition filters approach that support reasoning about the meaning of a program with filters:
- Filter types encapsulate domain or application knowledge for which it is usually possible to provide a specification, e.g. in terms of the resource model, of the accept respectively reject behavior of a filter type.
- The possible filter behavior is restricted, as they can only accept or reject but not both at the same time.
- The filter pattern is a very limited, declarative, message matching and substitution language, which can be analyzed statically.
- The selector expressions are expressed in Prolog, which is Turing complete, but which need to be evaluated at compile-time only, after which they can be resolved to a fixed set of join points. This means that, as long as we do not support dynamic weaving capabilities, all superimpositions can be predicted and analyzed statically. This yields a fixed set of filter modules for each concern.
- The ordering of filters within a filter module is defined by the declaration order.
- Optional (partial) ordering specifications can be provided by the programmer to determine the ordering constraints among filter modules at the same join point. If the full order is still not fixed, the compiler picks one (arbitrarily), resulting in one specific ordering to be used in the compilation process. From this we can derive a specific sequence of filters for each concern.

In summary, by restricting the expressiveness of the aspect language at several points, and given the specification of the—relevant—behavior of filter types, reasoning about the (composition of) behavior of aspects becomes feasible.

5.3 Filter types

Compose* has several predefined filter types and supports user-defined filter types. In this section we first show some of these filter types. Next, we present how these filter types are mapped to accept and reject actions. Finally, we map these actions to specific operations on resources. For more detailed information about these and other filter types we refer to [26] and [19].

**Dispatch** : The dispatch filter can be used for delegation or simulating (multiple) inheritance. If the filter accepts, it performs a dispatch action, else the next filter is evaluated, a continue action.

**Error** : The error filter can be used for assertion specifications, e.g. pre and post conditions. If the error filter accepts it performs a continue action, if it rejects it reports an error(throws an exception).

**Meta** : The meta filter enables the user to create user-defined advice. If the filter accepts, the message is reified and passed to an advice as an argument, this is referred to as a meta action. In this advice the user can introspect or manipulate the properties of the message and the message execution. This method is called an Advice Type(ACT). Rejection of the filter results in a continue action.

**Substitute** : The substitution filter allows the user to explicitly change certain properties of the message, without using a meta filter. If the filter accepts the specified substitutions are carried out. If the filter rejects it continues to the next filters.
ErrorPropagation: If the error propagation filter accepts, it will execute the propagate action, meaning that any encountered error will be propagated to the caller of the function. In case the filter rejects, the next filter is evaluated, the continue action.

ParameterChecking: In case of acceptance, action check is executed. This action will report the erroneous situation. The message will be passed to the subsequent filter, if the filter rejects.

We now translate the above stated filter actions to our resource model. We only show the operations on the resources: selector, target, arguments, errorvariable and on each condition(<condition>). It is important to realize that the condition, matching and substitution parts are specific to a concrete filter. Some operations are filter instance specific, e.g. whether the target is changed depends on a specific filter instance, only for the propagate and check action we are sure that the error variable will be read or written. Table 1 provides a mapping from the above described actions to operations on resources. The question mark indicates the possibility of operations, as stated, this depends on the filter instance.

Table 1. Filter action mapped to the resource model

<table>
<thead>
<tr>
<th>Filter action</th>
<th>Resources: selector</th>
<th>target</th>
<th>arguments</th>
<th>inputwrong</th>
<th>outputwrong</th>
<th>errorvariable</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>r</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matching</td>
<td>r?</td>
<td>r?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>substitution</td>
<td>w?</td>
<td>w?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>w?</td>
<td>w?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dispatch</td>
<td>w?</td>
<td>w?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta</td>
<td>w?</td>
<td>w?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>substitute</td>
<td>w?</td>
<td>w?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>propagate</td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>w</td>
<td>r</td>
</tr>
<tr>
<td>check</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>w</td>
</tr>
</tbody>
</table>

6 Example revisited

Not that we have defined the mapping between filter actions to the resource-operation model, we show how the two concerns described in listings 1.3 and 1.4 are used to detect the example conflict between these concerns. After resolving the superimposition of both concerns we can identify shared join points: in this case only one: the class CC.CX.FS. At this join point two filtermodules are superimposed: propagate and check. These filtermodules have to be composed in some order. Any order can be chosen, however we take the following order, as this order exposes the conflict we are interested in: propagate and check. After composing these filtermodules, we obtain the following sequence of filters:
Before we can transform the above stated filter expressions to our conflict detection model, we first elaborate on the second filter as it can provide more accuracy. We exploit this additional detail to reduce the number of false positives. We present the evaluation of the matching expression, $\text{inputwrong} \|| \text{outputwrong} => [*\cdot \text{compare\_data}] \cdot \cdot \cdot$ using a finite state automaton. The automaton for our example expression is presented in figure 3.

When the automaton is in the start state it can do one of two transitions, either the condition $\text{inputwrong}$ is true or false. In case the condition is true, it will try to match the selector of the current message to the one defined in the matching expression: $\text{compare\_data}$. If the match is successful the filter will accept the message, and reject otherwise, and the automaton will transition to the corresponding final state. A similar path can be taken for when $\text{inputwrong}$ yields false, and $\text{outputwrong}$ yields true. As figure 3 shows, there are several ways for the filter to either accept or reject. This also impacts our conflict detection model, as in some paths through the automaton the filter operates on certain resources and in other paths it does not. There are 5 paths through the automaton, each with a (possibly) different sequence of operations on resources. Although these 5 paths do influence our instantiation of the resource operation model, it does not impact our example conflict, as the conflict presents itself in the actions of the filters and not in the matching expressions.

Notice that condition $\text{inputwrong}$ is always read, whereas condition $\text{outputwrong}$ is possibly read. Furthermore, we can state that if the filter accepts it must at least also have read the selector of the message, and written resource $\text{errorvariable}$. 
6.1 Execution-Trace derivation

As explained before, a filter in the composition filters model can either accept or reject. This means that we can represent the sequence of filters as a binary (decision) tree of filter actions. In this tree we can incorporate knowledge about filters; e.g. after a dispatch or error action the remaining filters are no longer evaluated. Also, we know that errorpropagationfilter, can only accept (because it matches all messages: \([\ast]\)) thus allowing us to trim the tree substantially. Figure 4 shows this tree for our filter example.

![Filter action tree](image)

**Fig. 4.** Filter action tree

The figure shows each filter as a node with two edges, one for the accept action(A) and one for the reject (R) action. The edges are labeled with the action that is performed. The leave nodes of this tree are labeled with a “EOFS” label. This indicates that there are no more filters to evaluate, the End Of the Filter Sequence has been reached. The gray nodes and dashed edges indicate filters and actions which can not be reached. As stated previously, filter errorpropagationfilter can only accept, thus allowing us to trim the tree. We cannot trim the tree down further as the second filter contains conditionals.

To illustrate some of the features of composition filters which enables us to more accurate reason about the filters and to reduce the number of possible paths through the filter sequence, we now show two examples which are not expressed in our example. Consider the following example:

1. `filter1 : FilterType1 = \{ \[*\].foo \};`
2. `filter2 : FilterType2 = \{ \[*\].foo \};`

Here we see two filters, both filters accept the same messages, namely \([\ast\].foo\]. This implies that if filter `filter1` accepts, then `filter2` must also accept. Similar, if `filter1` rejects than `filter2` must also reject.

1. `filter1 : Dispatch = \{ \[*\].foo.bar \};`
2. `filter2 : FilterType2 = \{ \[*\].foo \};`

Here we see again two filters, filter `filter1` is of type `Dispatch`. If this filter accepts the message will be dispatched to in this case `foo.bar`, and as a consequence `filter filter2` will not be evaluated anymore. This trims down the filter evaluation tree depth wise.
By adding this domain information into the tree we can reduce the number of paths that must be considered for analysis.

From the in figure [depicted filter tree we now can derive all possible traces of filter actions. An action trace is a unique valid path from the root node to an EOFS node. The traces for the example are as follows:

1. $errorpropagationfilter \xrightarrow{propagate} paramcheckfilter \xrightarrow{check}$
2. $errorpropagationfilter \xrightarrow{propagate} paramcheckfilter \xrightarrow{continue}$

For each of these traces we generate a new set of resource usage tables. After all filter actions have been translated to operations on resources, we obtain, for each trace, a sequence of operations per resource. Table 2 shows the result of trace 1.

Table 2. Filter actions mapped to operation traces for trace 1

<table>
<thead>
<tr>
<th>action</th>
<th>selector</th>
<th>arguments</th>
<th>inputwrong</th>
<th>outputwrong</th>
<th>errorvariable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$errorpropagationfilter:propagate$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$paramcheckfilter:check$</td>
<td>read</td>
<td>read</td>
<td>read</td>
<td>read</td>
<td>write</td>
</tr>
</tbody>
</table>

Table 3 present the resulting resource operation model for trace 2.

Table 3. Filter actions mapped to operation traces for trace 2

<table>
<thead>
<tr>
<th>action</th>
<th>selector</th>
<th>arguments</th>
<th>inputwrong</th>
<th>outputwrong</th>
<th>errorvariable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$errorpropagationfilter:propagate$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$paramcheckfilter:continue$</td>
<td>read</td>
<td>read</td>
<td>read</td>
<td>read</td>
<td></td>
</tr>
</tbody>
</table>

In summary, through the analysis of the declarative filter specification, we are able to generate all possible resource operation sequences for a given concern. Using these sequences of operations on resources we detect (possible) conflicts.

6.2 Conflict detection

For each resource, a conflict specification is expressed as a conflicting pattern of operations. This pattern can be expressed in many ways, currently we adopt an approach based on regular expressions, but we envision support for other matching language like temporal logic. The pattern is subsequently tried to match on the sequence of operations
for that specific resource. If the pattern matches we have identified a conflict. In section 2 we stated that a conflicting situation occurs, in our example, if the sequence of operations of resource `errorvariable` ends with a write or has two consecutive writes. We define the following regular expression to capture the conflict: \(((\text{write})(\text{write}))|((\text{write})\text{\$})\). We now try to match this expression to the, in table 2, stated operation sequence, `read,write,read,write`, of resource `errorvariable`. The expression matches the sequence and the conflict has been detected. For the operation sequence presented in table 3, the expression does not match.

7 Conflict detection in Compose*

This section provides an overview of the approach as implemented in Compose*. Compose* provides an implementation of the Composition Filters approach. It is currently implemented on the Microsoft .NET framework. We are currently developing tooling to also support the Java and C programming language, as well. Both the Composition Filters and the .NET framework are language agnostic and Compose* thus provides language independent aspects on a language independent platform. The presented approach has been tested in our .NET implementation only, we would like to stress that the conflict presented here is between aspects, in Compose* these aspects are implemented as language agnostic filters and thus the approach and result remains the same across platforms. The Compose* architecture is developed in such a way that all platform specifics are created as front- and back-ends to the core compiler. This core compiler uses abstractions from the platforms to perform analysis and determine join points. The approach discussed in this paper is implemented in the core compiler, and is thus platform independent.

7.1 Tool architecture

Figure 5 presents an overview of the conflict detection process, in the context of Compose*. The picture is an instantiation of the figure as defined in section 4.

The Superimposition ANalysis Engine (SANE) takes all superimposition specifications and resolves these. The result is a set of filtermodules for each concern that is a join point, i.e. for each concern which has filtermodules superimposed on it.

The FILTer ordering and cHecking (FILTH) module iterates over all shared join points, in our case all concerns with filtermodules, and orders the filtermodules. This can be influenced by the developer via an explicit partial ordering specification. If there are multiple possible orders, FILTH selects an arbitrary order. The result is a sequence of filtermodules for each shared join point.

The SEmantiC REasoning Tool (SECRET) iterates over all shared join points and retrieves the sequence of filtermodules. These filtermodules are subsequently transformed into one sequence of filters. Next, SECRET retrieves all valid paths through this sequence of filters. This involves building the filter action tree, as in figure 4 and deriving all valid execution traces as presented in section 6.1.

For each valid path we first retrieve the filter instance specific resources and operations, this is the condition, matching and substitution specification. These parts are
evaluated to determine whether the filter accepts or rejects. These operations on resources related to these filter instance specific parts, have thus to be carried out before the operations which occur in either the accept or reject actions of that filter. The resulting resource-operation tuples are stored.

SECRET has a specification file where filter types are mapped to filter actions, which are mapped to resource operations tuples. An example of such a file is discussed in the next section. SECRET transforms the filter actions to the corresponding resource-operation tuples, these are attached after the filter instance specific resource-operation tuples. If the action is of type meta, the user defined semantics are retrieved, this will be explained in more detail in section 7.3.

The result is a sequence of operations on resources per execution trace and per shared join point. These operation sequences are matched to the conflict detection rules for that specific resource. These can be assertion or conflict patterns. If an assertion does not match the specific operation sequence, SECRET reports a violation. Likewise, if a conflict pattern matches, a conflict is reported. During this entire process the order of filters and filter elements is retained in the order of operations on resources.
7.2 Filter behavior specification

SECRET uses a specification file to look up the behavioral specification for a given filter type. In our current implementation the file consists of four elements:

1. A set of resource specifications. This states all resources in the system and which operations are allowed to operate on these resources. This can provide some consistency checking, and can enforce using the correct names of resources and operations. This verification can be turned on or off.

   ```xml
   <resources>
   <!-- ... -->
   </resources>
   
   Listing 1.6. Resource specifications
   ```

2. A mapping between filter types and filter actions. This consists of a set of entries for filters, where each filter has a filter type, an accept and reject action element. These action elements define the accept and reject actions of a specific filter type.

   ```xml
   </filters>
   <!-- ... -->
   </filters>
   
   Listing 1.7. Filter types to filter actions specifications
   ```

3. A mapping between filter actions and resource-operation tuples. This consists of a set entries for filter actions, where each action entry has a name and a set of operation elements. Each operation element has an operation and a resource on which it operates. Attribute endofset indicates whether the specific action will terminate the filter evaluation of the remaining filters in the filter sequence. This is used for trimming down the number of possible paths through the filter sequence. This value defaults to false, and has to be explicitly set to true, e.g. in case of a dispatch or error action.

   ```xml
   <actions>
   <!-- ... -->
   </actions>
   ```
Listing 1.8. Filter actions to resource-operations specifications

4. A set of conflict detection rules. This consists of a set of conflict and assertion elements. Both elements types have a pattern, a resource and a message. The pattern is a regular expression which is tried to match on all operation sequences for the specific resource. The message is displayed in the error report if the assertion pattern does not match or if the conflict pattern matches.

Listing 1.9. Conflict and assertion specifications

7.3 User defined advice semantics

For filter types we can retrieve the semantic specification from the filter behavior specification file. However for user defined advice, i.e. the ACTs, we have no such specification in that file. In this case the developer can manually add this specification to the user defined advice. The developer can add an annotation to his user defined advice. Imagine if we would have implemented the error propagation concern not as a filter type, but as a meta filter and an ACT. The resulting code and semantic specification would look like listing 7.3.

At line 1 the user-defined behavior specification (annotation) is attached to the propagate advice. SECRET extracts the semantic information from the user defined advices and translates these to sequence of resource-operation tuples per advice, which can be used when we encounter a meta action which causes the execution of this advice method.
7.4 Tool experience

The example presented in this paper has been implemented in our upcoming release. The current public version 0.5b features a limited version of the SECRET tool. The current version does not yet incorporate the mechanism to determine all valid paths through a filter and through a filter set. However for the conflict detection of our example this has no impact. SECRET was still able to detect the conflicts presented in the running example. In [7] we showed an example of an encryption and logging conflict. And in [24] we presented a different example based on a Jukebox system. Both these conflicting situation were implemented with the Compose* tooling, and the presented conflicts were detected. SECRET has three, user selectable, analysis modes:

- **SingleOrder** This mode only analyzes the chosen filter module order selected by FILTH.
- **AllOrders** This mode analyzes all valid filter module orders at shared join points. This can provide more information and provide the user with a possible conflict free ordering.
- **AllOrderAndSelect** The final option is that if, in the selected filter module order, a conflict is detected and there is another valid filter module order which is conflict free, than a conflict free order is chosen. Hence, this sometimes yields automatic resolution of the conflict.

Figure 6 shows the output file generated by SECRET. It only shows the shared join points, in our example case class CC.CX.SF. The report shows which filter module order was selected: ErrorPropagation.propagate followed by ParameterChecking.check. Subsequently the composed filter sequence is enumerated. The report only shows conflicting filter actions orders. In this case there is only offending filter action order, namely: errorpropagationfilter and incheckfilter. The CpsDefaultInnerDispatchFilter is automatically added to the filterset by the compiler. This is a catch-all filter, all messages that reach this last filter will be dispatched to the “inner” object. The report also states that there was a conflict on resource errorvariable, the corresponding conflict message and the offending sequence of operations.

7.5 Tool performance

Currently we have not been able to test the technique and tool on a truly large scale system. There are two elements to take into account when considering the performance. First we have to consider the time it takes to build up the automaton for one join point. This is polynomial with the number of filter elements and conditions. Secondly, we have to take into account the number of shared join points in the entire system. The analysis performed on these points is linear with the number of shared join points. However, as explained previously (see 6.1), there are still a lot of optimizations which can be done.

The complexity of the analysis algorithm for a sequence of filters is as follows:

\[ \text{#states} = \text{#messages} \times (\text{#filterelementstates} \times \text{#filterelements}) \]

Where \text{#messages} is the number of messages that the filter sequence can accept. The \text{#filterelementstates} is the maximum number of states that filter elements can have, this is linear with the number of conditions, conditional operators and the number of
Fig. 6. Example SECRET error report

...matching expressions. The \( \# \text{filterelements} \) is a function of the number of filter times the number of filter elements in these filters.

To check whether a conflict or assertion pattern matches, an automaton is created. This automaton is subsequently intersected with the automaton representing the filter sequence. If this intersection is empty, the filter sequence is conflict free, else it contains at least one conflict. The analysis algorithm of a filter sequence is thus polynomial.

7.6 False positives and false negatives

We can distinguish two classes of false positives; the first is because our model of the filters is not complete and lacks certain details. An example of such a fault would be the case where two filters match on the same condition, and thus if the first accepts than the second must accept as well. If the corresponding paths remain in the analysis system, we could detect a conflict which is not present. Our filter analysis framework transforms and optimizes most of the elements in the filter specifications. This framework is also used in other parts of the Compose* compiler, e.g. to determine the updated signatures of objects, due to method introductions. Some optimizations that the framework carries out, do provide some loss of detail, especially regarding the conditions, we plan to improve on this situation in our upcoming release of Compose*. Also, currently we do not distinguish between conflicts encountered inside a single filter module and those between different filter modules. One could argue that conflicts detected inside a single
filtermodule are intended, and that these need not be checked when the filtermodule is compiled.

The second class of false positives relates to the wrong or lack of semantic specification. The resource operation model is now taken from either the filter description file or from the user defined annotations in the code. If these are not correct or capture too much detail, we could detect conflicts which might not occur. Explicitly specifying the resources and the alphabet of the resources, in the filter specification file, can ensure the resource operation models match and thus can be joined. Although enforcing this alphabet limits the expressiveness, it does provide more robustness for the model. For example, one cannot add a resource in an annotation which has not been defined, similar for operations. This also ensures that the assertion and conflict rules only have to account for the operations in the alphabet of the resources, i.e. a closed-world assumption.

Using the filter specification file also limits the possibility of false negatives, as the possibility of disjoint resource operation models is limited. If one developer would use the \texttt{wr} operation to indicate a write, than he is obligated to also add this operation to the alphabet of a specific resource, in this case he might encounter that another developer already added this operation as \texttt{write}. Hence we can ensure that the resource operation models can accidentally be disjoint.

### 7.7 Errors vs. Warnings

In Compose\textsuperscript*, SECRET only produces warnings; if a conflict is detected, we do not halt the execution of the compiler, we merely report the violation to the user. We envision that it be resource dependent whether to halt the compiler or not. Some resources are deemed more important than other, e.g. if an security concern is not (correctly) applied, the entire system can be scrutinized. There is also a clear distinction between the normal compiler errors you would get from a regular aspect compiler. Imagine changing, through introductions, the inheritance hierarchy in such a way, that circular inheritance would result. This would be a clear error, as the base programming language would be unable to support this. In contrast to these kinds of conflict, the behavioral conflicts pose a different kind of conflict. As the system can still be executed, be it with possible flaws, we may never be certain that the composed problem is truly defective. In certain cases we might be sure and in these cases one should be able to flag these conflicts as errors. As a default we feel that these conflicts should be treated as warnings rather than errors.

### 8 AspectJ instantiation

In sections \[5\] and \[6\] we provided an instantiation of the conflict detection model for the composition filters approach. We now show that the presented approach is generic. We provide a worked out instantiation of the approach for the AspectJ language\[28\]. We will first explain briefly the AspectJ advice execution model, next we show the actions of the example concerns \texttt{ErrorPropagation} and \texttt{ParameterChecking} in AspectJ. Finally, we discuss the conflict situation and show how the conflict is detected with
our approach. We show an AspectJ instantiation because it is currently the most used and accepted AOP language, and most of the AOP languages resemble its advice and execution model.

8.1 The AspectJ execution model

AspectJ supports the following three advice types:
- **before**: A before advice is executed before the join point is executed.
- **after**: An after advice is executed after the join point is executed.
- **around**: An around advice presents advice which can conditionally execute the original join point, and it can share state between the before and after part of a join point.

These advices can be applied to a specific join point using a pointcut. If these advices are composed at the same join point, an execution order must be chosen. In AspectJ one can explicitly enforce a partial ordering among aspects, using the declare precedence construct. If one does not give such a specification, an arbitrary order is selected, where advice from the same aspect is ordered according to the declaration order.

Although there are many subtleties to the advice execution order of AspectJ, we abstract from these and use the following simplified model. We assume that before advices are always executed before the after advices. And that the before parts of around advice are executed immediately after the before advices. Similar, we assume that after parts of the around advice are executed before the after advices. This enables us to model all advices at a shared join point as a sequential composition of before, around and after advice. Although in general, around advice cannot simply be split in a before and after part, for our reasoning model this is appropriate. There are two reasons for this. Firstly, if there are multiple around advices, the composed behavior cannot be modeled with a sequential composition of the two around advices as a whole, but rather as a sequence of the before and after parts of these around advices. Secondly, we have to consider the case where one around advice does not proceed, this has impact on the execution of other advices at that same join point.

Next we have to map the advice behavior to our resource operation model. As we do not have a declarative specification as with composition filters, we have to rely on the developer to provide a complete specification of the behavior, rather than deriving parts of the behavior from the specification. We have chosen to annotate advice with a behavioral specification, which is similar to the one derived from the composition filters. The next section will show how this concretely works for our example.

8.2 AspectJ implementation of the Example

We illustrate the advice execution model of AspectJ through the use of our running example. Listing 1.10 shows an example implementation of concern ErrorPropagation as presented in sections 2 and 5.

---

5 We are aware that AspectJ supports two additional advice types, after returning and after throwing. For our conflict detection this distinction is not necessary, and we therefore classify these two advice types as after advice.
public aspect ErrorPropagation
{
@Semantics("errorvariable.read,errorvariable.write,errorvariable.read,message.
proceed")
int around(): execution(int *(..))
{
    int temp_error = ErrorFramework.OK;
    if(ErrorFramework.CURRENT_ERROR == ErrorFramework.OK)
    {
        temp_error = proceed();
        if(temp_error != ErrorFramework.OK)
        {
            ErrorFramework.CURRENT_ERROR = temp_error;
            ErrorFramework.LogError(ErrorFramework.CURRENT_ERROR);
        }
    }
    return ErrorFramework.CURRENT_ERROR;
}
}

Listing 1.10. ErrorPropagation in AspectJ

Listing 1.10 defines aspect ErrorPropagation, which has one around advice (lines 4-17). This advice will applied to all method executions returning an integer, see the pointcut, execution(int *(..)), at line 4. The advice first initializes (line 6) a local variable to store the return value of the original call. It subsequently verifies that the program is not in an erroneous state. If it is not, it calls the original method(line 9), and checks the result of this execution. If this is not OK, it updates the current error state and logs this error. Finally we return the current error, in case of an erroneous state, or the newly acquired error.

Line 3 states the behavioral specification of this advice; it uses a “Semantics” annotation to attach this specification to the advice. The semantics annotation expects a single string argument, which contains a comma separated list of resource operation tuples. In this case, the annotation states that this advice reads (line 7), writes (line 12) and reads (line 13) resource errorvariable. We also explicitly model the proceed call in this semantic specification, this enable us to specify which parts of the behavior are executed before the proceed, and which parts after.

A possible AspectJ implementation of aspect ParameterChecking is shown in listing 1.11.

public aspect ParameterChecking
{
    final int INVALID_PARAMETER_ERROR = -89;
    @Semantics("args.read,errorvariable.write")
    before(): execution(int CC.CX.FS.compare_data(..))
    {
        if(inputWrong(thisJoinPoint.getArgs())
        || outputWrong(thisJoinPoint.getArgs()))
        {
            ErrorFramework.CURRENT_ERROR = INVALID_PARAMETER_ERROR;
        }
    }
    private boolean inputWrong(Object[] obj)
    {
        // Verify that non of the input parameters are null
    }
    private boolean outputWrong(Object[] obj)
}
Aspect ParameterChecking defines a before advice (lines 6-13), which checks whether the input and output parameters are wrong. If so, the advice sets the current error state to an INVALID_PARAMETER_ERROR value. Again we specify the behavior of this advice through the use of an annotation (line 5). The annotation states that this advice reads resource arguments and subsequently writes resource errorvariable. We have omitted the exact implementation of determining which parameters are input and which are output, and whether these parameters are invalid.

8.3 Discussion

The two presented advices are applied to the same join points, in this case every time method CC.CX.FS.compare_data is executed. Now assume that advice errorpropagation is executed before advice parameterchecking. This results in a similar conflict situation as discussed in section 2. To verify this, we first have to collect the annotations from all advice and determine the correct sequencing of the behavioral specifications. Then we can append the operations to the list of operations of the corresponding resources. Subsequently, we try to match our conflict and assertion rules to the resulting sequences of operation on the resources.

Although this demonstrates that the conflict detection model is sufficiently generic to support many AOP languages, the use of a more restricted and declarative advice language has significant benefits when analyzing such a language. For example, we could leverage the declarative form of composition filters, to automatically derive parts of the behavioral specification. Parts of such a behavioral specification could also be derived from an AspectJ-like advice language, by looking at the usage of the thisJoinPoint variable and by looking at the context binding the pointcut does, e.g. args and target.

An initial implementation of this work has been created as part of the Common Aspect Proof Environment (CAPE).

9 Conclusion

9.1 Discussion

The presented approach relies on the fact that one has global knowledge of the program. Although this is usually described as a not so good property, we do feel that one needs this information to detect certain kinds of conflicts. Also, as we only consider conflicts between aspects at shared join points, the number of possible interactions is also limited. Using good tool support can help to provide an overview of the shared join points and the resource operation models of the advice that are applied at those join points.

Our approach assumes a very thorough domain analysis and ontology to determine the canonical and orthogonal set of resources, operations and conflict rules for a given
domain. In this paper we provided in analysis of the composition filters massage manipulation domain. Once such a detailed set of resources and operations are established, we can map behavior of advice to these sets, and detect conflicts between advice. This requires some –extra– effort of course, but we feel that benefits outweigh the costs of creating such domain models.

9.2 Future Work

Compose*.NET has a tool which can extract a part of the semantic specification for user-defined advice automatically[39]. This tool performs a detailed program analysis and could also be used to derive semantic specification from base code. This would enable us to detect base-aspect conflicts as well, using the same resource operation model.

An further extension would be to apply the approach at runtime, so we can draw definite conclusions when a conflict occurs. If we encounter such a conflict we could warn the user about this conflict or stop the execution of the program. We could use analysis results obtained during compilation to optimize our runtime checking.

Using the filter analysis tools currently available in Compose*, we can build up a global call graph of the system, i.e. we can link output filter sequences to input filter sequences. This could potentially allow us to reason about conflict between aspects at disjunct join points, but within the control flow of each other.

Compose* supports, via the meta filter, concurrent advice execution, before and after advice. We currently only take a subset of this execution model into account. We plan to expand the reasoning engine, to also support control flow, for this we do have introduce the appropriate operations to capture this.

In the current version of Compose*, we adopted regular expressions to express the conflict and assertion rules. These are very expressive but might require too much specification in some cases, i.e. all non relevant operations must be explicitly ignored. Using a temporal logic language, Linear Temporal Logic(LTL), might be a more interesting alternative. Also, if we were to include control flow in the resource operation model, regular expression and temporal logic would not be sufficient anymore. In this situation we could use path logic like Computational Tree Logic(CTL) to express conflict rules.

Our conflict detection model is generic and abstract. We can apply the same conflict detection approach at a design level, i.e. at the requirements or architectural level. As the resource model is generic enough, our approach is applicable in these cases as well and we may even reuse parts of the Compose* tool set for this.

9.3 Summary

This paper presents a novel approach for detecting behavioral conflicts between aspects. Our approach defines the behavior of advice in terms of operations on an (abstract) resource model. We first analyze the advice behavior and represent the behavior at each (shared) join point, according to our conflict detection model. Next we verify this representation against a set of conflict rules and assertion rules. The resource-operation model allows us to express knowledge about the behavior of advice at both concrete and abstract levels.
In section 2 we show an actual behavioral conflict encountered in an industrial situation. We foresee a need for tooling that checks for consistency and detects conflicts between aspects, before AOSD technology can be successfully applied in an industrial context; as aspect technology is incorporated in large and complex systems, and is used to implement not only systemic crosscutting concerns, but also to implement more component specific concerns, there will be an even stronger need to have verification tools for avoiding conflicts between aspects, such as presented in this paper. In section 3 we discuss the problem space of aspects among conflicts, working towards our focus on behavioral conflicts between advices at shared join points.

The presented approach is generic and can be applied to most, if not all, AOP languages. It requires the ability to detect shared join points for such a language and the ability to associate a resource operation specification to each advice, either by analyzing the advice or through annotations. If these requirements are met, our approach supports reasoning about the behavior of the composition of multiple advices. We discuss the application of the approach to both the Compose* language and AspectJ. In the case of Compose* it is possible to exploit its declarative advice specifications so that the programmer normally does not need to annotate the program.

We believe the approach presented in this paper offers a powerful and practical means of establishing behavioral conflict detection with a minimal amount of explicit behavior specifications from the programmer. The approach has been implemented and tested within the Compose* and CAPE tool sets.

Acknowledgments

We would like to thank Tom Staijen for his contribution to the work described in this paper. We would like to thank Gurcan Gulesir and Wilke Havinga for providing feedback on earlier drafts of this paper. Many thanks to Tom Staijen, Eyal Dror and Shmuel Katz, who have demonstrated that the conflict detection model we described in this paper is indeed generic: they applied our approach to a generic AOP program representation, by porting the SECRET tool, and demonstrating that the approach works correctly for both Composition Filters and AspectJ programs, in the context of the CAPE environment. Finally, we would like to thank Remco van Engelen from ASML for providing feedback on this paper. 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.

References

1.  
2.  
3.  
4.  
5.  
6.
28. A. Colyer. AspectJ. In Filman et al. [31], pages 123–143.