A Graph-based Approach to Modeling and Detecting Composition Conflicts Related to Introductions

Wilke Havinga  
Software Engineering group  
University of Twente  
P.O. Box 217, 7500 AE  
Enschede, The Netherlands  
havingaw@ewi.utwente.nl

Istvan Nagy  
SSD Architecture Group  
ASML Netherlands B.V.  
De Run 6501, 5504 DR  
Veldhoven, The Netherlands  
lstvan.Nagy@asml.com

Lodewijk Bergmans, Mehmet Aksit  
Software Engineering group  
University of Twente  
P.O. Box 217, 7500 AE  
Enschede, The Netherlands  
{bergmans,aksit}@ewi.utwente.nl

Abstract
The goal of this paper is to model and detect composition conflicts related to introductions. Within this context, we identify several categories of composition conflicts. To analyze the causes of these conflicts precisely, we first model the structure of programs as graphs. Next, we model introductions as graph transformation rules. We define explicit rules to describe when composition conflicts related to introductions occur. We built a prototype tool that detects and visualizes the occurrence of such conflicts in AspectJ programs, making use of an existing graph analysis and rewriting tool. The graph-based models are generated automatically from the source code of Java programs and AspectJ introductions. However, our approach does not make strong assumptions about either the aspect or base language; it has been designed to be applicable to other AOP languages.

Keywords  AOP, aspect composition, aspect interactions, aspect interference, introductions, conflict modeling, conflict detection, graph rewrite systems

Categories and Subject Descriptors  D.3.3 [Programming Languages]: Language Constructs and Features; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

1. Introduction
Aspect-oriented programming languages allow for the modular specification of crosscutting concerns. To facilitate this, aspect-oriented languages offer new kinds of composition mechanisms. Introductions are one such mechanism, also referred to as structural superimpositions or inter-type declarations [2]. Introductions are constructs that affect the structure of a program, for example by adding a method to a class or by changing the inheritance structure.

In this paper, we focus on analyzing conflicts related to introductions. For example, introductions may have unintended effects, cause a program to fail to compile, or cause the program to be ambiguous (i.e. the program source can be interpreted by the compiler in more than one way).

To precisely analyze such composition conflicts, we automatically convert the structure of Java and AspectJ programs (source code) to a graph-based model, which we refer to as program model. In addition, we automatically convert the introductions that are part of the AspectJ source code to graph transformation rules, which can be applied to such a program model.

Next, we use an existing tool set to analyze these graph-based models. We explicitly model several kinds of composition conflicts as graph matching patterns. Using a graph rewriting tool to match these patterns against our program models, we can automatically detect the occurrence of composition conflicts. The same tool is used to apply the introductions (represented by graph transformation rules) to the program model. This tool enables us to detect situations in which the transformations can be applied in different orders, leading to potentially different transformed program structures.

The rest of the paper is structured as follows: in section 2, we identify three categories of composition conflicts related to introductions, and illustrate these through various examples. In section 3, we introduce a graph-based model of introductions. In section 4, we use these models to detect and visualize the conflicts. In section 5, we discuss how aspect language developers can address or avoid particular types of conflicts. Section 6 discusses alternative approaches to detecting conflicts related to introductions. The paper finishes with a discussion of related work and a conclusion.

2. Issues related to introductions
Many aspect-oriented programming languages offer various composition mechanisms to adapt the structure of a program, for example by changing the inheritance structure or by introducing additional program elements, such as methods, instance variables or annotations. In this section, we distinguish three categories of composition conflicts that can occur when an aspect uses introductions.

2.1 Violation of language rules
Introductions may cause violations of basic language rules. Although such situations may seem obvious, this is not always the case because of the dependency inversion [16] introduced by aspects - that is, aspects may superimpose elements (e.g. methods) on existing program elements (e.g. classes). However, this change is transparent to the class on which the method is superimposed.

It is sometimes argued that violations of basic language rules are already detected by the base language compiler, and hence, that an AOP compiler does not need to detect them. In addition to giving examples of such violations, in this section we explain why this is not always the case.
2.1.1 Example: Multiple, conflicting method definitions

For example, consider the application fragment in listing 1.

```java
public interface Persistent { ... }

public class BusinessObject implements Persistent { ... }

aspect PersistenceImplementation {
    void Persistent.saveChanges() { db.update(...); }
}

aspect ObjectCache {
    void BusinessObject.saveChanges() { cache.setVal(...); }
}
```

Listing 1. Conflicting method introductions

In this example, two unrelated aspects both introduce a method named `saveChanges` to classes that match the specified type. The aspect `PersistenceImplementation` introduces such a method on all classes that implement the interface `Persistent` (line 6), whereas the aspect `ObjectCache` introduces a method with the same name on the class `BusinessObject` (line 10). It is not immediately obvious that these aspects conflict with each other, because the type patterns used to introduce these methods are different (and seem unrelated). However, because class `BusinessObject` implements the interface `Persistent` (line 3), both aspects introduce a method with the same name to the same class, which leads to a naming conflict.

One might argue that such conflicts can be detected by existing (base) language compilers. However, this is the case only when aspects are woven into base-language source (e.g. Java) and then compiled using an existing base language compiler (e.g. javac). In practice this is not always feasible, because aspect languages may introduce constructs that cannot comfortably be expressed in terms of base language source code – for example, because the aspect language extends the type system in a way that would not be accepted by the (strict) typechecker of the base language compiler.

In addition, the interpretation of base language rules may be extended by the aspect language. For example, in AspectJ projects, base classes may contain calls to methods that are introduced by an aspect. This means that even though the base code needs to be parsed before weaving (to accommodate pointcut evaluation), the compiler-level semantic checks may have to be postponed until the weaving is done. So, some phases of aspect and base code compilation may become interleaved, leading to various degrees of integration between aspect and base code compilers. As a consequence, the implementation of compiler-level semantic checks (i.e. enforcement of language rules such as those described in this section) may have to be reconsidered. For example, such rules should take introduced methods into account, even if no base language source-code level representation of these methods exists during any phase of the compilation.

In addition, design choices made by aspect language developers may sometimes lead to language semantics that may be different from what programmers expect. We demonstrate this using the example above. According to the actual semantics as implemented by the AspectJ compiler, the introduction on the interface `Persistent` (line 6) effectively supplies a default implementation which can be overridden by base classes implementing this interface. This means the above example compiles and runs in AspectJ; the method that is declared directly on the class itself (line 10) overrides the one declared on the interface (line 6). In other words, the way in which the interface `Persistent` now functions is similar to an abstract base class, except that the programmer is not bound by the single inheritance restrictions imposed by the Java base language - as classes may implement multiple of such “abstract class”-like interfaces. We argue that many programmers will expect aspects to add behavior or (maybe) override existing base behavior; supplying a default implementation that can be overridden by a base class is at least not the semantics we would have expected.

Even though the AspectJ compiler does not consider our example to be a language rule violation, it still has to implement other (additional) language rules because of its adapted interpretation of Java interfaces. To demonstrate this, listing 2 shows a revision of listing 1. In this example, two aspects declare a method `saveChanges` on two different interfaces (lines 5 and 9). To deploy these generic aspects in our particular application, a binding aspect (line 12-15) declares that the class `BusinessObject` implements both these interfaces. This is a commonly used technique to deploy a generic aspect in a specific application context, as (for example) demonstrated by AspectJ implementations of several design patterns [7].

```java
public interface Persistent { ... }

public interface Cache { ... }

aspect GenericPersistenceImplementation {
    void Persistent.saveChanges() { db.update(...); }
}

aspect GenericObjectCache {
    void Cache.saveChanges() { cache.setVal(...); }
}

aspect BindingAspect {
    declare parents: BusinessObject implements Persistent;
    declare parents: BusinessObject implements Cache;
}
```

Listing 2. Conflicting method introductions revisited

In listing 2, the class `BusinessObject` now effectively inherits two competing “default implementations” of the method `saveChanges`. It is undefined which method definition should take precedence. The AspectJ compiler recognizes this situation as an error and gives a message accordingly.

Based on these observations, we conclude that the definition and enforcement of even such basic language rules is not as straightforward as one might have expected.

2.1.2 Example: Cyclic inheritance

To show that many existing language rules are in some way affected by aspects (or, more specifically, introductions), we supply some more examples. Consider accidentally declaring a circular inheritance structure as in listing 3:

```java
public class Ellipse extends Circle { ... }

aspect CircularShapes {
    declare parents: Circle extends Ellipse;
}
```

Listing 3. Cyclic inheritance (caused by an aspect)

When AOP languages support a construct to change the inheritance structure, it is possible to define a circular inheritance structure (using that construct). However, OO languages usually assume that the inheritance structure cannot contain cycles. It is interesting to note that AspectJ defines an even more strict language rule, which also prevents an aspect from introducing circular inheritance (given that we already know that the base program does not have circular inheritance itself). The AspectJ compiler enforces the following rule: given that A extends B, it is only possible to declare A extends C (thus overwriting the original superclass of A, as Java does not support multiple inheritance!) when C is itself a subclass of B.

---

1 here we assume that the method is introduced on classes, not on the interface itself.
Clearly, this rule is always broken when we try to introduce a cycle in the inheritance tree (i.e. the AspectJ compiler detects this as an error). Thus, we see that aspect-oriented languages may also introduce additional language rules.

2.1.3 Example: Extending a final class

As another example, consider trying to extend a final class using 'declare parents'. Final classes are sometimes used in libraries to prevent applications from accessing protected fields or methods (which they could do by extending a library class); the AspectJ compiler applies this rule also to the 'declare parent' construct.

To conclude, the examples in this section have shown how basic object-oriented language rules can be violated by introductions: first, a class cannot contain two distinct program elements (e.g. methods or fields) with the same name/signature. Second, we usually do not want to 'break' existing language mechanisms such as inheritance or the finalization of classes. Note that the kind of rules mentioned in this example are in principle language specific, although the examples mentioned here probably apply to most, if not all, object-oriented languages. We have shown that developers of aspect languages have to make non-trivial decisions as to the exact (re)interpretation of existing language rules in the presence of aspects. Therefore, we see the need for a generic conflict modeling and checking tool that can express different rules for various languages.

2.2 Introduction has unintended effects

An introduction may have unintended effects. Consider the example in listing 4. Here, the aspect Printing introduces a method named getSize on the class AlertDialog. However, this introduction has another effect: it overrides a method with the same name, which the class DialogWindow already inherited from its parent class DialogWindow.

In this example, a method clear is introduced on all classes that are supposed to contain sensitive data (line 9). These classes are defined to be those that are part of our application package, but do not implement the interface Persistent. However, classes can be adapted to implement this interface using the declare parents construct (line 1). This way, the pointcut depends on this change to the program structure. This can be an intended effect, but such dependencies can also lead to ambiguous code. A simple example of such a case is given in listing 6.

We observe that the program structure is changed by introduction mechanisms, but also "queried" by pointcut designators. It is definitely possible that a pointcut refers to the same program elements that are also changed or introduced by an aspect. Therefore, introductions can influence the composition as specified by pointcuts.

Listing 5. Pointcut depends on an introduction

Listing 5 specifies a pointcut (line 3+4) which selects all classes that implement the interface PersistentRoot. However, classes can be adapted to implement this interface using the declare parents construct (line 1). This way, the pointcut depends on this change to the program structure. This can be an intended effect, but such dependencies can also lead to ambiguous code. A simple example of such a case is given in listing 6.

In this example, a method clear is introduced on all classes that are supposed to contain sensitive data (line 9). These classes are defined to be those that are part of our application package, but do not implement the interface Persistent (line 7). However, another aspect may declare specific classes to be persistent (line 13). In this case, the specification is ambiguous, because the order in which the two parent declarations can be evaluated and applied lead to different results. Whether or not the class User should implement the interface SensitiveData depends on the arbitrary choice whether the declaration in line 13 is applied before or after the declaration in line 7. Hence, we cannot determine whether the method clear should be introduced to the class User. For a more detailed discussion of this issue, see [8].

3. A graph-based model of introductions

In the previous section, we identified several types of conflicts related to introductions. To facilitate a precise analysis of such conflicts, we first present a concrete and exact model of introductions.

In general, we can say that a composition construct involves two parts: a selection (what to compose, e.g. two objects) and an action (how to compose, e.g. by sending a message from one object to the other). In the case of aspect-based composition, we can think of the selection mechanism as pointcuts or structural patterns (such as type patterns in AspectJ), whereas the actions in AOP terminology correspond to e.g. advices or (structural) introductions (see [14], chapter 2 for a detailed reference model of AOP constructs). In the remainder of this paper, we use this simple model of composition for the analysis of structural introductions in AspectJ.

In fact, the example in listing 4 works fine in AspectJ.

\[\text{Listing 4. Method introduction overrides an existing method}\]

There is a difference between this kind of conflict and the previous category: in this case, there is no inherent technical reason (such as violation of language rules) why this composition would be invalid. However, the implicit effect of effectively overriding an existing method may be unintended and undesired. A compiler or checking tool cannot generally judge whether overriding is deliberate or unintentional. Even so, a compiler should preferably flag such situations and issue a warning, as many compilers already do in similar situations that may indicate programmer errors. In section 5, we discuss this issue in more detail.

2.3 Ambiguous aspect specification

A third kind of composition problem that we study is caused by composition specifications referencing and modifying the same program model.

\[\text{Listing 5. Pointcut depends on an introduction}\]

\[\text{Listing 6. Ambiguous introduction}\]
3.1 Graph-based program representation

To analyze the effects of introductions, we first need to model the elements (i.e., a concrete program model) to which introductions can be applied.

In this section, we define a simple mapping of program structures to a graph-based representation. Next, we model introductions as transformation rules on such graphs. In the next section, we will use these mappings to analyze the identified types of composition conflicts in detail.

Figure 1 shows a graph-based representation of the structure of the program in listing 1. This graph is constructed using the following types of nodes and edges, representing a simple language model:

- **id-nodes**, defining a unique identifier for each program element
- **kind-nodes**, representing a kind of program element
- **name-nodes**, representing a program element name or signature
- **isa-edges** (between id and kind nodes)
- **named-edges** (between id and name nodes)
- Any other edges between two id nodes, representing relations between program elements. These edges are labeled using the name of the relation (e.g., `implements`).

A program is mapped to this graph representation as follows: each program element is mapped to a node with a unique identity (node `A`); each program element has (at least) two outgoing edges: one edge labeled `isa`, pointing to a node that represents the kind of program element (e.g., class or method), and an edge labeled `named`, pointing to a node that represents the name of this element. If several program elements have the same name, their `named` edges point to the same node. If nodes are of the same kind, their `isa` edges also point to the same node. All other edges model the relations between program elements in the given language model. For example, class nodes may have `implements` relations (edges) to interface nodes.

![Figure 1. Graph representation of the program in listing 1](image)

To demonstrate the mapping, the left-hand side of figure 1 represents the AST of the base program in listing 1. It contains only two program element nodes\(^4\), labeled `interface1` and `class1`. These nodes have edges to nodes representing their name and kind, and in addition, node `class1` (BusinessObject) has an `implements`-relation to node `interface1` (Persistent).

Similarly, the right-hand side shows a representation of the structure of the aspects defined in listing 1. Note that both aspects have edges labeled `hasMethod` to distinct method-nodes (as these methods are distinct program elements). However, both method-nodes have a `named` edge that points to the same `name-node`.

We wrote a prototype implementation that automatically maps (relevant) parts of Java base code and AspectJ introductions to this representation. As the model imposes no restrictions on the particular types of relations or program element kinds, it should be straightforward to create mappings from other languages, such as for example UML class diagrams.

3.2 Introductions as graph transformations

The graph-representation presented above was chosen because it enables us to express the selection part of introductions (type patterns, pointcuts) as simple graph matching patterns.

Figure 2 contains four examples of such matching patterns. The graph matching pattern in figure 2(a) matches all nodes that have an `isa`-edge to the node labeled `Class`. In other words, it matches all program element identifier nodes that represent a class in the system under consideration. A question mark in a graph pattern means the label of that node or edge is a "don’t care" in the matching process (i.e. it may contain any value). The pattern in figure 2(b) selects node(s) that have an `isa`-edge to `Class`, and a `named`-edge to `BusinessObject` - i.e. it selects the class `BusinessObject`. The pattern in figure 2(c) extends this example, by selecting any node that can reach the `class BusinessObject` node through one or more `extends`-edges, i.e. any subclasses of `BusinessObject`. The plus sign here signifies one or more occurrences of an edge with the given label. Finally, the pattern in figure 2(d) selects any node that has an `implements`-edge to the node representing the interface `Persistent`, i.e. it selects all classes implementing that particular interface.

We will now model the complete introduction specification. To this end, we use a single graph that specifies a selection (a matching pattern as described above) as well as an action (the actual introduction).

![Figure 3. Transformation: method introduction (1)](image)
Figure 3 shows the composition specification rule that corresponds to line 6 of listing 1. In this figure, all the nodes and edges with solid (black) lines together specify a selection pattern as described above. This example specifies the type pattern Persistent, i.e. it selects all classes that implement the interface Persistent, as in figure 2(d). Also, we select a node named saveChanges, which is a Method and is contained (hasMethod) by the program element aspect1. This is the unique identifier of the aspect that declares this introduction.

The thick gray (green) edges are not part of the selection pattern, but specify the action (introduction) that should be executed when this rule is applied. Here, we specify the introduction of an edge labeled hasMethod between the selected class(es) and the method defined within the aspect. Finally, the dotted (red) edge specifies an embargo. In the graph, there must not exist an edge labeled wasIntroduced between the selected nodes. As part of the transformation, we introduce an edge with this label. This prevents the same introduction from being applied at the same location more than once; i.e. after we perform the introduction, the rule will not again match the same location in the transformed program model.

This pattern matches the program model of listing 1, as shown in figure 4 - the nodes and edges involved in the match are in bold-face in this figure.

![Figure 4. Selection: matching program elements](image)

After application of the rule in figure 3, the program model from figure 1 is transformed into a new "state", as depicted in figure 5. The edges between nodes class1 and method1 have been added by application of the introduction rule.

![Figure 5. Introduction: applied to the model from figure 1](image)

### 3.3 Automatic model generation and conflict detection

The diagrams in this paper are visualizations made using the Groove (Graphs for Object-Oriented Verification) tool set [5, 17]. We explain the functionality of these tools as they are used in this paper.

The Groove Editor can be used to create graph representations, e.g. of program models and transformation rules as described above. The graphs are stored as XML files using the GXL (Graph eXchange Language [9]) format - a (de facto) standard used by many graph-based tools. As there exist standard libraries to read and write GXL files, it is easy to create tools that generate graphs in this format.

We have implemented [6] a research prototype of such a tool that automatically maps the structural model of Java programs to program graphs (as in figure 1). In addition, it can map several kinds of AspectJ introductions to graph transformation rules (as in figure 3). This tool is available for download [6].

The Groove Simulator implements a single-pushout graph rewriting algorithm with negative application conditions$^5$; i.e. it can execute the matching and application of transformation rules (introductions) such as those presented above. Given a begin state (e.g. a program model such as figure 1) and a set of transformation rules (e.g. representations of introduction constructs such as figure 3), the simulator tries to match the transformation rules. Each transformation (introduction) is then applied for each match found in the current state. Each transformation can lead to a new state (modified program representation). The simulator can explore the state-space of transformation applications, i.e. it can generate all orderings of matching and applying a given set of transformation rules. It detects states that are isomorphic, i.e. have the same configuration of nodes and edges. Optimized algorithms are used to ensure that graph matching and duplicate state detection can be done in polynomial time in most cases.

Although in this paper we show graph visualizations generated using the graphical user interface (GUI) of the Groove Simulator, this tool can also be run in command-line interface (CLI) mode, not showing any visible representation of the graphs whatsoever. This could be useful to integrate conflict detection in compilers or checking tools. To implement this, detected conflict patterns would have to be converted to a textual error message which is linked back to the original source code. This approach, based on the integration of Groove analysis in existing compiler technology, has already been successfully implemented in the context of semantic conflict detection [4].

To demonstrate the use of the Groove simulator, we complete the representation of listing 1, including the application of both introductions listed in that example.

![Figure 6. Transformation: method introduction (2)](image)

Figure 6 represents the selection pattern in listing 1, line 10. It is very similar to figure 3, but directly selects the class BusinessObject instead of referring to an interface.

$^5$ For a detailed explanation of the algorithms and the tool itself, please see [17].
Figure 7. Orders of applying the introductions in listing 1

Figure 7 shows a full state-space exploration of our example case. In this figure, each node represents a particular state of the program model. Node s7 corresponds to Figure 1. For every transformation rule that is applicable in this state, there is an outgoing edge. For example, the edge \(<introduce\_method1>\) denotes the application of the rule in Figure 3, whereas the edge \(<introduce\_method2>\) denotes the application of the rule in Figure 6. Node s8 refers to the state of the program model as in Figure 5. In this example, we see that the application of both introductions eventually leads to the same (=isomorphic) state (program model), independent of the order in which they are applied. The final state s10, in which both introductions have been applied, is shown in Figure 8. In this state, no more introduction rules can be applied.

These diagrams show us two things: first, because there is exactly one final state, we conclude that any order of applying the introductions led to the same result in this case. This means the program is unambiguous. Second, we can inspect each state for the occurrence of conflicts. The next section discusses this in detail.

4. Analysis of conflicts related to introductions

In this section we use the graph representation of compositions as described above to visualize examples of different types of composition problems.

4.1 Violation of language assumptions

As we have observed in section 2.1, the example in listing 1 violates a basic language assumption. By defining the violation of such language assumptions as matching rules over the program model, we can detect and visually represent the exact location of the problem. Figure 9 depicts a rule that matches violation of the first rule mentioned in section 2.1: if the program model contains a program element that is a class, which has two distinct method elements that have the same name, it violates this language assumption. In this diagram, the dotted (red) line labeled '=' means that the nodes connected by this edge must be distinct nodes in the graph (i.e., the two nodes must not have the same identity).

Figure 8. Final program model after applying introductions

Figure 10 shows that this rule indeed matches in the final state of our previous example (see Figure 8). We can see exactly which elements are involved in the conflict: the involved elements and edges are represented in bold in the figure. Also, we can trace back (by looking at Figure 7) which combination of introductions led to the matching of this rule, and are thus involved in causing the conflict.

It is possible to define such rules for all kinds of language assumptions - which can often be found in or derived from the language specification. To give another example, Figure 11 depicts a rule that matches circular inheritance between classes - another type of language rule violation mentioned in section 2.1. Note that the edge labeled extends+ will match one or more such edges (between arbitrary nodes). We do not include a full representation of this example here.

4.2 Introduction has unintended effects

To detect unintended effects of introductions, it is necessary to define rules that match situations in which such effects occur.
Tool- or compiler-developers can define such rules for their (AOP) language and make their tool issue warnings (or even errors) if these rules are violated. Using the graph-based approach we can trace back why the situation occurred (e.g. which introduction caused it), which could help a programmer decide whether the effect is desired or not.

Figure 12. Program model of listing 4

Figure 12 shows a graph representation of the program in listing 4. Figure 13 represents the introduction of the method getSize as defined by the aspect Printing on the class AlertDialog. As discussed before, this introduction effectively overrides the method getSize that class AlertDialog already inherits from class DialogWindow.

Figure 13. Introduction: AlertDialog.getSize()

By defining a rule that matches such situations, the state-space exploration will show us when a state matches this rule, and allow us to trace back the introductions that led to this situation.

Figure 14. Rule matching method overriding by introductions

Figure 14 depicts such a rule for overriding methods. It looks for a combination of 2 nodes that are both classes, of which one extends the other, directly or indirectly. Again, extends+ means there may be other nodes in between, as long as there are extends-edges between them. So effectively, this selects all the superclasses of a class-node. If the parent class has a method with the same name as the child class, and the method was introduced to the child class (by an introduction), the pattern matches. This means an existing method was overridden by an introduction, which may be an unintended effect.

Figure 15. Matched rule: method override by introduction

Applying the introduction in figure 13 to the original program model in figure 12 results in the transformed program model shown in figure 15. As we can see, this transformed model matches the pattern specified in figure 14. The elements involved in the match are represented with thick lines and in a bold typeface.

4.3 Ambiguous aspect specification

Using the state-space exploration offered by the Groove toolset, we can see whether a given combination of aspects and base program can be interpreted in more than one way.

To visualize the problem, we first represent the example in listing 6 using graphs and transformations. Figure 16 represents the (relevant) structural elements of the example.

Listing 6 contains two declare parents constructs, which are both depicted in figure 17. The first rule, figure 17(a), simply selects the class named User and the interface named Persistent, and introduces an implements-edge between the two. The second rule, figure 17(b), uses an embargo-edge as part of the selection pattern. The pattern selects every class that does not have an implements-edge to the interface Persistent. If such classes are found, an implements-edge is added to the interface named SensitiveData (but, as in all examples, only if this edge was not already introduced by a prior application of the same rule). Note that, for the sake of simplicity, we left out the additional constraint of selection by package name (listing 6, line 7). Finally, the introduction of the method clear defined by the aspect SensitiveDataHandling is a rule analogous to figure 3, except with different name-nodes.

Next, we can use the Groove Simulator to explore the possible orderings of matching the transformation rules and applying the introductions. Figure 18 shows that there are different orders in which the introductions can be applied.

Node s37 represents the original program model (figure 16). Node s39 represents the state after the rule in figure 17(a) has been applied. Node s38 represents the state after the rule in figure 17(b) has been applied. As can be seen in the diagram, in state s39 there are no more applicable rules. As the class User implements the interface Persistent, the pattern that selects classes that do not implement this interface does not match anything. However, if the declaration of the interface SensitiveData is applied first (as is the case in state s38), the other rules still match and can be executed in two orders, which lead to the same state (s42) when
both have been applied. In state $s^{39}$ as well as in state $s^{42}$, no more transformations (introductions) can be applied. Therefore, these states are considered "end states", as is indicated by their different background color (gray/red). The fact that there is more than one (non-identical) end state indicates that the aspect specification can be interpreted in more than one way.

5. Addressing composition conflicts

In this section, we discuss various design alternatives of preventing and handling the different types of composition conflicts that we identified previously.

5.1 Violation of language rules

A violation of a language rule always has to be detected by a compiler, and should result in an error message. When new language constructs are introduced into an existing language, or an existing construct is modified, language developers should take care of updating the checks executed by existing language rules, if necessary.

5.2 Unintended effects of introductions

Many compilers or checking tools warn their users when they use language constructs in a way that may lead to unintended effects. However, a drawback of this solution is that warnings are also generated in cases where the behavior is in fact intended. Such warnings may eventually become meaningless to the programmer, who might then decide to ignore any such warnings.

Alternatively, some language developers might want to forbid language constructs that can lead to an unintended effect. For instance, they would allow only introduction constructs that are guaranteed to be free of potentially undesired effects. This, however, restricts the expressiveness of the language.

Another solution to this problem is to make design intentions [15] (e.g. whether the overriding of an existing element is intentional) explicitly known to the compiler. Such design intentions can be explicitly indicated, for example, by using keywords or annotations that specify whether methods may be overridden by aspects. In fact, several non-AOP languages adapted such techniques for normal (object-oriented) method overriding. As an example for using a keyword, consider the keyword virtual in C++ or C#. As an example for using an annotation, consider the annotation @Override in Java (documented in [10], see java.lang.Override). If a method is indicated as override but does not actually override a method of a superclass, compilers are required to generate an error message. An aspect language compiler could similarly generate a warning or error message if an introduced method overrides an existing one, but was not marked by such an annotation or keyword.

5.3 Ambiguous aspect specification

There are several ways to make sure an aspect specification is unambiguous. First, the aspect language could introduce an ordering mechanism, which could be used to explicitly specify the order of applying introductions (e.g. lines 7 and 13 in Listing 6).

One could argue that an application where this is necessary is probably not well-designed. In the given example (listing 6), the knowledge that "data is sensitive" is inferred from the fact that "it should not be stored persistently". However, there can obviously be many other reasons why data should not be stored persistently, other than it being sensitive. For example, some data may just represent intermediate results, which can be recalculated and hence do not need to be stored persistently. By changing the design such that the knowledge whether "data is sensitive" is not inferred in this way, the ambiguity can also be resolved. A compiler needs to detect this situation, and present the user with an error message.

Another alternative is to make the semantics of the language 'smarter' (i.e. by defining fixed implicit ordering rules within the
compiler) to resolve the composition specification in an unambiguous manner, or generate an error/warning if this is not possible. Considering our example in section 2.3, this would mean that the parent declarations are evaluated and applied in a manner that never leads to an ambiguity. For example, the compiler could always handle introductions that involve negations before any others. However, this could sometimes lead to counter-intuitive results.

In either case, whether you prefer to apply ordering rules (implicitly as part of the compiler or explicitly specified as part of the aspects), or simply to enforce the declarativeness of introductions, in both cases you probably do not want the user to supply a total ordering specification. Instead, the user should only supply one when and where it is needed, i.e. if the program would otherwise be ambiguous. Thus, a compiler should always detect cases where the programmer should have provided an ordering spec (or used a different -declarative- design), but failed to do so, leading to an ambiguous specification.

6. Discussion and future work

There are several approaches to detect conflicts related to aspects. In the examples presented in this paper we reason about aspects in the context of a base system. Some approaches pursue a more modular way of reasoning by looking only at the aspects (i.e. without considering a particular base program). However, by the very nature of aspects, they make quantifications over the base program and superimpose behavior or additional program elements in (potentially) multiple places, which makes it hard to do so.

To illustrate this, we take another look at listing 1. In this example, it would not be possible to detect the conflict by looking only at the aspects (line 5-11): the base context (line 1-3) is needed to determine that the problem exists. Without such a base context, it is not apparent that the specified patterns may match the same classes. This is especially the case because the interpretation of (AspectJ) type patterns depends on the (base) application under consideration. In this example, there is no way to discern (without a base system) whether a type pattern will match a class or an interface implemented by multiple classes, because the same kind of textual pattern (e.g. the fully qualified name of a type) is used to select interfaces, as well as classes.

Even if we assume we can somehow discern interfaces from classes while looking only at the type patterns, we still cannot generally be certain whether there is a conflict by looking only at the aspects. In this case, it depends whether the base code defines a class BusinessObject that implements the interface Persistent. We could detect this case as a potential naming conflict based on only the aspects. However, we feel that this approach would probably lead to many "false positive" detections of potential problems. This is partly caused by the limited amount of information that can be inferred by inspecting only the aspects. Because of this, we would have to assume the worst-case scenario (i.e. in the example, that the base system will define a class BusinessObject that implements the interface Persistent), even if the problem would not occur in many base programs, in practice.

In this paper we present a graph-based approach. It would be possible to achieve the same results with other formalisms, however, we feel that for our purposes, the graph-based approach is very suitable:

- Our aim is to reason about - the structure of - programs. Graphs are convenient to represent program structures; for example as an Abstract Syntax Tree.
- Since introductions change the structure of the base program, modeling introductions as program (hence graph) transforma-

tions is very intuitive. However, it can be argued that this is not necessarily the case for other types of aspect compositions.
- Graphs are relatively suitable for human viewing and reasoning. However, the selected graph representation is not optimized for human understandability, and layout becomes a critical issue for larger graphs.

Because we use an existing, well-modularized set of tools and libraries, our approach is suitable to be included in compiler technology, while writing only a minimal amount of integration code. However, when limiting oneself to specific languages and specific conflicts, it may well be possible to write more efficient "hard-coded" detection algorithms. Our primary goal is to offer a generic model to precisely describe and detect conflicts. Running the conflict detection does not take a noticeable amount of time in the (admittedly small) examples described in this paper.

As a future work, we plan to investigate whether our model can also be applied to conflicts related to advice superimposition. We expect to find some additional problems which may or may not fall under one of the categories defined in this paper. For example, several advices may modify the same resource – not even necessarily at a shared join point – in a way that breaks the program. Such semantic conflicts caused by aspects could be considered a different category from our category "unintended effects".

7. Related Work

In [18], Rinard et. al. propose a classification system for aspect-oriented programs. This system characterizes two kinds of interactions between advices and methods: (1) control flow interactions between advices and methods; (2) indirect interactions that take place as the advice and methods access object fields. The classification system is supported by program analysis tools that automatically identify classes of interactions and hence help developers to detect potentially undesired/probabilistic interactions. However, this is not intended as a conflict detection (or warning) tool as such; it is left to the interpretation of the user what is, or might be, a conflict. Also, Rinard's work focuses on the interactions among woven advice, while we focus on conflicts caused by introductions in this paper.

In [12], Kessler and Tanter identify structural conflicts similar to our proposal. To this aim, the authors propose a dependency analysis technique. This technique is based on querying a logic engine (connected to their AOP platform) to infer dependencies between what has been looked at (while interpreting the pointcuts) and what has been modified in the structural model of a program. The proposal suggests to report the detected interactions to the programmer, who should then decide about an appropriate resolution.

In [13], Kniesel and Bardey analyze aspect interference, and propose a solution to resolve it. They observe conflicts related to unintended interactions (interference), which may be caused by incorrect or incomplete weaving. They represent the weaving of aspects as conditional transformations (expressed by logic predicates). Then, they analyze these transformations for potential interference. In their approach, only the conditional transformations (i.e., the aspects) are analyzed to detect potential conflicts, independently of any base program. Such modular reasoning clearly is a big advantage when reasoning about large systems. However, as we discussed in section 6, this may in some cases lead to the detection of many potential conflicts, most of which would only occur in "worst case" base systems. Only when the aspects are considered in combination with a concrete base system, it is possible to verify whether the potential conflict really occurs in that particular case. Hence, we suggest that a combination of early 'aspects-only'
checks (especially for the category of ’certain’ problems) and more detailed analysis (as we propose in this paper) that includes the base program, can be useful.

In [11], Katz shows how to identify situations in which aspects invalidate some of the already existing desirable properties of a system. He emphasizes the importance of specifications of the underlying system. To detect interactions that invalidate desirable properties, he recommends regression verification with a possible division into static analysis, deductive proofs and aspect validation with model checking. We do not focus on checking desirable system properties, and do not require the programs to be augmented with specifications.

In [3], Douence et. al. analyze interactions between aspects written in a formally defined stateful aspect language. They model the transformations done by aspects by precisely defining the semantics of the aspect weaver. Then, they detect interactions between aspects using static analysis. When conflicts are detected, they can be resolved by extending the specification of the aspects, i.e. by supplying the desired ordering. In this paper, we use graph transformations to simulate aspect compositions, thereby also modeling part of the semantics of the language. However, we focus on the detection of conflicts related to introductions.

In [1], Aßmann and Ludwig present a weaving approach based on Graph Rewriting Systems (GRS). In this approach, aspects, join-points and weaving have well-defined and precise semantics in terms of graph-rewriting. In GRS-based aspect-oriented programming, aspect composition operators correspond to graph rewrite rules, weavings are direct derivations, and weaved programs are normal forms of the rewrite systems. In our work we focus on composition conflicts rather than weaving; we introduce a graph notation as a means to precisely model the composition, with the goal of helping the understanding and detection of composition conflicts.

In [8], we investigated the issue of inter-dependent introductions. Inter-dependent introductions can, but do not always, lead to the third category of conflicts: ambiguous aspect specifications. The primary goal of that paper is to investigate how to resolve (possibly cyclic) inter-dependencies whenever this is possible, leaving to detect those cases where this is not possible, because the introduction specifications are ambiguous.

8. Conclusion

The aim of this paper is to contribute to the understanding of aspect-oriented composition conflicts, in particular within the scope of structural composition (introductions). To this extent we propose and illustrate a systematic approach to analyze such composition conflicts in a precise and concrete manner. We employ graph-based formalisms to represent aspect-oriented programs, to represent introductions (as a graph transformation), and to express conflict detection rules. These formalisms have been introduced to deliver a precise explanation why and when some forms of composition cause a conflict, and to ensure that the categories are not overlapping. Also, the precise formulation makes it possible to perform the conflict detection fully automatic, for example as part of an aspect language compiler or consistency analyzer.

The main contributions of this paper are: (a) It proposes a general approach to the systematic and precise analysis of aspect composition conflicts. (b) It presents a classification of composition conflicts related to introductions as caused by either: violation of language rules, unintended effects, or ambiguous aspect specifications. (c) It offers a precise specification of the underlying causes for each of these conflict categories, which confirms that these are fundamentally different categories. (d) We have shown that the proposed techniques are suitable for the automatic detection of composition conflicts; we have implemented a prototype that performs automatic conflict detection for each of the three categories. It can handle AspectJ source code and works for introductions (inter-type declarations). The implementation can detect and reveal conflicts in the graph representation of the program, using the Groove tool set. (e) We discuss several alternatives of how to avoid or deal with the composition conflicts.

Although this is not the scope of this paper, we believe that this approach is general enough to be able to model other types of composition conflicts, for example related to advice weaving. We intend to use this approach to further explore the modeling and detection of composition conflicts in aspect-oriented programming.

9. Acknowledgments

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe). We would like to thank the participants in the European Workshop on Aspects in Software 2006 (EWAS) for their feedback on an early version of this paper. We also thank the authors and maintainers of the Groove tool set.

References


