An Analysis of Aspect Composition Problems

Wilke Havinga havingaw@cs.utwente.nl

Istvan Nagy nagyist@cs.utwente.nl

Lodewijk Bergmans bergmans@cs.utwente.nl

TRESE/Software Engineering group
University of Twente, The Netherlands

ABSTRACT
The composition of multiple software units does not always yield the desired results. In particular, aspect-oriented composition mechanisms introduce new kinds of composition problems. These are caused by different characteristics as compared to object-oriented composition, such as inverse dependencies. The aim of this paper is to contribute to the understanding of aspect-oriented composition problems, and eventually composition problems in a more general context. To this extent we propose and illustrate a systematic approach to analyze composition problems in a precise and concrete manner. In this approach we represent aspect-based composition mechanisms as transformation rules on program graphs. We explicitly model and show where composition problems occur, in a way that can easily be fully automated. In this paper we focus on structural superimposition (cf. intertype declarations) to illustrate our approach; this results in the identification of three categories of causes of composition problems.

1. INTRODUCTION
Composition is a key issue in computer science and particularly programming language research. There are many ways to compose software units, through so-called *composition operators*, such as method invocation, inheritance and aggregation (in the OO paradigm). In the object-oriented model, there are many known composition problems, such as the *fragile base class problem* [7], the *inheritance anomaly* [6], and problems related to the composition of crosscutting concerns [5].

The latter have been addressed by introducing new kinds of composition operators, which enable the composition of a modularly specified component with crosscutting behavior (a.k.a. an *aspect*). This paper focuses on composition in aspect-oriented programming (AOP), and investigates the issues that are involved in aspect-oriented composition. Because of the characteristics of its composition operators, such as dependency inversion, AOP introduces many new kinds of composition issues.

Many of these composition issues have been observed in the literature in the recent years, and for most of these at least partial solutions have been proposed, typically in the context of one or more AOP languages.

In this paper, we use a simple model of aspect-based composition to identify the issues that composition may cause. Specifically, we represent aspect-based composition mechanisms as transformation rules on program graphs. Using an existing toolset to explore such models, we explicitly model and show where composition problems occur. This detection mechanism can be fully automated.

This paper is a work in progress; we currently focus on composition issues related to structural superimposition (or intertype declarations). However, we expect that the model can be extended to other aspect-oriented composition mechanisms as well, such as composing base code with advices or composing advices at shared joinpoints, for instance.

2. COMPOSITION PROBLEMS
Many aspect-oriented programming languages offer various composition mechanisms to adapt the structure of the 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 problems that can occur when such structural changes are specified in the aspect definition.

2.1 Composition violates language rules
Aspect compositions may cause violations of basic language rules or assumptions. Although such situations may seem obvious, this is not always the case because of the dependency inversion introduced by aspects (i.e. AOP uses pointcuts or patterns to select elements for composition).

For example, consider the application fragment in listing 1.

```
public interface Persistent { ... }

public class BusinessObject implements Persistent { ... }

aspect PersistenceImplementation {
  Persistent+.saveChanges() { database.update(...); }
}

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

Listing 1: Conflicting method introductions

In this example, two unrelated aspects both introduce a method *saveChanges* to classes that match the specified typepattern. 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.

As a side note, in this case it would not be possible to detect this conflict by looking only at the aspects (line 5-11): the application context (line 1-3) is needed to determine that the problem exists[1].

```
public class A extends B { ... }

aspect C {
  declare parents: B extends A;
}
```

Listing 2: Cyclic inheritance (caused by an aspect)

Another example is given in listing 2. When AOP languages support a construct to change the inheritance structure, it is possible to define a circular inheritance structure (using that construct). However, most OO languages assume that the inheritance structure cannot contain cycles. It is interesting to note that AspectJ defines an additional language rule itself (which is enforced by the compiler): given that *A extends B*, it is only possible to declare *A extends C* (thus *overwriting* the superclass of A) when C is itself a subclass of B. Clearly, this rule is also broken by the given example.

These examples show how two basic object-oriented language assumptions are broken: first, a class cannot contain two distinct program elements (of the same kind) with the same name/signature. Second, inheritance cannot be circular. Note that such assumptions are in principle language specific, although the examples mentioned here probably apply to most, if not all, object-oriented languages. Usually, AOP extensions to OO languages do not change such basic language assumptions. (Even if they do, we can still define such rules over the combined language; see for example the additional language rule as introduced by AspectJ, mentioned above).

In cases like this, a compiler (or checking tool) should always give an error message.

2.2 Composition has (unwanted) side effects

Composition can affect a program in more than one way. First, it influences the program in the way as explicitly described by the composition action. In addition however, it may have (implicit) side effects. Consider the example in listing 3. Here, the aspect *Printing* introduces a method *getSize* on the class AlertDialog. However, this introduction has a side effect: it overrides a method with the same name, which the class *AlertDialog* already inherited from its parent class *DialogWindow*.

[1]It would be possible to detect *potential* conflicts by looking only at the aspects, but this could lead to many 'false positive' detections

```
public class DialogWindow {
  public Rect getSize() {
    // return window dimension..
  }; ..
}

public class AlertDialog extends DialogWindow {
  public AlertDialog(String alertMsg) {..}
}

aspect Printing {
  public Rect AlertDialog.getSize() {
    // return paper dimension..
  } ..
}
```

Listing 3: Method introduction overrides an existing method

There is a big difference between this kind of composition problem and the previous category: in this case, there is no inherent technical reason (such as violation of basic language assumptions or undefined semantics) why this composition is invalid[2]. However, the side effect of effectively overriding an existing method may be unintended and undesired. Whether this is the case depends on the requirements, i.e. whether it is the design intention to override an existing element. A compiler or checking tool cannot generally judge whether this is the case.

A compiler or checking tool should at least be able to detect such situations. Note that it can react in many different ways, which are not really the subject of this paper. To give an intuition still, we mention how similar situations are usually handled in existing languages. First, a compiler could just emit a warning to the programmer. Second, some language developers might want to forbid the occurrence of such situations at all and allow only application of introductions (or advices) when this is guaranteed to be free of possibly undesired side effects. A third solution is to make design intentions [9] (e.g. whether an override is intentional) explicitly known to the compiler, for example by using keywords or annotations that specify whether methods may be overridden by aspects. Several non-AOP languages use similar constructs for normal (object-oriented) method overring. Consider for example the *virtual* keyword in C++, or the *virtual/override* keywords in C#. In C#, a subclass cannot override methods inherited from its parent class, unless the new method is declared using the keyword *override*, and the existing (parent) method was declared using the keyword *virtual*.

2.3 Composition specification is ambiguous

A third kind of composition problem is caused by composition specifications referencing *and* modifying the same model.

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.

```
declare parent: BusinessObject implements PersistentRoot;

```

[2]In fact, the example in listing 3 works fine in AspectJ.

```
 3 pointcut persistence():
 4   execution(* PersistentRoot+.*(..));
```

Listing 4: Pointcut depends on an introduction

Listing 4 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 situations where the code is ambiguous. A simple example of such a case is given in listing 5.

```
 1 public interface Persistent { ... }
 2 public class User implements Persistent { ... }
 3 public class Group { ... }
 4
 5 aspect SensitiveDataHandling {
 6   !Persistent+.clear() { ..clear all values.. }
 7 }
 8
 9 aspect PersistenceHandling {
10   declare parents: Group implements Persistent;
11 }
```

Listing 5: Ambiguous introduction

In this example, a method *clear* is introduced on all classes that are supposed to *not* store their data persistently (line 6). However, another aspect may declare specific classes to be persistent (line 10). In this case, the specification is ambiguous, because it is unclear whether or not the method *clear* should be introduced to the class *Group*. We observe that the order of applying the introductions leads to different results. For a more detailed explanation of these and similar issues, see [2].

3. MODELING ASPECT COMPOSITION

In the previous section, we identified several problem categories related to aspect composition. To facilitate a precise analysis of such problems, we first present a concrete model to represent aspect-based compositions.

A distinguishing feature of aspect-oriented composition is the way in which it selects how elements of a program should be composed. In regular (e.g. object-oriented) composition mechanisms, a component often explicitly depends on another component by directly referring to it. For example, a business object *A* may contain *calls* to a persistence framework *B* (composition: A *calls* B). However, when using aspect-oriented composition mechanisms, this dependency is inverted: the persistence framework can be *superimposed* on objects that match certain criteria (composition: B *is superimposed* on A).

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 [8], chapter 2 for a detailed reference model of AOP constructs).

We believe that the simple representation above will be sufficient to model most aspect-oriented composition mechanisms. In the remainder of this paper, we use this model of composition for the analysis of structural introductions in AOP languages, in a way that could also be used in an automated conflict detection tool. Although it is not the focus of this paper, we are investigating whether the same approach can also be used to analyze other types of aspect-based compositions, such as advice composition at shared join points - which seems promising.

3.1 Modeling composition using graphs

As explained above, a composition specification consists of two important parts: a selection and an action. To analyze such composition specifications, we need to model the elements (i.e. a concrete program model) to which the composition is applied[3]. For example, if we want to analyze introductions on static program structure, it makes sense to take an abstract syntax tree (AST) representation of the program model–as it contains all the elements relevant to static introductions.

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

Figure 1 shows a graph-based representation of the structure of the program in listing 1. Each program element (e.g. method, class, ..) is mapped to a node with a unique identity (node label), e.g. method1, method2, class1. These 'generated' labels have no other meaning but to uniquely identify that node. Each such program element node has–at least–the following two outgoing edges: one edge labeled *isa*, pointing to a node that represents the kind of program element, 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. The same applies to kinds and *isa* edges. All other edges model relations between program elements in a given language model; e.g. class-nodes may have *implements* relations (edges) to interface nodes, and/or *hasMethod* edges to method nodes, etc.

The left-hand side of figure 1 represents the AST of the base program in listing 1. It contains only 2 program element nodes, labeled *iface1* 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 *iface1* (Persistent).

Similarly, the right-hand side shows a representation of the aspects defined in listing 1. Note that both aspects have *hasMethod*-edges to distinct method-nodes, although both method-nodes point to the same *name*-node (saveChanges).

Figure 1 only includes the structural elements relevant to this example. In addition, we also need to model the composition specifications defined within these aspects. To this end, we use a single graph that specifies the selection (type

[3]Alternatively, we could detect *potential* conflicts based on only the composition specifications, independent of any application. This option is not explored here.

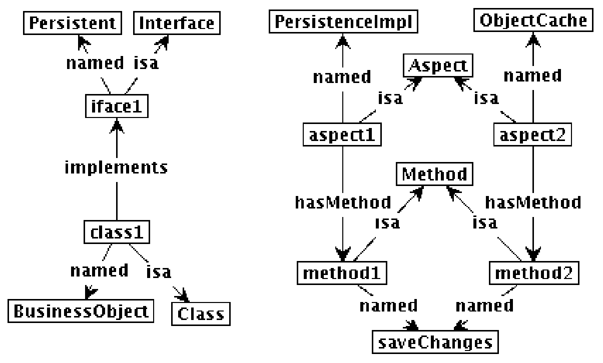

Figure 1: Graph representation of the program in listing 1

pattern) as well as an action (structural introduction). Figure 2 shows the composition specification rule that corresponds to line 6 of listing 1.

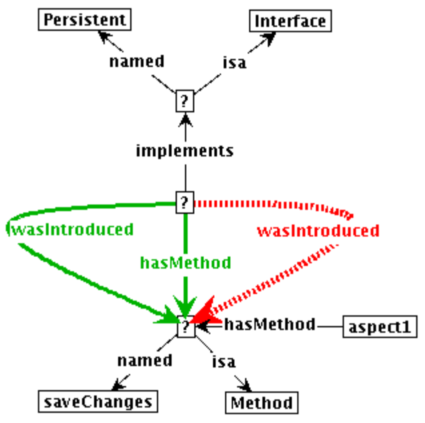

Figure 2: Introduction: Persistent+.saveChanges()

In figure 2, all the nodes and edges with solid (black) lines together specify the selection pattern. Nodes or edges marked with a question mark match any value. This example specifies the type pattern *Persistent+* (i.e. all classes implementing the interface *Persistent*). It matches a program element node that has a *named*-edge to a node labeled *Persistent* and an *isa*-edge to a node labeled *Interface*. We are interested in all program element nodes that have an *implements*-edge to this node. Also, we select a node named *saveChanges*, which is a *Method*, and is contained (*hasMethod*) by the program element *aspect1*, which is the unique identifier of the aspect that specifies this introduction. If the specified pattern can be found in the program graph, this rule can be applied.

The 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 a *hasMethod*-edge between the selected class(es) and the method defined within the aspect. Finally, the dotted (red) edge specifies an *embargo*, which means an edge with the specified label must *not* exist between nodes to be selected. In this case, we specify that a *wasIntroduced* edge must not exist, and as part of the action, we add such an edge. This prevents the same introduction from being applicable 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.

The program model of listing 1 matches this pattern, as shown in figure 3–the nodes and edges involved in the match are in bold-face in this figure.

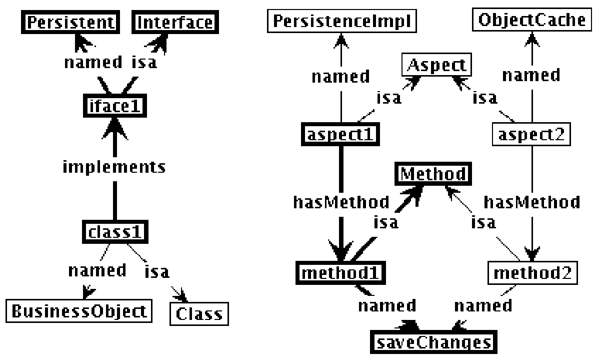

Figure 3: Selection of elements that match

After application of the rule in figure 2, the program model from figure 1 is transformed into a new "state", as depicted in figure 4. The edges between nodes *class1* and *method1* have been added by application of the introduction rule.

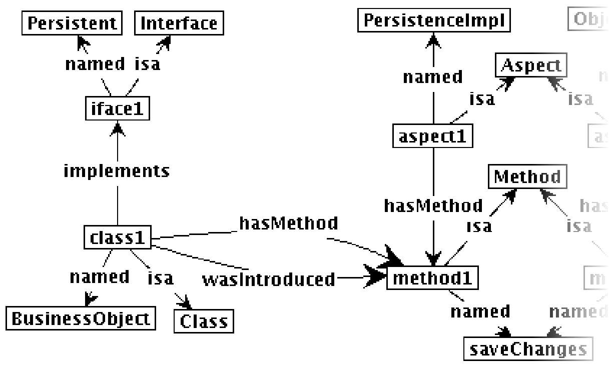

Figure 4: Introduction applied to the program from figure 1 (partial diagram)

3.2 The Groove tool set

The diagrams above have been created using the existing Groove (Graphs for Object-Oriented Verification) tool set [1, 10]. Given a graph-based (program) representation and graph-based transformation rules, Groove can determine which transformation rules can be applied to the current state; resulting in a new state (modified program representation). Groove can explore the complete state-space, i.e. all combinations and 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 isomorphism detection can be done in polynomial time in most cases. We reuse the analysis capabilities of the tool set; our models (program representation and transformations) are however completely unrelated to those used in the Groove project itself.

To demonstrate the workings of the tool, we complete the representation of listing 1, including the application of aspect-oriented composition constructs (here, intertype declarations).

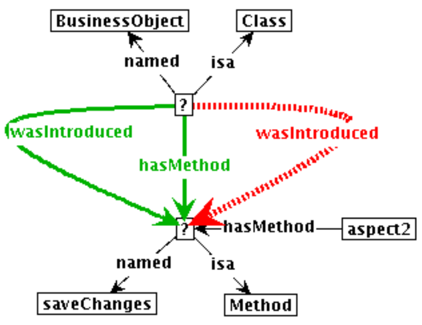

Figure 5: Introduction(2): BusinessObject.saveChanges()

Figure 5 represents the selection pattern in listing 1, line 10. It is very similar to figure 2, but directly selects the class *BusinessObject* instead of referring to an interface.

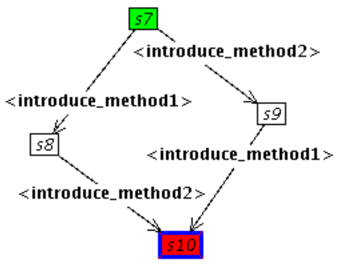

Figure 6: State-space of the compositions in listing 1

Figure 6 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 2. Node *s8* refers to the state of the program model as in figure 4. The final state *s10* is (partially) shown in figure 7. 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 an isomorphic 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. COMPOSITION PROBLEM ANALYSIS

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

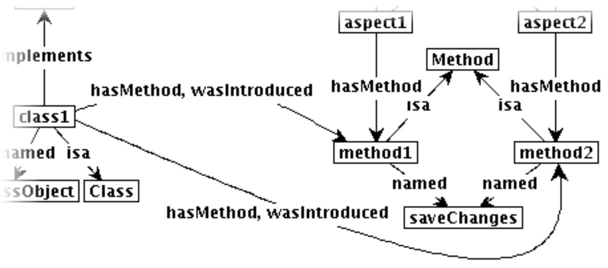

Figure 7: Final program model after applying introductions (partial diagram)

4.1 Composition violates language rules

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 8 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 (i.e. non-equal) nodes.

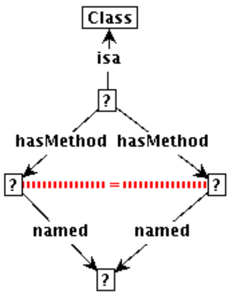

Figure 8: Violation rule: Naming conflict/double definition

Figure 9 shows that this rule indeed matches in the final state of our previous example (see figure 7). We can see exactly which elements are involved with the conflict. Also, we can trace back (by looking at figure 6) which combination of introductions led to the matching of this rule, and are thus involved in causing the conflict.

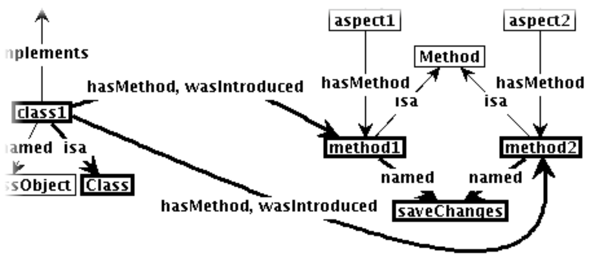

Figure 9: Program matches the violation rule in figure 8 (partial diagram)

It is possible to define such rules for all kinds of language assumptions - often, they can be found in the language specification. For example, figure 10 depicts a rule that matches circular inheritance between classes - the other type of language assumption violation mentioned in section 2.1. Note that the edge labeled *extends+* will match one or more such edges (between arbitrary nodes). Due to space constraints, we do not include a full representation of this example here.

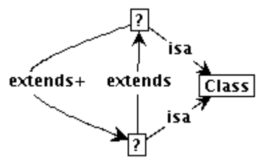

Figure 10: Violation rule: circular inheritance

4.2 Composition has (unwanted) side effects

To detect undesired kinds of side effects, 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 (i.e. which AO composition(s) caused it), which could help a programmer decide whether the side effect is desired or not.

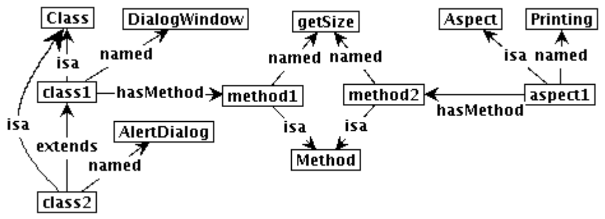

Figure 11: Program model of listing 3

Figure 11 shows a graph representation of the program in listing 3. Figure 12 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*.

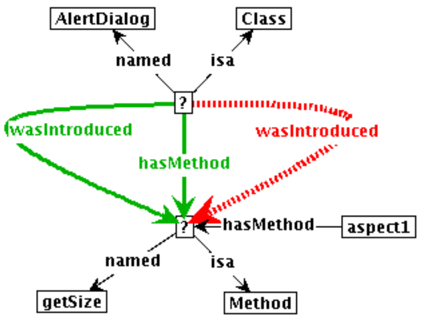

Figure 12: Method introduction: AlertDialog.getSize() {..}

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.

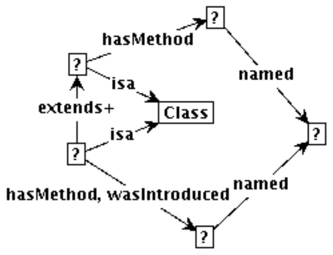

Figure 13: Rule matching method overrides by introductions

Figure 13 depicts such a rule for method overrides. It looks for a combination of 2 nodes that both are classes, and one extends the other, directly or indirectly (*extend+* means there may be other nodes in between, as long as there are *extend*-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 AOP composition action), the pattern matches. This means an existing method was overriden by the introduction.

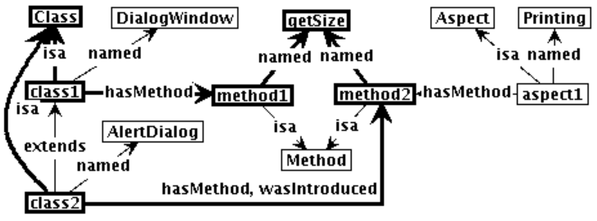

Figure 14: Matched rule: method override by introduction

Applying the introduction in figure 12 to the original program model in figure 11 results in the transformed program model shown in figure 14. As we can see, this transformed model contains the pattern specified in figure 13. The elements involved in the match are represented with thick lines and in a bold typeface.

4.3 Composition specification is ambiguous

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 5 using graphs and transformation. Figure 15 represents the (relevant) structural elements of the example.

Listing 5 contains two introduction specifications, which are represented by figure 16 and 17. The first rule is straightforward: it simply selects the class named *Group* and the interface named *Persistent*, and adds an *implements*-edge between the two.

The second rule (also) uses an embargo-edge as part of the selection pattern. The pattern selects every class that does

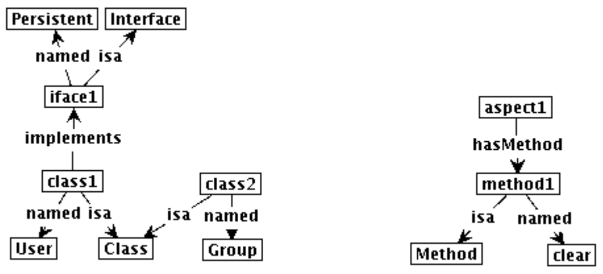

Figure 15: Program model of listing 5

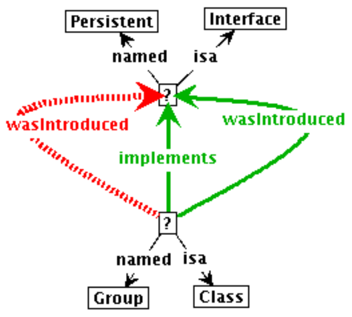

Figure 16: Declare parents: Group implements Persistent

not have an *implements*-edge to the interface *Persistent*. If such classes are found, a *hasMethod*-edge is added to the method named *clear* (but, as in all examples, only if this edge was not already introduced by a prior application of the same rule).

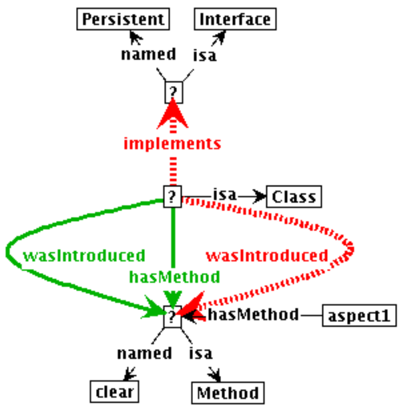

Figure 17: Method introduction: !Persistent+.clear() {..}

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

Node *s29* represents the original program model (figure 15). Node *s30* is the state after the rule in figure 16 has been applied. Node *s32* represents the state after the rules in figure 17 and 16 have been applied (in that order). In this case, the two paths lead to different end results (i.e. states where no more transformation rules can be applied to the program model). The cause is that when we apply the rule *declare_persistent*, it adds an *implements*-edge that prevents the rule *introduce_not_persistent* (figure 17) from matching. Without any constraints on the order of applying the transformation rules, it is clear that the program is ambiguous.

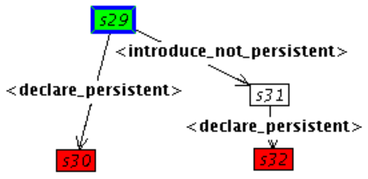

Figure 18: Alternative orderings of applying introduction rules

5. RELATED WORK

In [11], 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. In our work, we propose a classification system that characterizes structural conflicts (structural interactions) that can potentially occur when structural composition mechanisms are used in aspect-oriented languages. Both our and Rinard's classification systems are supported by program analysis tools that automatically identify classes of interactions and helps developers to detect potentially undesired/problematic interactions. The main difference between our and Rinard's work is that Rinard's work focuses on behavioral (i.e. control and data flow related) interactions, while we focus on structural conflicts and interactions in this work.

In [4], 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 (by pointcuts) and 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 [3], 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. In our work, we do not assume the aspects to be woven need to be augmented with specifications; i.e. we do not focus on checking the desirable properties of a system. Instead, we focus on general conflicts and interactions that are caused by the violation of basic language assumptions and interdependencies in the weaving specifications.

6. CONCLUSION

The aim of this paper is to contribute to the understanding of aspect-oriented structural composition problems, and eventually composition problems in a more general context. To this extent we have proposed and illustrated a systematic approach to analyze composition problems in a precise and concrete manner. We have employed graph-based formalisms to represent aspect-oriented programs, to represent structural superimposition (as a transformation), and to express consistency rules that can be automatically verified to indicate composition problems. These formalisms have been introduced to deliver a precise explanation why and when some forms of composition are a problem, and to ensure that the categories are not overlapping. Currently, we do not claim that the identified categories are complete, however.

A summary of the main contributions of this paper: (a) A classification of structural superimposition problems as caused by (1) violation of language assumptions (2) side effects (3) an ambiguous specification. (b) A general approach to the systematic and precise analysis of composition problems. (c) The proposed techniques are suitable for the automatic detection of composition problems; currently, only the transformation of a program to its graph representation is done manually (this includes the mapping of pointcuts/-patterns to transformation rules). The actual checking is already working automatically within the Groove toolset. Note that this assumes that rules about the language assumptions and (undesired kinds of) side effects are available. For new languages, these will have to be reconsidered.

We plan to extend this work into other categories of aspect composition, such as regular behavioral superimposition (e.g. before and after advice weaving). As we do so, we particularly aim to identify common underlying causes that are universal for many kinds of composition techniques. We are also interested in investigating a category of problems that is caused by interference among aspects, to see whether we can identify these problems independent of any concrete base program.

7. ACKNOWLEDGEMENT

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008

8. REFERENCES

[1] Groove homepage - http://groove.sf.net.

[2] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and Resolving Ambiguities caused by Inter-dependent Introductions. In *Proceedings of 5th International Conference on Aspect-Oriented Software Development*, pages 214–225. ACM, 2006.

[3] S. Katz. Diagnosis of harmful aspects using regression verification. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, Mar. 2004.

[4] B. Kessler and É. Tanter. Analyzing interactions of structural aspects. In *Workshop on Aspects, Dependencies and Interactions @ECOOP 2006*, July 2006.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[6] S. Matsuoka, K. Taura, and A. Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 109–126, New York, NY, USA, 1993. ACM Press.

[7] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer, 1998.

[8] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* PhD thesis, University of Twente, June 2006.

[9] I. Nagy, L. Bergmans, W. Havinga, and M. Aksit. Utilizing design information in aspect-oriented programming. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Gergmany, Sep 2005. Gesellschaft für Informatik (GI).

[10] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[11] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FSE)*, pages 147–158. ACM, Oct. 2004.