EWAS 2006

3rd European Workshop on Aspects in Software

University of Twente, Enschede, the Netherlands
August 31, 2006

Günter Kniesel (Editor)

Technical Report IAI-TR-2006-6
ISSN 0944-8535

Institut für Informatik III
Universität Bonn

Foreword

EWAS.06, the third European Workshop on Aspects in Software, follows the tradition of its two predecessor workshop series organized independently by the German "Gesellschaft für Informatik", Belgian and Dutch research groups. It focuses on providing AOSD researchers and practitioners with an interactive forum for fostering novel ideas and turning submitted material into conference-ready papers.

Improving submitted papers is emphasized by highly interactive "writer's workshop" sessions that are designed to give authors constructive feedback regarding their paper's content and presentation. In addition, the EWAS reviewers are typically asked not to confine their comments to criticizing weaknesses but to provide their opinion also on the strengths of a paper and to give advice on how to improve it. This requires more time and consideration than other review formats and therefore EWAS would not be possible without the commitment of many expert reviewers. We are deeply thankful to all the program committee members for the time and effort they put again into their thoughtful reviews.

After EWAS 2005 two of the discussed papers were revised, submitted to the AOSD'06 conference and accepted for the main technical programme. We believe this is a good indication of the usefulness of the EWAS format.

The EWAS.06 programme contains four full papers (included in these proceedings), a tool demonstration (abstract included) and two "early papers" that where considered worth discussing at the workshop but not yet mature enough for inclusion in the proceedings. The early papers are available at the EWAS.06 website http://janus.cs.utwente.nl:8000/twiki/bin/view/EIWAS2006.

An overview of current and future EWAS workshops, as well as its predecessor AOSD-GI Workshops and Belgian-Dutch AOSD Workshops is available at the AOSD-GI portal site: http://www.cs.uni-bonn.de/~gk/aosd/.

Günter Kniesel
EWAS.06 programme chair

Workshop Schedule

10.00 Registration and welcome coffee

10.30 Welcome and mutual introduction

Full papers session (Chair: Günter Kniesel)

11.00 An Analysis of Aspect Composition Problems
Presented by Wilke Havinga (writer's workshop session: 20 +20 min.)

11.40 Separating Constraints in Validated Model Transformations
Presented by Gergely Mezei (writer's workshop session: 10 +20 min.)

12.10 Made in MAKAO
Presented by Bram Adams (demonstration session: 30 min.)

12.30 Lunch and poster session

Early papers session (Chair: István Nagy)

14.00 GAP: Generic Aspects for PHP
Presented by Sebastian Bergmann (writer's workshop session: 10 +20 min.)

14.30 Evolvable Integration of Activities with Statecharts
Presented by Gurcan Gulesir (writer's workshop session: 10 +20 min.)

15.00 Coffee break and poster session

Full papers session (Chair: Lodewijk Bergmans)

15.30 On Dynamically-Scoped Crosscutting Mechanisms
Presented by Günter Kniesel (writer's workshop session: 10 +20 min.)

16.00 Introducing mixin layers to support the development of context-aware systems
Presented by Brecht Desmet (writer's workshop session: 10 +20 min.)

Invited talk and panel

16.30 How to get your paper accepted at AOSD
Invited talk by Mehmet Akşit (presentation and discussion)

17.15 Wrap up and closing

Table of Contents

Papers

An Analysis of Aspect Composition Problems 1
Wilke Havinga, István Nagy, Lodewijk Bergmans
University of Twente, The Netherlands

Separating Constraints in Validated Model Transformations 9
László Lengyel, Tihamér Levendovszky, Gergely Mezei, Hassan Charaf
Budapest University of Technology and Economics, Hungary

On Dynamically-Scoped Crosscutting Mechanisms 17
Éric Tanter
University of Chile, Chile

Introducing mixin layers to support the development of context-aware systems 23
Brecht Desmet, Jorge Vallejos, Pascal Costanza
Vrije Universiteit Brussel, Belgium

Demonstration Abstracts

Made in MAKAO 31
Bram Adams
Ghent University, Belgium

Organizers

- Lodewijk Bergmans, University of Twente, The Netherlands
- Mariano Ceccato, ITC-irst, Italy
- István Nagy, University of Twente, The Netherlands

Program Comittee

- Günter Kniesel, University of Bonn, Germany (PC Chair)

- Alexandre Bergel, Trinity College, Ireland
- Lodewijk Bergmans, University of Twente, The Netherlands
- Mireille Blay-Fornarino, ESSI, France
- Johan Brichau, Vrije Universiteit Brussels, Belgium
- Mariano Ceccato, ITC-irst, Italy
- Siobhán Clarke, Trinity College, Ireland
- Robert Filman, Google Inc., USA
- Maja D'Hondt, Universit de Lille, Villeneuve d'Ascq, France
- Jan Hannemann, University of Tokio, Japan
- Stephan Herrman, Technical University Berlin, Germany
- Shmuel Katz, Technion Haifa, Israel
- Hidehiko Masuhara, University of Tokio, Japan
- Kim Mens, Université Catholique de Louvain, Belgium
- Mira Mezini, Technical University Darmstadt, Germany
- Oege De Moor, Oxford University, UK
- István Nagy, University of Twente, The Netherlands
- Tobias Rho, University of Bonn, Germany
- Tom Tourwe, CWI Amsterdam, The Netherlands
- Jianjun Zhao, Shanghai Jiao Tong University, China

Contact

- Organizers and PC chair: ewas@ewi.utwente.nl
- Proceeding reprints: gk@cs.uni-bonn.de

Papers

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.

```
1  public interface Persistent { ... }
2
3  public class BusinessObject implements Persistent { ... }
4
5  aspect PersistenceImplementation {
6    Persistent+.saveChanges() { database.update(...); }
7  }
8
9  aspect ObjectCache {
10   BusinessObject.saveChanges() { cache.storeValue(...); }
11 }
```

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].

```
1 public class A extends B { ... }
2
3 aspect C {
4   declare parents: B extends A;
5 }
```

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

```
1  public class DialogWindow {
2    public Rect getSize() {
3      // return window dimension..
4    }; ..
5  }
6
7  public class AlertDialog extends DialogWindow {
8    public AlertDialog(String alertMsg) {..}
9  }
10
11 aspect Printing {
12   public Rect AlertDialog.getSize() {
13     // return paper dimension..
14   } ..
15 }
```

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.

```
1 declare parent: BusinessObject implements PersistentRoot;
2
```

[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.

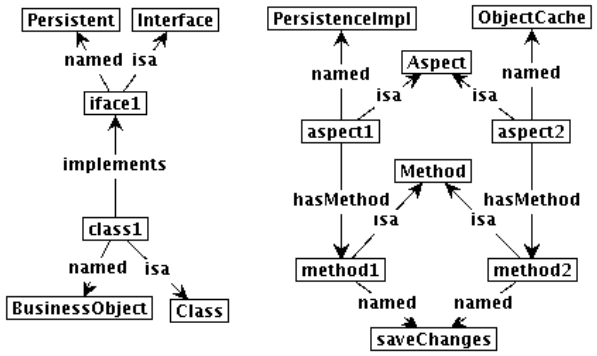

Figure 1: Graph representation of the program in listing 1

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

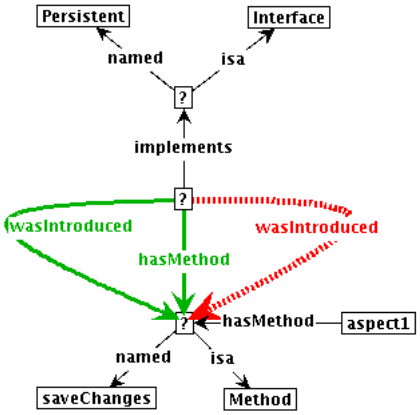

Figure 2: Introduction: Persistent+.saveChanges()

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.

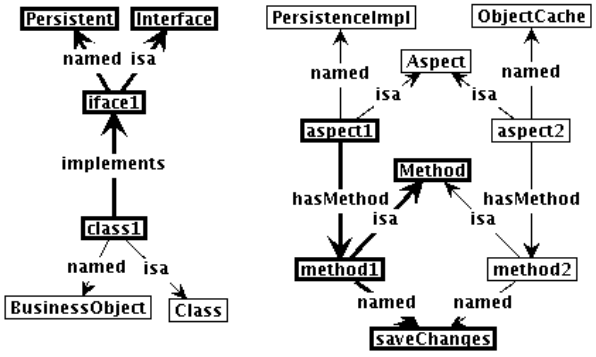

Figure 3: Selection of elements that match

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.

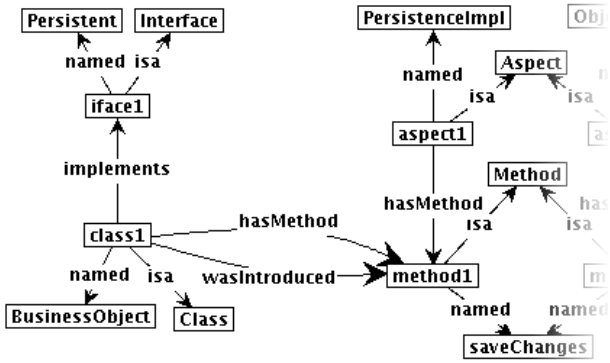

Figure 4: Introduction applied to the program from figure 1 (partial diagram)

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).

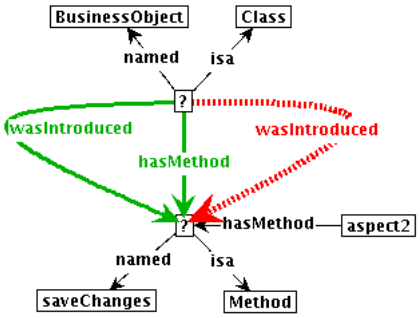

Figure 5: Introduction(2): BusinessObject.saveChanges()

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.

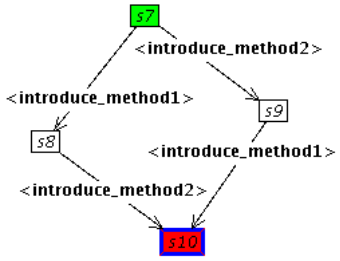

Figure 6: State-space of the compositions in listing 1

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.

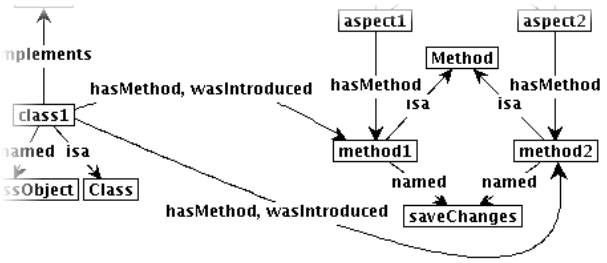

Figure 7: Final program model after applying introductions (partial diagram)

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.

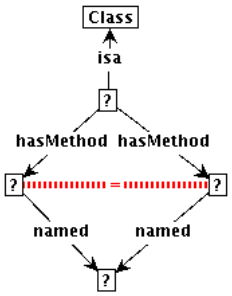

Figure 8: Violation rule: Naming conflict/double definition

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.

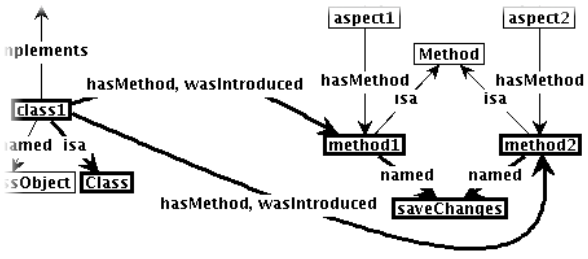

Figure 9: Program matches the violation rule in figure 8 (partial diagram)

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.

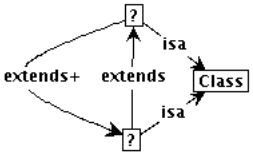

Figure 10: Violation rule: circular inheritance

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.

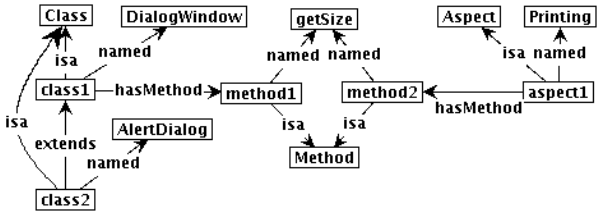

Figure 11: Program model of listing 3

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*.

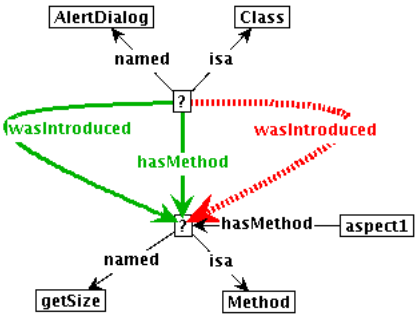

Figure 12: Method introduction: AlertDialog.getSize() {..}

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.

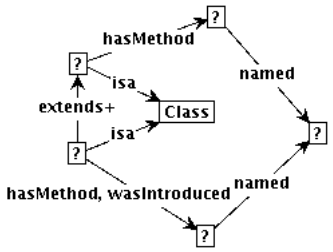

Figure 13: Rule matching method overrides by introductions

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.

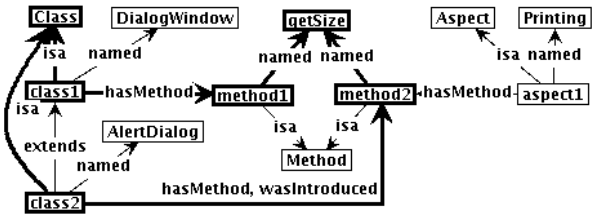

Figure 14: Matched rule: method override by introduction

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

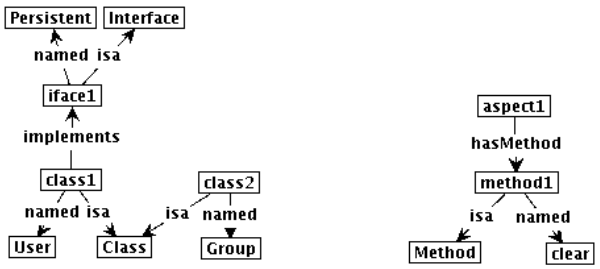

Figure 15: Program model of listing 5

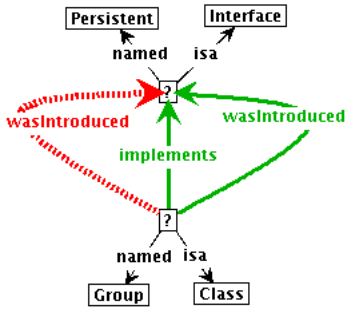

Figure 16: Declare parents: Group implements Persistent

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).

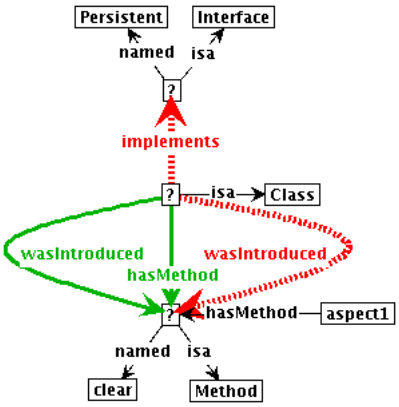

Figure 17: Method introduction: !Persistent+.clear() {..}

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

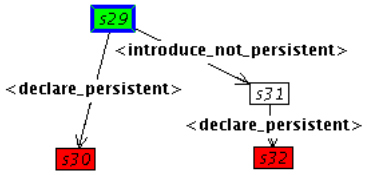

Figure 18: Alternative orderings of applying introduction rules

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.

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.

Separating Constraints in Validated Model Transformations

László Lengyel, Tihamér Levendovszky, Gergely Mezei, Hassan Charaf
Budapest University of Technology and Economics
Goldmann György tér 3.
1111 Budapest, Hungary
{lengyel, tihamer, gmezei, hassan}@aut.bme.hu

ABSTRACT
Model-based approaches in software development are widely recognized as a potential way of increasing productivity in software engineering. Model-based development is driven by model transformations. There is a demand for researching the ways how model transformation can become more flexible, efficient, and highly-configurable as well as validated. Extensive research of graph transformation provides a strong formal background for graph rewriting-based model transformation. In the tool Visual Modeling and Transformation System (VMTS), metamodel-based transformation rules enable assigning OCL constraints to model transformation rules that facilitate constraint-driven online validated model transformations. Using aspect-oriented methods, this paper introduces the separation of the refining and validation constraints that crosscut the transformation rules, and discusses the constraint realization possibilities with negative and positive application conditions (NAC, PAC). As a result of the constraint-driven validation, we can not only ensure the validation of the whole transformation, but the transformation becomes more modular and applicable in larger projects as well.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques;

General Terms
Algorithms, Design and Languages.

Keywords
Aspect-Oriented Constraints, Constraint Separation, Constraint Weaving, Model Transformation.

1. INTRODUCTION
OMG's Model-Driven Architecture (MDA) [1] emphasizes the use of models at all stages of system development. It has placed model-based approaches to software development into focus. MDA offers a standardized framework to separate the essential, platform-independent information from the platform-dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), one or more platform-specific models (PSM), including complete implementations, one on each platform that the application developer decides to support. The platform-independent artifacts are mainly UML and other software models containing enough specification to generate the platform-dependent artifacts automatically by model compilers.

Transformations appear in many different situations in a model-based development process. A few representative examples are as follows. (i) Refining the design to implementation; this is a basic case of PIM/PSM mapping. (ii) Aspect weaving; the integration of aspect models/code into functional artifacts is a transformation on the design. (iii) Analysis and verification; analysis algorithms can be expressed as transformations on the design.

One can conclude that transformations in general play an essential role in model-based development, thus, there is a need for highly reusable model transformation tools. The approach presented here uses graph rewriting-based visual model transformation. Models can be considered special graphs, which contain nodes and edges between them. This formal background facilitates treating models as labeled graphs and to apply graph transformation algorithms to models using graph rewriting. To define the transformation rules precisely and support the validated model transformation beyond the structure of the visual models, additional constraints must be specified, which ensure the correctness of the attributes, or other properties can be enforced. Using Object Constraint Language (OCL) [2] constraints provides a solution for these issues. The use of OCL as a constraint and query language in modeling is found to be simple and powerful.

Current work fits into the research related to the constraint-driven online validated model transformation. Applying aspect-oriented methods the main contribution of this paper is the separation of the *refining* and *validation* constraints. As a result of the constraint-driven online validation, we can ensure the validation of not only the individual transformation rules, but the whole transformation. Furthermore, the separation of the *refining* and *validation* constraints facilitates that the transformation becomes more modular, and its elements, such as transformation rules and constraints, become reusable.

The rest of this paper is organized as follows. Section 2 provides background information and introduces a graph transformation-based model transformation system (Visual Modeling and Transformation System, VMTS [3] [4]). Using the features of this system, Section 2.1 presents the principles of the online validated model transformation and Section 2.2 introduces aspect-oriented constraint management. Using a real word transformation example, Section 3 (i) discusses the separation of the *refining* and *validation* constraints, and (ii) presents how transformations become modular as a result of the constraint separation, furthermore, (iii) it deals with the reusable transformation elements. Section 4 gives related work and discusses how to express constraints with negative and positive application conditions. Moreover, it provides algorithms to create NACs and

PACs from OCL constraints and construct left from right application conditions. Finally, conclusions are provided.

2. BACKGROUND

Graph rewriting [5] [6] is a powerful technique for graph transformation with a strong mathematical background. The atoms of graph transformations are rewriting rules, each rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of LHS in the graph the rule being applied to (host graph), and replacing this subgraph with RHS.

Visual Modeling and Transformation System (VMTS) is an n-layer metamodeling environment which supports editing models according to their metamodels, and allows specifying Object Constraint Language (OCL) constraints. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary").

Also, VMTS is a model transformation system which transforms models, using graph rewriting techniques. Moreover, the tool facilitates the verification of the constraints specified in the transformation rule during the model transformation process.

In VMTS, LHS and RHS of the transformation rules are built from metamodel elements. This means that an instantiation of LHS must be found in the host graph, instead of the isomorphic subgraph of LHS. Both sides of the transformation rule use formalism similar to that of the UML class diagram [3].

The Object Constraint Language is a formal language for the analysis and design of software systems. It is a subset of the UML standard, and OCL allows software developers to write constraints and queries over object models.

In our experience, rewriting rules can be made more relevant to software engineering models if the metamodel-based specification of the transformations allows assigning OCL constraints to the individual transformation rules. This technique facilitates a natural representation for multiplicities, multi-objects, and assignments of OCL constraints to the rules with a syntax close to the UML notation.

VMTS facilitates a refined description of the transformation rules. When the transformation is performed, the changes are specified by the RHS and *internal causality* relationships defined between the LHS and the RHS elements of a transformation rule. Internal causalities can express the modification or removal of an LHS element, and the creation of an RHS element. XSLT scripts can access to the attributes of the objects matched to the LHS elements, and produce a set of attributes for the RHS element to which the causality points.

Classical graph grammars apply any production that is feasible. This technique is appropriate for generating and matching languages, but model-to-model transformations often need to follow an algorithm that requires a stricter control over the execution sequence of the rules, with the additional benefit of making the implementation more efficient. Since VMTS is a visual approach, it also uses graphical notation for control flow: stereotyped UML activity diagrams. VMTS Visual Control Flow Language (VCFL) is a visual language for controlled graph rewriting and transformation, which supports the following constructs: sequencing transformation rules, branching with OCL constraints, hierarchical rules, parallel execution of the rules, and iteration [7].

In VCFL, if a transformation rule fails and the next element in the control flow is a decision object, then it can provide the next branch based on the OCL statements and the value of the *SystemLastRuleSucceed* variable. If no decision can be found, the control is transferred to the parent rule, if there is no parent rule, the transformation terminates with error.

The interface of the transformation rules allows the output of one rule to be the input of another rule (parameter passing), in a dataflow-like manner. This is used to sequence expression execution. In VCFL, this construction is referred to as *external causality*.

2.1 Validated Model Transformation

At the implementation level, system validation can be achieved by testing. Various tools and methodologies have been developed to assist in testing the implementation of a system (for example, unit testing, mutation testing, and white/black box testing). However, in case of model transformation environments, it is not enough to validate the transformation engine. The transformation specification should also be validated.

There are only few and not complete facilities provided for testing offline transformation specifications in an executable style. Also, there are several model transformation environments that provide well-defined interfaces that allow implementing transformations using an optional object-oriented programming language. Related to the expected output, there are only a few constraints that can be guaranteed by these transformations. The transformation should be tested: not only the syntactical but the semantical correctness is required as well. In fact, testing requires huge efforts, and even after executing the tests, it is not guaranteed that the transformation produces the expected output for all valid input. The reason is that there is no real possibility that the testing covers all the possible cases.

In general, the offline validation of model transformations is algorithmically hard, therefore, validation should be performed online: during the model transformation process. There is a need for a solution that can validate model transformation specifications: online validated model transformation that guarantees if the transformation finishes successfully, the generated output is valid, which means that it satisfies a set of high-level constraints.

A *precondition* assigned to a transformation rule is a Boolean expression that must be true at the moment when the transformation rule is fired. Similarly, a *postcondition* assigned to a transformation rule is a Boolean expression that must be true after the completion of a transformation rule. If a precondition of a transformation rule is not true then the transformation rule fails without being fired. Also, if a postcondition of a transformation rule is not true after the execution of the transformation rule, the transformation rule fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the transformation rule, and an OCL expression in RHS is a postcondition to the transformation rule. A transformation rule can be fired if and only if all conditions enlisted in LHS are true. If a transformation rule

finished successfully, all conditions enlisted in RHS must be true [8].

There are three types of high-level constraint constructs: *validation, preservation,* and *guarantee* [4], which can be used during the rewriting process. A transformation rule *S validates* a property *P*, when the following condition always holds: if a property *P* was true before the rule *S* it remains true after the execution of the rule *S*, and if *P* is false, the rule *S* fails. A rule *S preserves* a property *P*, when the following condition always holds: if a property *P* was false (true) before the rule *S* it remains false (true) after the execution of the rule *S*. A transformation rule *S guarantees* a property *P*, when the following condition always holds: if a property *P* was true before the rule *S* it remains true after the execution of the rule *S*, and if *P* is false, the rule *S* changes property *P* to true.

The principles of metamodel-based model transformation in VMTS are depicted in Fig. 1 and 2. Fig. 1 describes that the transformation is specified by the VCFL control flow model that defines the exact execution order of the transformation rules. The input model is accompanied by the input metamodel, and the output model is associated with the output metamodel.

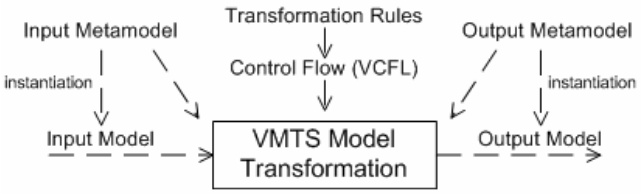

Figure 1. Principles of VMTS metamodel-based model transformation.

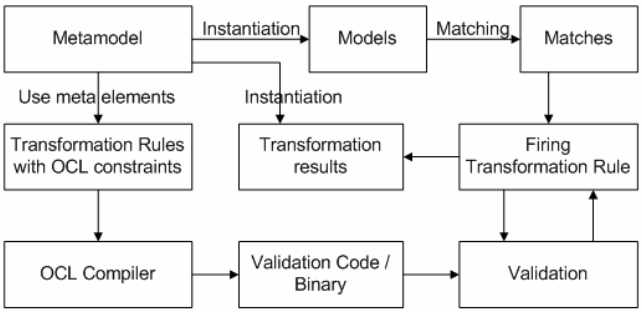

Figure 2. Constraint validation during transformation.

Recall that LHS and RHS of a transformation rule are built from the metamodel elements. LHS and RHS can use different metamodels (Fig. 2). The transformation rule contains OCL constraints. The transformation uses matches found by the matching process and the compiled binary to validate the constraints on the matched parts of the input model. The transformation generates the transformation result if and only if a match satisfies the constraints (preconditions). Moreover, if and only if the transformation result satisfies the postconditions, then the rule is successful. In Fig. 1, the transformation result is an instance model of the output metamodel.

The constraints assigned to the transformation rules validate the requirements expected from the transformation rules. As it has been discussed, after a successful rule execution, the conditions hold and the output is valid. This cannot be achieved without constraints.

2.2 Aspect-Oriented Constraint Management

Constraints propagated to transformation rules means the fundamentals of the online validated model transformation. Often, the same constraint is repetitiously applied in many different places in a transformation and crosscuts it. We have found that the source of our transformation problems (maintainability, reusability) is often related to the lack of support for modularizing crosscutting concerns. It would be beneficial to describe a common constraint in a modular manner and designate the places where it will be applied. In [9] and [10] an aspect-oriented method is presented to solve the problem of crosscutting constraints. As we have extended our metamodel-based visual transformation language with an aspect-oriented constraint management approach, it was observed that the maintainability and understandability of our transformation rules have been increased along with the attached constraints. As far as the correctness of the transformation is considered, the understandability of the constraints is crucial, since the verification is still left to the intuition and the experience of the designer.

Using aspect-oriented constraints in metamodel-based model transformations, we achieved the following. Constraints are defined separately from transformation rules, and they are propagated to the pattern rule nodes by constraint weaving methods. The same constraint does not appear repetitiously in many different places, consistent constraint modification, and simple constraint removal has become possible.

Aspect-oriented constraint management makes the transformation rules and the constraints reusable: transformation rules can be executed with a different set of constraints, and constraints can be propagated to different transformation rules. This construct makes the approach oblivious [11]. The result of the constraint weaving procedure is a weaving configuration, not a new transformation with the copies of the original transformation rules. The weaving configuration contains references to the original transformation, the transformation rules, and the propagated constraints. Furthermore, it describes the result of the weaving: the links between the rule nodes and the aspect-oriented constraints. The weaving mechanisms facilitate to require not only the individual rules but the whole transformation to validate, preserve, or guarantee certain properties during the execution of the transformation.

3. CONSTRAINT SEPARATION

This section (i) presents the separation of the refining and validation constraints that make the transformations more modular, and (ii) introduces the concepts of the separation on a real world transformation case study.

3.1 Refining and Validation Constraints

There are several issues that cannot be expressed by only structural conditions. In order to specify the transformation rules precisely, additional constraints must be specified beyond the structure of the transformation rules. These constraints are expected to ensure the correct instantiation of the transformation rule elements during the matching process (preconditions).

Topological and attribute transformation methods cannot perform and express the issues, which can be addressed by constraint validation. Constraints require the transformation rules to validate, preserve or guarantee certain model properties. Of course, transformations can be executed without conditions, but in that case they are not validated any more. This means that after the successful execution of the rules additional check of the resulted model is required to ensure that the result is valid. In order to handle the constraints in the most appropriate way, we differentiate refining and validation type constraints.

Definition. A *refining constraint* completes the conditions required by the structure of the LHS of a transformation rule.

Definition. A *validation constraint* expresses a semantically motivated constraint without which the transformation would work correctly, except for abortion.

Refining constraints and the LHS structure of the rule define the exact places in the input model where the transformation should be applied. All refining constraints propagated to a LHS of a transformation rule should hold for a valid match. With or without different refining constraints, a transformation rule can be applied on different parts of the input model, therefore, it can result different outputs. Thus, refining constraints are an important part of the precise transformation rule definition.

As it has been mentioned above the transformation can be executed in the same way without validation constraints, except for abortion. But if there are no validation constraints defined for the transformation, then the transformation loses its validated property. This means that after the execution of the transformation, even if it has finished successfully, we cannot be sure that certain conditions hold for the output model. If a validation type constraint does not hold during the execution of a transformation, then the transformation exits at the actual rule and the transformation finishes unsuccessfully. Obviously, in this case all the changes performed by the already executed rules are rolled back.

The input of the constraint weaver is a transformation rule (or a transformation that contains the transformation rules) and the constraints: refinement, validation and weaving constraints. The output is the constrained transformation rule. A weaving constraint is also an OCL constraint that specifies the weaving but it is not propagated or used during the transformation. Weaving constraints requires optional conditions (attribute values, relations) during the weaving process [9] [10].

Refinement, validation and weaving constraints are aspect-oriented constraints. They are defined and stored independently from transformation rules, and they are propagated with weaving methods to their appropriate place.

3.2 A Case Study

The case study is a variation of the "class model to relational database management system (RDBMS) model" transformation (also referred to as object-relational mapping).

The requirements stated against the transformation that it should guarantee are the following properties:

- Classes that are marked as non-abstract in the source model should be transformed into a single table of the same name in the target model. The resultant table should contain one added primary key column, one column for each attribute in the class, and one or more columns for associations.

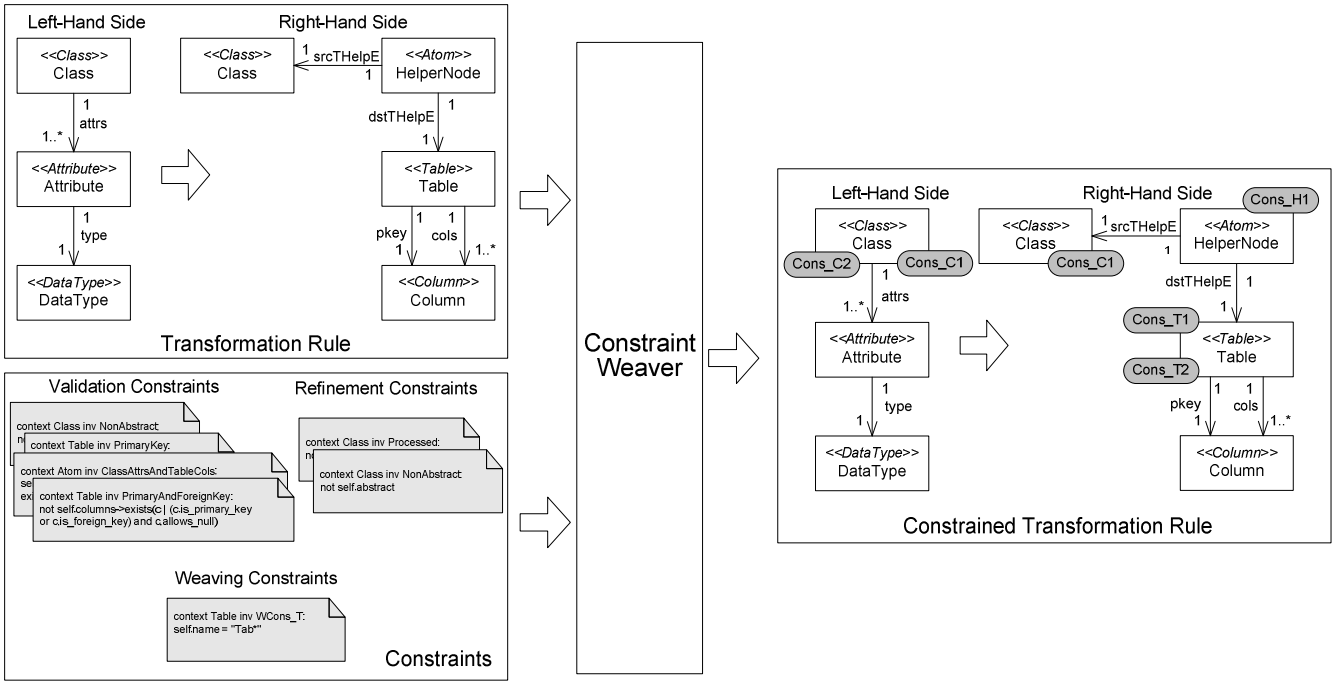

Figure 3. Constraint weaving in VMTS.

- In general, an association may, or may not, map to a table. It depends on the type and multiplicity of the association.
  - Many-to-many (N:N) associations, should be mapped to distinct tables. The primary keys for both related classes should become attributes of the association table (foreign keys). Foreign keys do not allow NULL values.
  - One-to-many (1:N) and associations using one or more foreign key columns should be merged into the table for the class on the "many" side.
  - For one-to-one (1:1) associations, the foreign key should also be buried optionally in one of the affected tables.
- Parent class attributes should be mapped into tables created from inherited classes.

In this paper only two transformation rules (*CreateTable* and *ProcessAssociation*) of the whole transformation are discussed in the point of the constraint separation view. For the complete solution of the case study refer to [4] and [7].

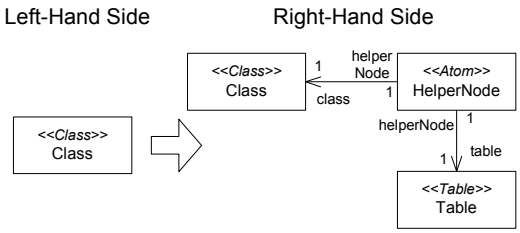

Figure 4. Transformation rule *CreateTable*.

The rule *CreateTable* is depicted in Fig. 4. It matches a non-abstract class and creates a table based on it. In order to require certain properties of the transformation rule *CreateTable*, the following constraints are applied:

```
context Class inv NonAbstract:
not self.abstract

context Class inv IsProcessed:
not self.is_processed
```

The constraint *NonAbstract* is assigned to the rule node *Class* in both LHS and RHS of the rule *CreateTable*. It requires the rule to process only non-abstract classes. This constraint is a *validation* type constraint. Without it the rule cannot be executed properly. (i) If we do not apply this condition, then all the classes are processed, that means tables are created even for abstract classes. (ii) Without the *refinement* type constraint *IsProcessed* the rule finds the same class again and again that results an infinity loop.

```
context Table inv PrimaryKey:
self.columns->exists(c | c.datatype = 'int' and
c.is_primary_key)
```

The constraint *PrimaryKey* is a postcondition of the rule *CreateTable*, it is assigned to the rule node *Table*. This *guarantee* type constraint requires the rule that all created table has a primary key of *int* type.

```
context Table inv PrimaryAndForeignKey:
not self.columns->exists(c | (c.is_primary_key or
c.is_foreign_key) and c.allows_null)
```

The constraint *PrimaryAndForeignKey* of *guarantee* type is also a postcondition that necessitates the primary and foreign key columns do not allow NULL values.

```
context Atom inv ClassAttrsAndTableCols:
self.class.attribute->forAll(self.table.column->
exists(c | (c.columnName = class.attribute.name))
```

The *guarantee* type constraint *ClassAttrsAndTableCols* is linked to the rule node *TableHelperNode*, it requires that each class attribute should have a created column with the same name in the resultant table.

Constraints *PrimaryKey*, *PrimaryAndForeignKey* and *ClassAttrsAndTableCols* are *validation* type constraints. This means that the transformation rule is validated by them. The rule can be executed without these constraints, and, obviously, it can generate the required result, but in that case the transformation rule is not validated any more. Consequently, if the validated property is important, validation type constraints should be applied.

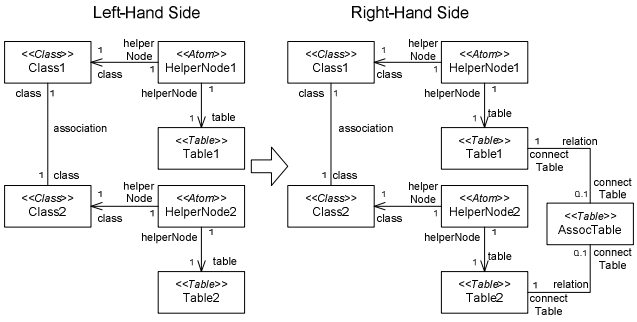

Figure 5. Transformation rule *ProcessAssociation*.

The rule *ProcessAssociation* (Fig. 5) processes the associations between classes. It creates association tables (N:N associations), and completes the already existing tables (1:N and 1:1 associations) with new foreign key columns. The following constraints are assigned to the rule *ProcessAssociation*:

```
context Association inv NotProcessedAssiciation:
not self.is_processed
```

The constraint *NotProcessedAssociation* is a precondition in LHS that requires matching only the not processed associations. This is a *refinement* type constraint. Without this constraint the transformation could not produce the required output: the transformation could not terminate, because the rule *NotProcessedAssociation* can match the same association in an infinite loop.

```
context Association inv OneToOneOrOneToMany:
  (self.leftMaxMultiplicity = '1' or
  self.rightMaxMultiplicity = '1') implies
  self.attribute->forAll
  (self.class.helperNode.table.column->
  exists(c | (c.columnName = attribute.name))
```

The constraint *OneToOneOrOneToMany* guarantees that the attributes of the one-to-one and the one-to-many association are buried into one of the tables created for the classes connected by the actually processed association.

```
context Association inv ManyToMany:
   (self.leftMaxMultiplicity = '*' and
   self.rightMaxMultiplicity = '*') implies
   self.attribute->forAll(self.class.
   helperNode.table. connectTable.column->exists(c
   | (c.columnName = attribute.name))
```

The constraint *ManyToMany* guarantees that there is a distinct table for each many-to-many type association in the resulted model. Furthermore, the table contains all attributes of the association with the same name.

Constraints *OneToOneOrOneToMany* and *ManyToMany* are *validation* type constraints.

As a result, we can separate the refinement type constraints that have functional role in transformations from validation type constraints (Fig. 3). Separating these constraints manually is a hard task that requires intuitive skills. This method facilitates to reduce the complexity of validation type constraints that makes them more understandable and working with them becomes easier.

4. RELATED WORK

This section elaborates on related work with respect to aspect-oriented constraint management, constraint management in model transformation tools, and provides a comparison of OCL-based constraints with negative and positive application conditions.

An aspect-oriented approach is introduced in [12] for software models containing constraints, where the dominant decomposition is based upon the functional hierarchy of a physical system. This approach provides a separate module for specifying constraints and their propagation. A new type of aspect is used to provide the weaver with the necessary information to perform the propagation: the strategy aspect. A strategy aspect provides a hook that the weaver may call in order to process the node-specific constraint propagations.

Constraint-Specification Aspect Weaver (C-SAW) [13] [14] is an aspect-oriented approach to modeling. The C-SAW weaver framework serves as a generalized transformation engine for manipulating models. C-SAW is a plug-in for the Generic Modeling Environment (GME) [15]. The result of model weaving is a new model that contains adaptations that are spread across the model hierarchy. These adaptations can be undone, and new concerns can be woven. The weaver can be used to distribute any system property endemic to a specific domain across the hierarchy of a model. A weaver can also be used to instrument structural changes within the model according to the dictates of some higher-level requirement that represents a crosscutting concern.

Both strategy aspect and C-SAW are related to the system models, but the constraint management and weaving methods of VMTS [9] [10] are related to model transformation rules. VMTS approach focuses on model transformation; therefore, its methods support the efficient and validated model transformation [7].

4.1 Constraint Management in Model Transformation Tools

Many approaches have been introduced in the field of graph grammars and transformations to capture graph domains; for instance, the GReAT [16], the PROGRES [17], the FUJABA [18], the VIATRA [19], AGG [20], and AToM^3 [21]. These approaches are specific to the particular system, and each of them has some features that others do not offer.

The GReAT framework is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts. The control structure of GReAT allows specifying an initial context for matching to reduce the complexity of the general matching case. In GReAT, LHS of the rules can contain OCL constraint to refine the structure but postconditions are not supported. PROGRES is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. In rules PROGRES uses attribute constraints, cardinality, and negative edges. In FUJABA, a combination of activity diagrams and collaboration diagrams (story-diagrams) are used to express control structures. Constraints in rewriting rules are also expressed by story-diagrams and Java code. VIATRA is a model transformation framework, its attribute transformation is performed by abstract state machine statements, and there is built-in support for attributes of basic Java types. The model and rule constraints can be expressed by graph patterns with arbitrary levels of negation. AGG is a visual tool environment consisting of editors, interpreter and debugger for attributed graph transformation; attribute computation by Java. The control structure of AGG is given by layers. In AToM^3, the rule constraints can contain generalized negative application conditions and can be pre- and postconditions to events. Constraints can be both semantic and graphical constraints. Similarly to AGG, the control flow consists of layers; the rules are sequenced by priority numbers within the layers.

Visual OCL (VOCL) [22] is a graphical representation of OCL that follows the UML notation and its graphical representation. VOCL is a formal, typed and object-oriented language. Data types such as collections and operations (e.g. *forall*, *select*, *union*) are represented by graphics. Logical expressions are denoted as *Peircian* graphs using different kinds of box to express disjunctions and conjunctions.

The full visual expression of the OCL constraints makes a complex constraint to big, it can reach the size of a whole page, and therefore, it cannot be used efficiently on a class or a different software model. Constraint aspects (Section 4.3) of VMTS approach [9] combine the advantages of the visual and the textual approaches: the structures of the constraints are expressed by the pattern of the constraint aspects, but the exact conditions with textual OCL constraints. Defining visual model transformations (both model-to-model and model-to-code transformations), we found that this is the most efficient and comfortable approach to constraint satisfaction. In addition, constraint aspects incorporate the possibility of the normalization [23] that makes them more efficient during the transformation execution.

4.2 Comparing OCL-Based Constraints with NACs and PACs

Application conditions for graph productions have firstly been introduced in [24]. In [25], a special kind of application conditions is considered which can be represented in a graphical way. Contextual conditions like the existence or non-existence of certain nodes and edges or certain subgraphs in the given graph

can be expressed. Furthermore, in [26], conditional application conditions were discussed.

Application conditions define the terms under which productions can be applied. Such application conditions include context conditions like the existence or non-existence of nodes, edges, or certain subgraphs in the given graph as well as embedding restrictions concerning the morphisms from the left-hand side of the production to the given graph.

We differentiate negative and positive application conditions (NACs and PACs). Finding subgraphs in graph, one frequently wants to express not only the presence but the absence of certain nodes or edges. Therefore one extends the matcher generator to handle negative application conditions (NACs). Similarly, PACs demand the existence of the defined structure [27].

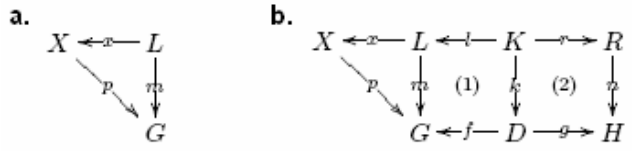

Figure 6. (a) Simple negative application condition, (b) Direct typed graph transformation with negative application condition.

Simple negative application condition is depicted in Fig. 6a. In Fig. 6b double pushout (DPO) [27] diagram is given with a graph production rule ($p:(L \xleftarrow{l} K \xrightarrow{r} R)$) and a NAC. The graphs *L*, *K*, and *R* are called the left-hand side, the interface, and the right-hand side graph of *p*, respectively.

Unfortunately, in the current DPO approach NACs and PACs are restricted only to simple structures. This approach is not able to deal with multiplicity, *ForAll* constructs and attribute value conditions yet [27].

An example graph production with application conditions and their metamodels are depicted in Fig. 7. The top row of the diagram presents the metamodel-based model transformation rule and the metamodels of the application conditions. The top row constructs are the metamodels of the middle row constructs. The dashed lines between the top and middle rows represent instantiation relations. In the top row *ML* and *MR* are LHS and RHS of the metamodel-based model transformation rule. This rule (*ShiftParentClassHelper*) is part of the transformation *Class2RDBMS*. One of the major challenges is to process the inheritance hierarchy properly, thus, the transformation must traverse the inheritance chains, because the parent class association should be taken into account recursively by subclasses. The rule *ShiftParentClassHelper* removes the original *ParentClassHelperNode* and adds a new one which links the subclass to the parent of the parent class, thus, the rule supports to process the whole inheritance hierarchy.

The bottom row in Fig. 7 introduces a concrete application of the transformation rule on an input model. The interpretation of the application conditions on the instance layer means the following: *p* can be applied to a graph *G* via a match *m* if there is no edge from *m(2)* to *m(1)* in *G* and there is at least one *Class* type neighbor node of node *m(1)*.

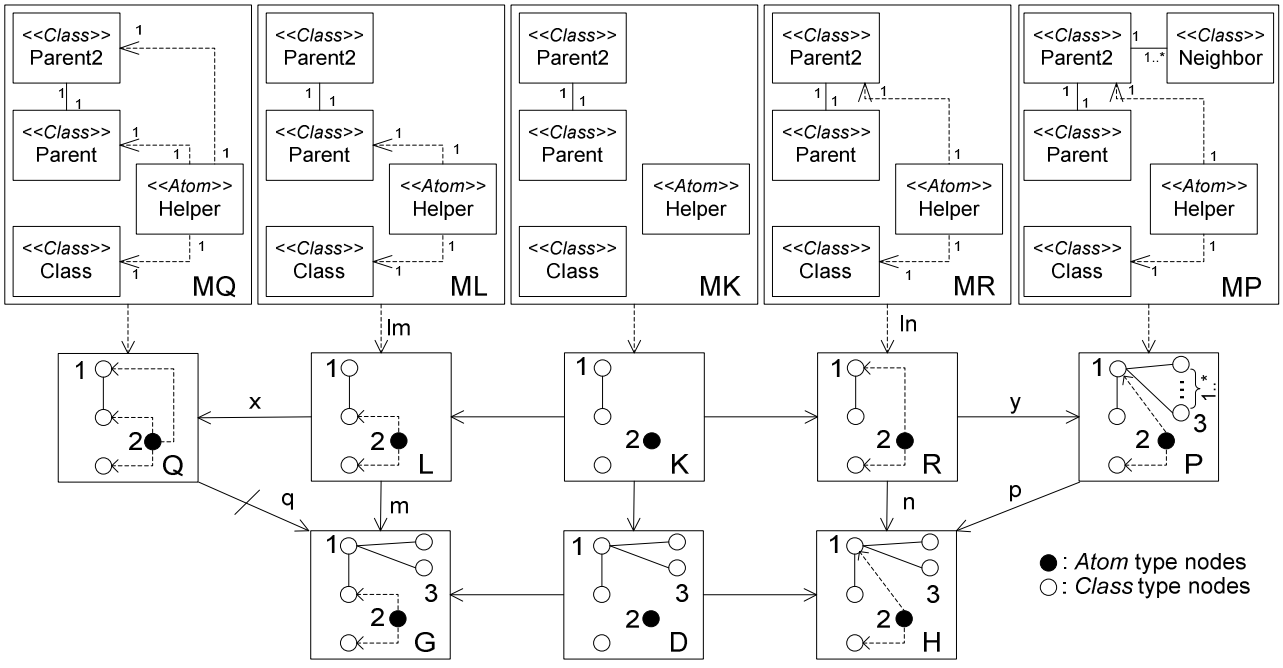

Figure 7. Example metamodel-based NAC and PAC.

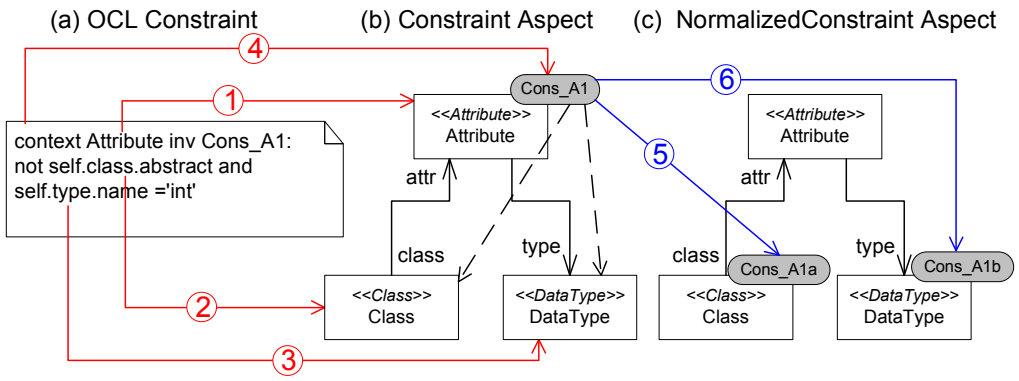

Figure 8. Creating constraint aspect from OCL constraint.

4.3 Creating NACs and PACs from OCL Constraints

There are various ways to express constraints:

- Cardinalities: the multiplicities that restrict the number of attached edges, thus the number of the neighbors e.g. in class or entity-relation diagrams.
- Logic: $\neg\exists t_1 : Table; t_2 : Table; t_1.name = t_2.name$.
- Graphical: forbidden or required subgraphs: NAC and PAC in Fig. 7.

As it has been discussed in the previous section, NAC and PAC constructs do not support multiplicity and attribute value conditions. But based on the case study (Section 3.2), we can state that in practice, there is a high demand for these constructs to express refining and validation type constraints.

In VMTS, metamodels, models and transformations are defined by graphical models. But OCL constraints used to specify the transformations rules are textual expressions. Therefore, the concept of constraint aspect has been developed. A *constraint aspect* expresses the same conditions as an OCL constraint in a visual way.

A *constraint aspect* is a pattern (structure) built from metamodel elements to which OCL constraints are assigned. A constraint aspect contains not only textual conditions described by the OCL constraints but structure, type and multiplicity conditions and weaving constraints as well [10].

The structure, type conditions and weaving constraints are checked at propagation time, while the OCL constraints are validated during the model transformation.

Because of their structure, constraint aspects are similar to NACs and PACs. The main difference is that constraint aspects are on the metamodel layer, therefore, they are more powerful: they support multiplicity and attribute value conditions.

Fig. 8 depicts an example how to create constraint aspects from OCL constraints.

The pseudo code of the algorithm CREATECONSTRAINTASPECT:

```
CREATECONSTRAINTASPECT (Constraint C): VMTSConstraintAspect
 1 CREATE CA by Context of C
 2 foreach Navigation Step N in C
 3   CREATE PRN Pattern Rule Node with type of the DestinationNode of the N
 4   LINK PRN to CA
 5 end foreach
 6 PROPAGATE C to the root node of the CA
 7 NORMALIZECONSTRAINTS (CA)
 8 return CA
```

In fact, it is more efficient to work with constraint aspects than OCL constraints, because during propagation of the constraint aspects the metatype-based searching can be used for pattern matching [3] to reduce the possible places of the constraint assignment. Furthermore, the result of the normalization [23] is that the constraints are in their optimal place. The normalization of the OCL constraints can be achieved only after their propagation when they are linked to the appropriate rule nodes. There are two related problems: (i) the result of the weaving process is a configuration, where constraints are assigned virtually to rule nodes that makes the execution of the normalization algorithm difficult. (ii) The normalization should be executed on all places to which the constraint is propagated. This means that the same operation should be executed several times. Contrarily, the normalization of a constraint aspect can be executed before the weaving, and it should be performed only once. Therefore, the normalization is performed only for the constraint aspects, thus, during the transformation process it is more efficient to work with constraints that contain propagated constraint aspects instead of linked OCL constraints. Consequently, constraint aspects accelerate both the weaving and the transformation processes.

5. CONCLUSIONS

The current paper has introduced the (i) separation of the *refining* and *validation* constraints and (ii) the constraint realization possibilities with negative and positive application conditions (NAC, PAC). We have illustrated these two aspects on a case study taken from a real-world transformation.

We have summarized the main principles of the constraint-driven validated online model transformation and the aspect-oriented constraint management. These constructs facilitate to validate not only the individual transformation rules, but the whole transformations. Furthermore, the separation of the *refining* and *validation* constraints facilitates that the transformation becomes more modular and its elements, such as transformation rules and constraints, have become reusable.

We have presented that the VMTS approach supports multiplicity, logical and graphical-based constraints. An algorithm is given to create constraint aspects from OCL constraints. The generated constraints aspects have structure that represents graphical constraints and can be used as NACs or PACs. Using simple NACs and PACs, it is possible to express structures, but multiplicity and attribute value conditions cannot be defined without additional constraints. It has been also shown that constraint aspects combine the advantages both the structure and logic based constraints.

6. ACKNOWLEDGMENTS

The activities described in this paper supported, in part, by Information Technology Innovation and Knowledge Centre.

7. REFERENCES

[1] OMG MDA Guide Version 1.0.1, OMG, doc. number: omg/2003-06-01, 12th June 2003, http://www.omg.org/docs/omg/03-06-01.pdf

[2] OMG Object Constraint Language Spec. (OCL), www.omg.org

[3] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, International Workshop on Graph-Based Tools (GraBaTs) ENTCS, 2004.

[4] The VMTS Homepage. http://www.vmts.aut.bme.hu

[5] G. Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol.1 World Scientific, Singapore, 1997.

[6] H. Ehrig, G. Engels, H-J. Kreowski, G. Rozemberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools, Vol.2. World Scientific, Singapore, 1999.

[7] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf, Model-Based Development with Strictly Controlled Model Transformation, The 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006, May 23-24, 2006, Paphos, Cyprus, pp. 39-48.

[8] L. Lengyel, T. Levendovszky, G. Mezei, B. Forstner, H. Charaf, Metamodel-Based Model Transformation with Aspect-Oriented Constraints, International Workshop on Graph and Model Transformation, GraMoT, ENTCS Vol. 152, Tallinn, Estonia, 2005, pp. 111-123.

[9] L. Lengyel, T. Levendovszky, H. Charaf, Eliminating Crosscutting Constraints from Visual Model Transformation Rules, ACM/IEEE 7th International Workshop on Aspect-Oriented Modeling, October 2, 2005, Montego Bay, Jamaica.

[10] L. Lengyel, T. Levendovszky, H. Charaf, Optimizing Constraint Weaving in Model Transformation with Structural Constraint Specification, 8th Int. Workshop on Aspect-Oriented Modeling, March 21, 2006, Bonn, Germany.

[11] R. E. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2004.

[12] J. Gray, T. Bapty, S. Neema, and J. Tuck, Handling Crosscutting Constraints in Domain-Specific Modeling, Communications of the ACM, October 2001, pp. 87-93.

[13] K. Balasubramanian, A. Gokhale, Y. Lin, J. Zhang, J. Gray. Weaving Deployment Aspects into Domain-Specific Models. International Journal on Software Engineering and Knowledge Engineering, 2006.

[14] J. Gray, Y. Lin, J. Zhang, S. Nordstrom, A. Gokhale, S. Neema, S. Gokhale. Transformations to Address Model Scalability. In Model Driven Engineering Languages and Systems (MoDELS), number 3713 in LNCS, pp. 295–308. Springer-Verlag, 2005.

[15] A. Lédeczi, M. Maróti, P. Völgyesi, Framework for the Rapid Development of Modeling Environments, Chapter in Advanced Topics in Database Research, Idea Group Publishing, 2004.

[16] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Computer Science, 2003.

[17] J. Reekers, A. Schürr, Defining and Parsing Visual Languages, Journal of Visual Languages and Computing, 8, Academic Press, 1997, pp. 27-55.

[18] H. J. Köhler, U. A. Nickel, J. Niere, A. Zündorf, Integrating UML Diagrams for Production Control Systems, ICSE, Limerick Ireland, ACM Press, 2000, pp. 241-251.

[19] D. Varró and A. Pataricza, "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML", SoSyM, 2003.

[20] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, In J. Pfaltz, M. Nagl, and B. Boehlen (eds.), Application of Graph Transformations with Industrial Relevance (AGTIVE'03), vol. 3062. Springer LNCS, 2004.

[21] J. Lara, H. Vangheluwe, M. Alfonseca, Meta-modelling and graph grammars for multi-paradigm modelling in AToM, Software and Systems Modeling (SoSyM), 3(3):194-209, August 2004.

[22] P. Bottoni, M. Koch, F. Parisi-Presicce, G. Taentzer, Consistency Checking and Visualization of OCL Constraints, UML 2000, pp. 294-308.

[23] L. Lengyel, T. Levendovszky, H. Charaf, Normalizing OCL Constraints in UML Class Diagram-Based Metamodels - AND/OR Clauses, EUROCON 2005 Int.Conference on "Computer as a tool", Proceedings of the IEEE, Belgrade, Nov. 21-24, 2005, pp. 579-582.

[24] H. Ehrig and A. Habel. Graph Grammars with Application Conditions. In G. Rozenberg and A. Salomaa, editors, The book of L, pages 87–100. Springer, 1986.

[25] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. Fundamenta Informaticae, 26(3/4):287–313, 1996.

[26] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Rewriting - a Constructive Approach. In Proceedings of SEGRAGRA 1995, volume 2 of ENTCS, 1995.

[27] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamentals of algebraic graph transformation. In EATCS Monographs in Theoretical Computer Science. Springer, 2006.

3rd European Workshop on Aspects in Software (EWAS 2006)

On Dynamically-Scoped Crosscutting Mechanisms

Éric Tanter

DCC / CWR – University of Chile
Santiago – Chile
www.dcc.uchile.cl/∼etanter

Abstract

Many crosscutting mechanisms proposed in the literature offer means to restrain aspects to some dynamically-defined scopes. Dynamically-scoped mechanisms are particularly interesting because of the flexibility, expressiveness, and control they give over structural and/or behavioral changes made by aspects. Since the nature of dynamic scopes as well as the scoping mechanisms themselves greatly vary among proposals, it is relatively complex to compare them. This paper aims at filling this gap by proposing a characterization of dynamically-scoped crosscutting mechanisms, hence providing a reference frame for comparing different approaches. As a result, this work clarifies some differences between related approaches, suggests possible tracks for further exploration of the design space of such mechanisms, and discusses some issues raised by dynamically-scoped aspects.

1. Introduction

Aspect-Oriented Programming [9] and other related approaches propose mechanisms for the modularization of crosscutting concerns. Although all these mechanisms were initially statically-scoped, there is a growing trend of interest in dynamic scoping mechanisms in the field. This interest in dynamically-scoped crosscutting mechanisms is justified by the flexibility and expressiveness offered by such mechanisms, that make it possible to conveniently express structural and/or behavioral changes that should only occur under certain dynamic circumstances.

Examples include conditionals like `if` pointcuts in AspectJ and others, control and data flow conditions, tracematches and stateful aspects, and context-aware aspects [12, 10, 14, 8, 1, 7, 20, 2, 17]. Similarly, related proposals which are not AOP as such, but still promote better modularization of crosscutting concerns, such as metaobject protocols and structural refinements, have introduced dynamic scoping mechanisms [19, 4, 5]. It is important to note that by dynamic scope, we do not only refer to thread-local values or code, but also to any technique that allows for subjecting aspects to the past, current and surrounding states of the program.

As a side effect of this proliferation of dynamic scoping mechanisms, it is relatively complex to compare each proposal with respect to how dynamic scoping is actually provided. In this paper we propose a characterization of dynamically-scoped crosscutting mechanisms. The objective of this classification is to give a reference frame for comparing different approaches. Therefore, beyond the classification itself, this paper contributes to the field by *(a)* clarifying the differences between related approaches, *(b)* highlighting similar combination of characteristics of some approaches, thus calling for deeper comparison, and *(c)* pointing out some tracks for further exploring the design space of dynamically-scoped crosscutting mechanisms.

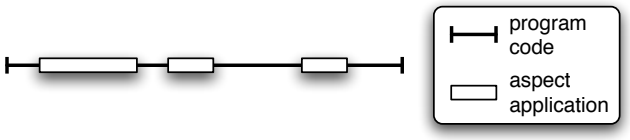

Figure 1. Unidimensional view of the application of an aspect.

In the following, we use aspect in a broad sense, which includes mechanisms for handling crosscutting such as Classboxes [4] and mixin layers as provided in ContextL [5].

2. Characterizing Dynamic Scope

2.1 Dynamically-Scoped Aspects in a Nutshell

An aspect *applies* to a program by modifying either its structure and/or its behavior. We refer to these as structural and behavioral aspects, respectively. We say that a structural aspect applies if the changes it makes to the structure are effective[1].

The application of an aspect is always statically scoped: some criteria define the parts of the program that are altered by the aspect. Fig. 1 represents a unidimensional view of the application of an aspect: the entire program code is represented as a line, affected by the application of an aspect. If we consider time, that is, program execution, then the application of a statically-scoped aspect is depicted in Fig. 2: the parts of the program that are affected by the aspect are *always* affected, from the start to the end of the execution.

A dynamically-scoped aspect is different. Depending on the program being in a certain scope or not during execution, then the aspect applies or not. This is depicted in Fig. 3, where we see that the aspect only applies within a certain dynamic scope, which is active four times in the picture. Note that we indifferently refer to *"the program is in a given scope"* and *"the given scope is active"*. The above relates to an aspect that is subject *as a whole* to a certain scope. Most aspect languages and tools allow for the definition of aspects composed of different subparts, each potentially subject to different dynamic scopes.

[1] In many implementations, structural changes are made once for all, and an additional logic ensures that the changes are visible only in the desired dynamic scope.

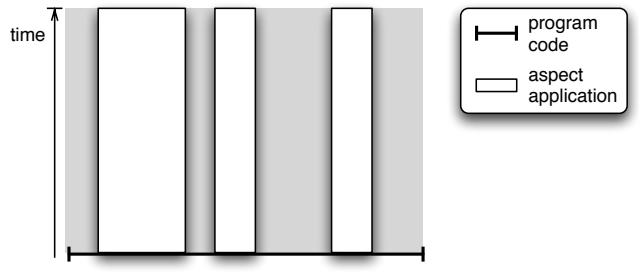

Figure 2. Bidimensional view of the application of a statically-scoped aspect.

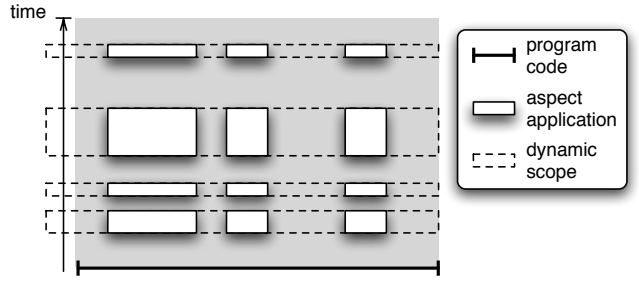

Figure 3. Bidimensional view of the application of a dynamically-scoped aspect.

2.2 Dimensions of Characterization

We propose several dimensions of characterization for dynamically-scoped crosscutting mechanisms. These dimensions have been chosen by focusing on semantics and software engineering issues. We do not consider the mechanisms from a point of view of implementation techniques and possible runtime efficiency. The reason for this is the need we see for a clear *conceptual* framework relating to dynamically-scoped aspects. We identify the following dimensions:

aspect action. What is the nature of the aspect action? It can be either structural (*e.g.* a refinement as in Classboxes [4] and mixin layers [5] or an inter-type declaration as in AspectJ) or behavioral (*a.k.a.* advice).

scope definition. How are the boundaries (*i.e.* the entry and exit points) of the scope specified by the programmer? Is it done implicitly or explicitly at the points where the scope starts and ends? This characteristic determines the *intrusiveness* of the scope definition (*i.e.* how much base code has to be modified in order to define the scope), as well as its tractability by the user (*i.e.* how easy it is for the programmer to infer when a given scope is active or not). It has to be noted that in case of intrusive scope definitions, aspects can normally be used to insert start and end points transparently.
Another question, which is highly dependent on the implementation platform (and hence not considered further) is how the scope boundaries are determined by the runtime infrastructure[2].

[2] This refers to either "push"-like mode whereby the aspect is effectively installed upon entering the scope, or "pull"-like mode in which the aspect is always installed and a systematic check is performed before evaluating the aspect action. A system which truly supports dynamic weaving can choose the "push" option, while other systems have to resort to "pull" techniques.

scope information exposure. Can information associated with the scope (*e.g.* a value of a variable bound in the dynamic scope) be exposed to the aspect action? This characteristic is particulary important to foster reuse of aspect actions, through genericity. Without such a feature, workarounds have to be devised in order to share scope-specific information with the aspect action, if at all possible.

scope-aspect binding. When is the binding between a scope and an aspect made? We use the characterization of binding *time* and *mode* introduced in [16]. The *binding time* is the moment at which the binding is established; it can be either compile time[3] or runtime. The *binding mode* is another dimension referring to whether the binding can be undone/redone during execution. If yes, then the binding mode is said to be dynamic, otherwise it is said to be static.

thread locality. Considering a multi-threaded execution, is the scope defined locally for each thread? If not, should thread locality be manually implemented? This characteristic is very important for the actual semantics of the dynamic scoping mechanism, and the programmer should therefore be well aware of the default semantics of the mechanism (*e.g.* a cflow in AspectJ is by default thread local, while a tracematch in the AspectJ extension of [1] is by default global). Note that thread locality is explicitly considered in this classification because dynamically-scoped mechanisms are not necessarily related to execution stack properties; they can for instance relate to past execution states, or to elements pertaining to the physical environment of the running system.

3. Dynamic Scoping Mechanisms

We now analyze a number of proposals for dynamically-scoped aspects, in the light of the different criteria mentioned above. We consider the following proposals: AspectJ [12] and similar systems, AspectJ with tracematching [1], Reflex [19, 18], AspectS [10], context-aware aspects [17], CaesarJ [2], Classbox/J [4, 3], and ContextL [5]. The two last approaches are the only dynamically-scoped approaches to structural changes we know of (intertype declarations in AspectJ and structural links in Reflex are both statically scoped). We also only focus on language-level approaches, not architecture-level ones. Our analyis is summarized on Fig. 4.

If/restriction. This is the basic means to specify an arbitrary boolean condition to be evaluated at runtime, as with the `if` pointcut in AspectJ or restrictions in Reflex to name a few. This kind of conditions is statically bound at compile time. The dynamic scope is defined by the condition evaluating to true. Such conditions typically cannot expose context information to the aspect action. Thread locality has to be manually handled.

Dynamic join point properties. We hereby refer to dynamic conditions over join point properties, such as receiver or target runtime type, actual arguments, etc. This mechanism is supported by most AOP systems, and usually permits to expose dynamic information to the aspect action. This contrasts with the previous mechanism.

First-class activation. Some AOP systems like AspectS and Reflex support first-class activation conditions. These are objects encapsulating a dynamic condition. The difference with the if/restriction mechanism is binding mode: activation conditions can be unbound/rebound at runtime. So both Reflex and AspectS offer the same binding mode (dynamic), but differ in the binding time due to implementation environments (Java vs. Smalltalk); Reflex offers dynamic binding at compile time, while AspectS has dynamic binding at runtime. First-class activation is typically global

[3] For simplicity, we refer to compile, deploy and link times as compile time.

scoping mechanism	*example systems*	*scope*	*action*	*definition*	*exposure*	*binding time/mode*	*thread local*
if/restriction	*most*	condition true	B	I	no	CT/S	M
receiver, target, etc.	*most*	JP dynamic property	B	I	yes	CT/S	M
1st-class activation	Reflex, AspectS	condition true	B	I	no	*depends*/D	M/A
cflow	*most*	dyn-ext. of nested pointcut	B	I	yes	*depends*	A
tracematches	AJ+TM	execution history	B	I	yes	CT/S	M/A
context-aware aspects	Reflex extension	any context	B	I	yes	CT(*)/D	M
deploy block	CaesarJ, Steamloom	dyn-ext. of `deploy`	B	EI	no	RT/S	A
on/off	AspectS, Reflex	between `on` and `off`	B	EE	no	RT/D	M
classboxes	Classbox/J	dyn-ext. of oldest client package	S	I (**)	no	CT/S	A
mixin layers	ContextL	dyn-ext. of `with-active-layer`	S	EI	no	RT/D	A

B: behavioral – S: structural – I: implicit – EI: explicit start, implicit end – EE: explicit start and end
CT: compile-time – RT: run time – S: static – D: dynamic – M: manual – A: automatic

(*) The context-aware aspects of [17] actually support *dynamic* binding without exposure, in addition to static binding with exposure.
(**) Classboxes imply an implicitly-defined scope, but still rely on *explicit* import declarations between modules.

Figure 4. Summary of the characterization of different dynamic scoping mechanisms.

to all threads, but thread locality can be developed in a reusable and generic manner [11] (hence the `M/A` characterization on Fig. 4).

Cflow. Most AOP languages and systems support the definition of scope based on control flow, like the `cflow` and `cflowbelow` pointcut designators in AspectJ. These are higher-order constructs, and the scope they define is the dynamic extent of the nested pointcut. Although bound statically in AspectJ, in Reflex one can actually choose between a statically-bound restriction or a dynamically-bound activation condition to define a scope based on control flow. An AspectS implementation could actually be dynamically-bound at runtime. A major use of this mechanism is to make information up in the stack available to an aspect action (like in the wormhole design pattern [13]). All control flow implementations are indeed thread local, since control flow is hereby a stack-related property (and each thread has its own stack).

Tracematches. Stateful aspects or tracematches make it possible to restrain the application of an aspect to the occurrences of certain execution event patterns. Compared to the above mechanism, tracematches greatly enhance the expressiveness of the scoping, although not differing significantly in most other regards. An exception that we already mentioned is the thread locality of the tracematch: the default assumption of the AspectJ extension proposed in [1] is that the tracematch is checked globally, that is, matching events in all threads of the system. Still, a special `threadlocal` keyword is supported, to easily switch to a thread-local semantics, whereby sequences of events are looked for in the execution history of a single thread. We consider the syntactical equivalent of a reusable activation condition for thread locality. The most mature implementations to date [20, 1] are statically-bound at compile time.

Context-aware aspects. A Reflex framework for context-aware aspects was proposed in [17]. The idea is to extend pointcut languages with context-specific restrictions, allowing both parameterization of context definitions and exposure of context state to the aspect action. Context here refers to more than just the program execution context, as it includes external context perceived by sensors, for instance. The clear separation of aspects and contexts fosters evolution and reuse of both. The dynamic scope definition in this approach is actually the context definition. In addition to being active or not, a context exposes state associated to its being active. The notion of context is very general and subsumes all previous approaches. It is similar to the if/restriction mechanism, but allows scope information exposure. Context-aware aspects in Reflex can be implemented using either restrictions or activation conditions, offering different binding modes. Although by default thread global, if first-class activation is used as an implementation, thread locality can be obtained in context-aware aspects by reusing a thread-local semantics (as discussed previously).

Deploy block. CaesarJ supports a dynamic scoping mechanism that consists in restricting an aspect to the dynamic extent of a `deploy` block. For instance, `deploy(A){...}` implies that the aspect `A` is applied to all execution occurring within the dynamic extent of the block. In contrast with all the other scoping mechanisms we have seen so far, here the scope is *explicitly* embedded within the application code. Still, the definition has an *implicit* exit point (the end of the deploy block). Deploy blocks cannot be parameterized with values for access later down in the execution. The scope-aspect binding is done at runtime, but is indeed static: within the dynamic extent of the deploy block, the deployment cannot be undone and redone. Finally, the aspect is deployed locally to the thread executing the deploy block.

Explicit on/off. Some proposals support (de)activation of an aspect by explicit calls to `on/off`-like methods. In AspectS this can be done by executing `A install` or `A uninstall` at any time (dynamic binding at runtime). Less flexible versions of this feature can be implemented using first-class activation (as in Reflex) or even if/restriction mechanisms. In this approach, the dynamic scope to which an aspect is bound consists in the execution taking place in between a call to the `on` method and a call to the `off` method. Therefore both entry and exit points of the scope definition are explicitly defined. Like other low-level conditional mechanisms, thread locality has to be manually implemented.

Classboxes. Classboxes are a mechanism for dynamically-scoped structural *refinements*. Similar to open classes, refinements in classboxes make it possible to extend a class definition "from the outside" with new fields and methods, as well as extending methods with a mechanism similar to overriding in standard object-oriented programming. However, while changes made with open classes are globally visible, classboxes introduce a dynamic scoping mechanism based on import relations between modules: a refinement to a class is only visible for all execution that originates from a client of the module in which the refinement is defined. This property is checked at runtime: it can be extracted from the stack, and is therefore thread local. The definition of the scope of a refinement with classboxes is implicitly defined in the sense that the programmer does not explicitly says when the entry or exit points of a scope

occur, but still requires explicit import declarations which are used by the runtime system to determine these points automatically. The binding is done at compile time, and is static. No context information can be exposed to the refinements.

Mixin layers. ContextL is a CLOS extension that introduces a mechanism of dynamic mixin layers, which are dynamically scoped. In ContextL, there is a notion of layers in which one can define structural refinements. These structural refinements can then be dynamically activated, by executing a set of expressions in the dynamic extent of a `with-active-layer` call. In other words, this mechanism can be seen as a combination of the deploy block mechanism of CaesarJ, but for structural actions like in classboxes. Therefore the definition of the scope is similar to the deploy block (explicit entry point, implicit exit point). However, as opposed to the deploy block a Caesar, a mixin layer can be temporarily deactivated, using a `with-unactive-layer`. Hence this mechanism supports dynamic binding at runtime. Thread locality is automatic, and, like the last three mechanisms presented, mixin layers cannot expose information to the refinements.

4. Conclusion and Discussion

We have described a frame of reference to compare different dynamically-scoped crosscutting mechanisms, including both structural and behavioral approaches. As a result of this study, differences between related mechanisms are clarified. For instance, the precise difference between an `if` pointcut and a first-class activation condition lies in the binding mode and possibly in the binding time (*e.g.* in the case of AspectS). Also, the contribution of context-aware aspects versus these conditional mechanisms comes to light: the possibility to expose information bound in the conditional to the action. Tracematching appears similar and therefore a deeper comparison of both mechanisms should be valuable. An interesting parallel can be made between the deploy block mechanism as in CaesarJ and the mixin layers of ContextL: apart from the fact that one is used for behavioral aspects and the other for structural changes, they actually differ with respect to the binding mode (the binding time is the same). More comparative results appear directly from looking at Fig. 4.

Among the suggestions for future development in the area, it seems noteworthy that all mechanisms that are explicitly defined (deploy block, on/off, classboxes, mixin layers) actually lack the context information exposure feature. As a matter of fact, such a feature does make sense for these mechanisms. This remark also holds for first-class activation.

Finally, it seems that a discussion about the different definition mechanisms is valuable. Most mechanisms embed the scope definition outside of the base code (implicit). The explicit ones require intrusive specification (*i.e.* base code embeds the scope boundaries definition) but prove useful in some cases, thanks to the dynamicity of the binding that is thereby obtained. As mentioned before, it is usually feasible to turn an explicit definition into an implicit one (by using an aspect!).

With respect to tractability, it is clear that dynamic scoping requires more effort from the programmer. In the case of explicit scope definitions, both deploy blocks and mixin layers as in ContextL have a clear scope, in the sense that the end of the scope is implicitly associated to the end of the lexical structure that specifies the begin of the scope (*i.e.* the closing } of a deploy block). Conversely, the on/off mechanism is much less "structured", in the sense of structured programming vs. GOTO-like programming [6]. If many explicit on/off statements are used in various places of an application, it can easily turn out to be impossible for a programmer to precisely foresee when an aspect will actually apply. If we make the analogy with the history of GOTO-like programming, then we may foresee that these kinds of (too?) flexible schemes may be bound to disappear in the light of a sound compromise.

In any case, dynamically-scoped aspects definitely challenge traditional visions of program understanding, much more than AOP in its implicit and lexically-scoped flavor –which is already a challenge compared to more traditional paradigms–.

Acknowledgments. The author thanks the following colleagues for the interesting discussions about one or more of the presented mechanisms: Alexandre Bergel, Eric Bodden, Pascal Costanza, Marcus Denker, Brecht Desmet, Kris Gybels, Robert Hirschfeld, and Theo D'Hondt.
É. Tanter is partially financed by the Milenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In OOPSLA 2005 [15], pages 345–364. ACM SIGPLAN Notices, 40(11).

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[3] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In OOPSLA 2005 [15], pages 177–189. ACM SIGPLAN Notices, 40(11).

[4] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer-Verlag, 2003.

[5] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA, October 2005.

[6] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[7] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150, Lancaster, UK, March 2004. ACM Press.

[8] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[9] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[10] Robert Hirschfeld. AspectS – aspect-oriented programming with Squeak. In Mehmet Akşit, Mira Mezini, and R. Unland, editors, *International Conference NetObjectDays on Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, 2002.

[11] Robert Hirschfeld and Pascal Costanza. Extending advice activation in AspectS. In *2nd European Interactive Workshop on Aspects in Software (EIWAS 2005)*, Brussels, Belgium, September 2005.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[13] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Press, 2003.

[14] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121, November 2003.

[15] *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA, October 2005. ACM Press. ACM SIGPLAN Notices, 40(11).

[16] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behavior. In Boris Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in Lecture Notes in Computer Science, pages 205–230, Málaga, Spain, June 2002. Springer-Verlag.

[17] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria, March 2006. Springer-Verlag.

[18] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert Glück and Mike Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, September/October 2005. Springer-Verlag.

[19] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[20] Wim Vanderperren, Davy Suvee, María Augustina Cíbran, and Bruno De Fraine. Stateful aspects in JAsCo. In *Proceedings of Software Composition (SC 2005)*, volume 3628 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2005.

Introducing mixin layers to support the development of context-aware systems

Brecht Desmet, Jorge Vallejos, and Pascal Costanza
Programming Technology Lab - Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels, Belgium
{bdesmet,jvallejo,pascal.costanza}@vub.ac.be

ABSTRACT

The domain of pervasive computation introduces new opportunities to make software systems aware of the context in which they exist in order to respond more adequately to user expectations. As important as it is to have appropriate ways to obtain context information, it is to provide programmers with clean language features to model the context influence inside of software systems. This paper explores the mixin layer language construct to implement context-dependent adaptations separate from the application core logic. Together with a mechanism that dynamically composes and activates mixin layers, we argue that mixin layers possess a great potential to support the development of context-aware systems.

1. INTRODUCTION

Context-aware computing envisions scenarios in which system behaviour is parameterized by context information such as location, activity or battery level. As these context parameters change over time, context-aware systems should adapt their behaviour accordingly. Although we currently find an important amount of research on context-awareness, most of the approaches focus on the way software systems can perceive their surrounding context. We believe that this is just half-way of the development of a context-aware system. A new and totally different problem emerges when programmers start using the context information to dynamically adapt the behaviour of software systems. In this paper, we claim that mainstream language constructs to implement the contextual influence on program behaviour easily results in unmanageable designs and code scattering.

We introduce a model in which context-aware systems are described by means of adaptations. Depending on the context in which a system appears, these adaptations can be dynamically applied to the application core logic of a system. The main issue of this paper is a motivation why mixin layers are a good candidate to implement these context-dependent adaptations. We will clarify our approach with a concrete implementation of a context-aware system.

This paper is organised as follows. Section 2 presents an example of a context-aware scenario. The problems with traditional language constructs to implement context-aware scenarios are discussed in Section 3. Next, Section 4 explains how mixin layers are related to context-aware systems. Sections 5 and 6 present a concrete implementation of the example from Section 2. Additionally, Section 8 pinpoints where our approach suffers from a computational overhead and proposes a solution for that. Finally, the main ideas of this paper are summarised in Section 10.

2. MOTIVATING EXAMPLE

We present the software of a simplified cellular phone as an illustration of a context-aware system. The phone example consists of the following functionalities. First, the phone contains a list of contacts, some of them marked as VIPs. This information is encapsulated in the `contacts` class. Second, the `messages` class provides facilities to read and send messages. Third, the `journal` class keeps track of all phone traffic. Finally, the main task of the phone is to ring whenever somebody calls and to provide the means to answer calls. This functionality is offered by the `phonecalls` class. These different functionalities constitute the *application core logic* of the cellular phone.

The behaviour of the application core logic can be adapted at runtime according to context changes. We introduce three *context-dependent adaptations*, each of which contain two parts: a *context condition* that explains when the adaptation is applicable and the *actual behaviour* of the adaptation with regards to the application core logic.

IgnoreAdaptation If the battery level is low, ignore and log all phone calls except for contacts that are classified as VIPs.

AnswerMachineAdaptation If the time is between 11pm and 8am, activate the answering machine for incoming phone calls and the auto-reply service for messages.

RedirectAdaptation If the user is in the meeting room, redirect all calls and messages to the secretary.

Although the three context conditions (battery low, time between 11pm-8am and meeting room location) can all be true

at the same time, the behaviour of the adaptations cannot be freely combined. This is because adaptations might contradict each other, like e.g. *IgnoreAdaptation* and *RedirectAdaptation*. In case of a contradiction, the user can make an arbitrary decision about what should happen. For instance, in our phone example, the following set of policy rules describes the valid combinations of adaptations and how contradictions should be resolved.

PolicyRule I All adaptations can exist individually.

PolicyRule II *IgnoreAdaptation* and *AnswerMachineAdaptation* can coexist. Only VIP contacts will get in touch with the answering machine, all other contacts will be ignored. All messages receive auto-reply.

PolicyRule III *IgnoreAdaptation* and *RedirectAdaptation* cannot coexist. *RedirectAdaptation* has priority.

PolicyRule IV *AnswerMachineAdaptation* and *RedirectAdaptation* cannot coexist. *AnswerMachineAdaptation* has priority.

3. PROBLEM STATEMENT

Traditional language constructs to implement scenarios like the cellular phone example include both design patterns (e.g. strategy or decorator pattern) and conditional tests (e.g. `if` or `case`). We discuss both approaches and explain how their usage might lead to cumbersome designs.

The strategy design pattern could be a good candidate to implement the cellular phone example. In this case, we regard each possible adaptation as another strategy that deals with incoming communication. This approach does not scale since context-aware systems typically involve multiple context parameters like *BatteryLevel* = {*low, high*} or *Location* = {*meetingroom, elsewhere*}. In the worst case, a combinatorial explosion of possible behavioural variants might arise. The strategy design pattern constructs a new object for each possible behavioural variant. This approach becomes unmanageable if the number of context parameters increases.

Alternatively, one could implement the adaptations using the decorator design pattern (aka wrapper). Here, we regard the adaptations as decorators that refine the behaviour of individual objects through delegation. Since context-dependent adaptations might affect multiple classes, the developer is burdened with the implementation of non-functional behaviour that manages multiple decorators simultaneously. For example, the activation or deactivation of multiple decorators that constitute a context-dependent adaptation should be an atomic operation to ensure consistent program behaviour. Hence, the decorator design pattern is in such cases inevitably complemented with additional "bookkeeping code" which troubles the system design. This is because the decorator design pattern is actually not expressive enough to adequately implement context-dependent adaptations.

The use of conditional statements also provide no solace since the implementation of context-dependent adaptations easily leads to code scattering. Furthermore, the context information of a system, which is actually application-specific domain knowledge, is also scattered around within the test-clauses of the conditional statements. This makes it extremely difficult to extend context-aware systems. Consider for example the case in which an additional context-dependent adaptation is introduced in a context-aware system. Such an extension requires full understanding of all context information used throughout the system and might affect multiple modularisation units. Therefore, this approach is not feasible for large-scale context-aware systems.

In summary, we agree that design patterns and conditional tests are a plausible solution to implement context-aware systems. However, these traditional constructs lack expressiveness and hence induce cumbersome designs. We therefore believe that more dedicated language constructs can alleviate the design of context-aware systems substantially.

4. CONTEXT-DEPENDENT ADAPTATIONS AS MIXIN LAYERS

The claim of this paper is that the modularisation capabilities of mixin layers offer important contributions to the development of context-aware systems. The basic idea is to separate the context-dependent adaptations from the application core logic and modularise them using mixin layers. However, current practices with mixin layers are not aligned with our specific dynamic requirements in the domain of context-aware computing. Before we explain these dynamic requirements in Section 4.2, we first explain briefly what mixin layers are about in the next section.

4.1 Mixin layers

The notion of mixin layers [8] was introduced by Smaragdakis et al. as an implementation technique to support refinement of collaboration-based designs. A mixin layer is a modularisation unit that encapsulates different mixin classes each refining a single class of the collaboration. Such mixin classes (or just mixins) are also commonly known as abstract subclasses. The distinguishing feature between ordinary and abstract subclasses is that the latter have parameterized superclasses. This property enables the instantiation of mixins with various superclasses and thus supports reusability.

In practice, refinement by using mixin layers is achieved through the ability to add or specialize methods and classes. Moreover, mixin layers can also refine other mixin layers because they can be composed in an inheritance hierarchy, yielding a layered design. Furthermore, since mixin layers can affect multiple classes, they support cross-cutting modularisation to a certain degree. These properties are illustrated in Figure 1.

4.2 Dynamic requirements

Mixin layers hold some promise to implement the context-dependent adaptations because they adapt behaviour through inheritance, support cross-cutting modularisation and can be composed in an inheritance hierarchy. Unfortunately, current practices with mixin layers are not aligned with the specific characteristics of context-aware systems. This section focusses on two important issues. First, since the context information of a system is not known beforehand (e.g. location of user), we require a mechanism that *selects and composes mixin layers at runtime* based on the actual con-

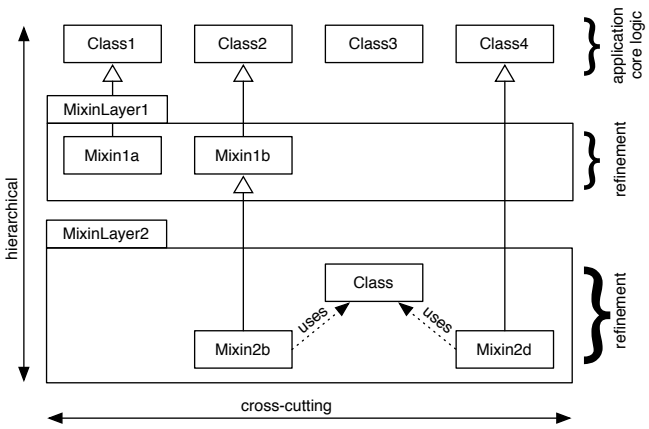

Figure 1: Mixin layers.

text information. Second, we additionally require a mechanism that *dynamically activates and deactivates mixin layers* accordingly during program execution.

Mixin layer selection and composition Our notion of this issue contrasts currentday practices with mixin layers at two levels. First, both the selection and composition of mixin layers are computed automatically based on context information. In current mixin layer implementations, this selection and composition is done manually at design time. Second, the composition of mixin layers evolves over time as the context changes. At present, compositions of mixin layers are not supposed to change at runtime and are therefore fixed at design time. Hence, we conclude that there exists a huge gap between existing practices where the composition of mixin layers does not change at runtime and the kind of dynamic composition mechanism that we require to reconfigure compositions at runtime according to context changes.

Dynamic layer activation Context-aware adaptations are accomplished by activating and deactivating mixin layers at runtime according to context changes. This pluggability can be achieved by redefining classes at runtime. Existing instances of redefined classes should be updated accordingly. On the one hand, this might look like a harsh requirement to implement in a static language like Java. On the other hand, by using the reflective capabilities of dynamic languages such as CLOS or Smalltalk, it is much more straightforward to perform class redefinitions at runtime.

The following section presents a concrete implementation of the cellular phone example from Section 2. This implementation is continued in Sections 6 and 7 that respectively discuss the mixin layer selection and composition issue and dynamic layer activation.

5. EXAMPLE IMPLEMENTATION

In an attempt to implement the cellular phone example using mixin layers, we encountered many programming languages that provide different flavours of mixin layers like CaesarJ [3], Classboxes in Smalltalk [4] and LasagneJ [9]. We chose to exploit ContextL [5] which is written in Common Lisp [2]. This implementation introduces the notion of so-called layers as an extension of the Common Lisp Object System. Our choice for this implementation is driven by the fact that ContextL supports a flexible and efficient mechanism for activating and deactivating its layers [6]. The importance of this issue is discussed in Section 7. The remainder of this section explains how the context-dependent adaptations of the cellular phone example can be implemented in ContextL.

5.1 Application core logic

We first take a closer look at the application core logic of the cellular phone. Only the classes `phonecalls` and `messages` are subject to refinement. We therefore define them as layered classes.

```
(define-layered-class phonecalls () ...)
(define-layered-class mesages () ...)
```

Layered classes provide the opportunity to define layered methods. Such methods can be refined with layers later on. In this example, the most important layered methods are `accept-call` and `receive-message` which are respectively part of the layered classes `phonecalls` and `messages`. We omit the actual implementation of the layered methods because this is not relevant here.

```
(define-layered-method accept-call
  ((p phonecalls) nr)
  ...)

(define-layered-method receive-message
  ((m messages) nr text)
  ...)
```

We additionally provide the class `cellphone` that acts as a facade for the classes of the application core logic. All incoming communication passes through the generic function `in`. This can be either a phonecall (represented by the structure `phonecall`), or a message (represented by the structure `message`). These structures contain information like caller-id, date, time, etc.

```
(defmethod in ((c cellphone) (p phonecall))
  (accept-call (get-phonecalls c) (phonecall-nr p)))

(defmethod in ((c cellphone) (m message))
  (receive-message (get-messages c) (message-nr m)
                   (message-text m)))
```

5.2 Layer definitions

We now define the `ignore-layer`, `answermachine-layer` and `redirect-layer` in ContextL for the *IgnoreAdaptation*, *AnswerMachineAdaptation* and *RedirectAdaptation* respectively.

```
(deflayer ignore-layer)
(deflayer answermachine-layer)
(deflayer redirect-layer)
```

The behavioural part of the *IgnoreAdaptation* ignores and logs phone calls from callers that are not classified as VIP. We implement this behaviour with a layered method that is part of `ignore-layer`. The latter is specified with the `:in-layer` keyword. The predicate `vip-p` checks whether the phone number `nr` is classified as a VIP contact. All ignored calls are logged in the journal with the method `add-log`.

```
(define-layered-method accept-call
  :in-layer ignore-layer ((p phonecalls) nr)
  (if (vip-p (get-contacts p) nr)
      (call-next-method)
    (add-log (get-journal p) 'ignored nr)))
```

The activation of `ignore-layer` yields the design presented in Figure 2.

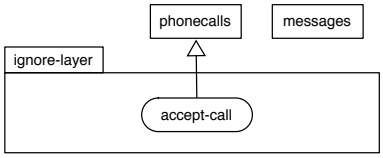

Figure 2: System design if battery level is low.

The *AnswermachineAdaptation* is cross-cutting since it affects the layered classes `phonecalls` and `messages`. The method `get-voice-message` records and returns the voice message of the caller. The method `send-message` is used to implement the auto-reply of incoming messages.

```
(define-layered-method accept-call
  :in-layer answermachine-layer ((p phonecalls) nr)
  (let ((msg (get-voice-message)))
    (add-log (get-journal p) 'auto nr msg)))

(define-layered-method receive-message
  :in-layer answermachine-layer
    ((m messages) nr text)
  (send-message m nr "I will reply asap.")
  (call-next-method))
```

Figure 3 illustrates how layers can be combined in an inheritance hierarchy. This example is an application of *PolicyRule II* that allows the combination of `ignore-layer` and `answermachine-layer`. Such composition yields the following behaviour: only callers that are classified as VIP get in touch with the answering machine. All other calls are ignored and logged in the journal. Messages always receive an auto-reply.

Finally, the implementation of the *RedirectAdaptation* is as follows. The method `redirect-call` forwards the incoming calls to the secretary `*secr*`.

```
(define-layered-method accept-call
  :in-layer redirect-layer ((p phonecalls) nr)
  (redirect-call p nr *secr*))
```

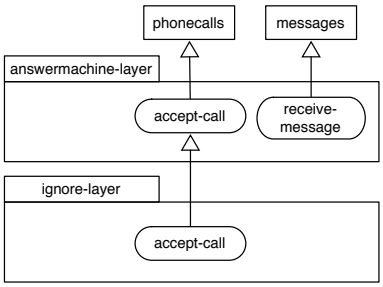

Figure 3: System design if battery level is low and time is midnight.

6. LAYER SELECTION AND COMPOSITION

The context information of context-aware systems is not known beforehand and might change over time (e.g. location of user). We therefore require a mechanism that selects and defines layer compositions automatically at run-time based on actual context information. This task is accomplished by means of a forward chainer. The production rules of such a reasoning system describe when layers are applicable and how they can be composed. We identify three categories of rules which are discussed in Sections 6.1, 6.2 and 6.3 respectively.

TransformationRules transform primitive sensor data into meaningful application-specific context information.

SelectionRules associate the conditional part with the behavioural part (i.e. layers) of an adaptation.

CompositionRules generates valid compositions of layers based on the user-defined policy rules. For example, the cellular phone scenario of Section 2 has four policy rules that precisely describe what should happen if multiple layers coexist.

We employ an efficient forward chainer called LISA [12] to perform the reasoning job.

6.1 Deducing high-level context information

The following *TransformationRules* deduce meaningful context information for the application (e.g. `(battery (level low))`) out of low-level sensor information (e.g. `(battery (percentage 15))`). LISA provides the facilities to group relevant rules in so-called rule-contexts. For example, the following rules are all part of the rule-context `"in"` which stands for incoming communication. The body of a LISA-rule consists of a condition and an action part which are situated before and after the `=>` symbol respectively. Variables are indicated with a leading question mark.

```
(defrule battery-low (:context "in")
  (?inst (battery (percentage ?x (< ?x 20))
                  (level (not low))))
  =>
  (modify ?inst (level low)))

(defrule time-night (:context "in")
  (?inst (time-of-day (hour ?x (or (>= ?x 23)
                                   (< ?x 8)))
                      (period (not night))))
  =>
  (modify ?inst (period night)))

(defrule location-meeting (:context "in")
  (?inst (location (longitude 5) (latitude 13)
                   (place (not meeting))))
  =>
  (modify ?inst (place meeting)))
```

This is only a light-weight approach in comparison with contemporary context modelling and reasoning tools like [10] and [11]. Since our major concern is language design for context-aware systems, we intentionally do not devote special attention to the modelling issue.

6.2 Layer selection

We are now able to decide which mixin layers should be activated according to the current (high-level) context information. This is done by the *SelectionRules* which are also grouped in the rule-context `"in"`. They represent the adhesive means between the context conditions on the one hand and the context-dependent behaviours (i.e. layers) on the other hand.

```
(defrule ignore-adaptation (:context "in")
  (battery (level low))
  =>
  (assert (layer (name ignore-layer))))

(defrule answermachine-layer (:context "in")
  (time-of-day (period night))
  =>
  (assert (layer (name answermachine-layer))))

(defrule redirect-layer (:context "in")
  (location (place meeting))
  =>
  (assert (layer (name redirect-layer))))
```

6.3 Composing adaptations

The final step consists of composing the selected layers of the previous step in a linear inheritance hierarchy. We therefore added a `before` slot to each `layer` fact in LISA. In this way, we can unambiguously define the order between the layers using *CompositionRules* which implement the policy rules of the cellular phone example. For instance, the following LISA rule `compose-2` implements *PolicyRule II*. It says that, if both `ignore-layer` and `answermachine-layer` are selected, `answermachine-layer` should appear before `ignore-layer`.

```
(defrule compose-2 (:context "in")
  (layer (name answermachine-layer))
  (?y (layer (name ignore-layer)
             (before (not answermachine-layer))))
  =>
  (modify ?y (before answermachine-layer)))
```

PolicyRule IV gives priority to `answermachine-layer` if both `redirect-layer` and `answermachine-layer` coexist. This is implemented as follows.

```
(defrule compose-4 (:context "in")
  (?x (layer (name redirect-layer)))
  (layer (name answermachine-layer))
  =>
  (retract ?x))
```

7. DYNAMIC LAYER ACTIVATION

Once the appropriate layer selection and composition is determined by the reasoning system, we need to activate these layers dynamically. ContextL supports an efficient mechanism to activate layers at runtime within a dynamic scope. Unfortunately, the current implementation requires that the layer composition is specified manually at design time. We therefore extend ContextL with a construct that computes the layer composition automatically using the LISA rules from Section 6. For example, a context-aware incoming phone call looks as follows.

```
(with-current-context ("in")
  (in *cellphone* *phonecall*))
```

The `with-current-context` macro carries out the following steps. First, low-level sensor information is added to the facts database of LISA. Next, the forward chainer evaluates all rules of the rule-context `"in"`. The parameter list of `with-current-context` might contain an arbitrary number of rule-contexts. The result of the reasoning process is a list of layers that describes the linear inheritance hierarchy. Finally, these layers are applied to all objects of the current thread within the dynamic scope of `with-current-context`. This approach ensures that all method calls within the boundary of the dynamic scope will be subject to the active layers and all other method calls remain unaffected. We therefore consider dynamic scoping in this case as a powerful means to ensure consistent program behaviour.

8. LAZY LAYER ACTIVATION

The current implementation of `with-current-context` suffers from a substantial overhead in cases where some layers are activated that are not relevant within the dynamic scope of `with-current-context`. We illustrate the problem with a simple example.

8.1 Extended cellular phone example

We extend the functionality of the cellular phone from Section 2 to support multiple connection types for outgoing communication: wifi, bluetooth, and default mobile connection. The connection types bluetooth and wifi are context-dependent adaptations because they are only applicable if there is a wifi and/or bluetooth connection in the surrounding environment. If the user is trying to make a phone call while there is a wifi connection available, the system will first try to perform the phone call via VoIP. If this fails (e.g. the receiver is not online), the system will use the mobile connection instead. The cellular phone also supports message sending. In this case, the system will try bluetooth and/or wifi connection to send messages before using the

mobile connection. The main difference between the outgoing phone call and message is that outgoing phone calls only check for wifi connection, whereas outgoing messages check for bluetooth and/or wifi connection. This policy is followed because the bluetooth connection type is not suited for phone traffic.

The application core logic of the cellular phone always employs the default mobile connection for making phone calls or sending messages. In addition, we define two mixin layers `bluetooth-layer` and `wifi-layer` that implement the context-dependent behaviour of bluetooth and wifi connection type respectively. These layers are only applicable if their signal level is higher than 20 percent. The user policy states that if both bluetooth and wifi connection type are available, wifi connection type is tried first. All this is shown in the following *Selection- and CompositionRules*.

```
(defrule bluetooth (:context "out")
  (bluetooth (signal ?x (> ?x 20)))
  =>
  (assert (layer (name bluetooth-layer))))

(defrule wifi (:context "out")
  (wifi (signal ?x (> ?x 20)))
  =>
  (assert (layer (name wifi-layer))))

(defrule compose (:context "out")
  (layer (name bluetooth-layer))
  (?x (layer (name wifi-layer)))
  =>
  (modify ?y (before bluetooth-layer)))
```

The following functions `perform-call` and `send-message`, which are respectively part of the layered classes `phonecalls` and `messages`, deal with the outgoing communication of the cellular phone. (This example does not deal with the actual implementation of performing phone calls or sending messages. Instead, we just print some informative sentences on the screen.)

```
(define-layered-method perform-call
  ((p phonecalls) nr)
  (print "phone call via mobile network"))

(define-layered-method send-message
  ((m messages) nr text)
  (print "message via mobile network"))
```

Next, we implement the `wifi-layer`. This layer attempts to employ the available wifi network to make phone calls or sending messages. We assume the existence of the method `contact-available-p` that checks whether the receiver is available via wifi connection type.

```
(deflayer wifi-layer ()
  ((connection :accessor wifi-connection)))

(define-layered-method perform-call
  :in-layer wifi-layer ((p phonecalls) nr)
  (if (contact-available-p
       (wifi-connection
        (find-layer 'wifi-layer)) nr)
      (print "performing phone call via wifi")
    (call-next-method)))

(define-layered-method send-message
  :in-layer wifi-layer ((m messages) nr text)
  (if (contact-available-p
       (wifi-connection
        (find-layer 'wifi-layer)) nr)
      (print "sending message via wifi")
    (call-next-method)))
```

The `bluetooth-layer` looks very similar, but is limited to messages.

```
(deflayer bluetooth-layer ()
  ((connection :accessor bluetooth-connection)))

(define-layered-method send-message
  :in-layer bluetooth-layer ((m messages) nr text)
  (if (contact-available-p
       (bluetooth-connection
        (find-layer 'bluetooth-layer)) nr)
      (print "sending message via bluetooth")
    (call-next-method)))
```

Similar to the generic function `in` that deals with all incoming communication, we implement `out` that deals with all outgoing communication (`phonecall` or `message`).

```
(defmethod out ((c cellphone) (p phonecall))
  (perform-call (get-phonecalls c)
                (phonecall-nr p)))

(defmethod out ((c cellphone) (m message))
  (send-message (get-messages c) (message-nr m)
                (message-text m)))
```

Finally, the code for making phone calls or sending messages looks as follows. The variable `*phone*` is an instance of the class `cellphone` which represents the cellular phone system. The variable `*x*` can be an object of type `phonecall` or `message`. The generic function `out` takes care of the type dispatching.

```
(with-current-context ("out")
  (out *phone* *x*))
```

8.2 Efficiency issue

The system always needs to check and possibly activate the `bluetooth-layer` or `wifi-layer`. Although, if `*x*` is of type

phonecall, the bluetooth-layer is not of interest. Hence, this approach suffers from a potential overhead. One might suggest to overcome this problem by placing the layer activations in the respective method specializers and putting the wifi and bluetooth production rules in different rule-contexts. The following modified code shows what this alternative approach looks like.

```
45 (defrule bluetooth (:context "bluetooth-out") ...)
46 (defrule wifi (:context "wifi-out") ...)
47
48 (defmethod out ((c cellphone) (p phonecall))
49   (with-current-context ("wifi-out")
50     ...))
51
52 (defmethod out ((c cellphone) (m message))
53   (with-current-context ("bluetooth-out" "wifi-out")
54     ...))
55
56 (out *phone* *x*)
```

This approach is not desirable because of the following two reasons. First, the programmer is responsible for doing some abstract interpretation about the source code in order to decide where and which rule-contexts should be activated to avoid overhead. Depending on the complexity and the size of the context-aware system, this job can become unmanageable. Second, from a language engineering perspective, it is not advised to pollute code with scattered layer activations (line 49 and 53) for avoiding overhead only. The pollution makes the code difficult to comprehend and extend because there are implicit relationships between the scattered layer activations. These implicit relationships become visible in cases where we, for example, want to deactivate the context-awareness of the outgoing communication.

8.3 Delaying layer activation

The essence of the previous problem is that the selection, composition and activation of layers is accomplished before the body of with-current-context is executed. This can be solved by delaying the layer activations until it is required. We call this approach *lazy layer activation* inspired by the notion of lazy evaluation.

We explain the semantics of this approach using the extended cellular phone example from Section 8.1. Consider for example that the variable *x* is of type message in line 44. The layer selection, composition and activation is delayed until the layered method send-message is called in line 41. This method can be adapted with both the wifi-layer (lines 19-25) and bluetooth-layer (lines 29-35). We employ the LISA rules to determine whether these mixin layers should be activated according to the current context information. Let us assume that there is only a bluetooth connection available in the surrounding environment. From now on, the bluetooth-layer remains active for all subsequent method calls within the dynamic scope of with-current-context (line 43).

The delayed layer selection requires a backward chainer to avoid computational overhead of the reasoning system. For example, when the control flow reaches the layered method send-message in line 41, there are only two candidate layers bluetooth-layer and wifi-layer. A backward chainer determines the applicability of these candidate layers by trying to satisfy their context conditions. The way how the layer composition is computed remains unchanged, a forward chainer is still the ideal candidate for this task.

9. RELATED WORK

This section discusses two approaches that deal with dynamic adaptation of system behaviour and briefly explains the differences with our approach.

Amano et al. propose the *LEAD++* description language [1] to support dynamic adaptability of software systems driven by changing runtime environments (i.e. context information). To this end, LEAD++ introduces the mechanism of so-called *adaptable procedures* and adaptable methods which are very similar to *generic functions* and methods in the Common Lisp Object System (CLOS). The main difference between both is that adaptable procedures have a more fine-grained method dispatch. LEAD++ contrasts our approach in the following ways. First, adaptable procedures have no explicit support to deal with cross-cutting modularisation. Next, the dispatch conditions are written in an operational style and reside in the adaptable method definitions. The latter design choice makes it hard to make collaborative decisions among multiple adaptable procedures. On the contrary, we distinctly separate the conditional (i.e. declarative rules) and behavioural (i.e. mixin layers) part of adaptations in order to support reusability, extensibility, and cooperation. Finally, the behavioural variations in LEAD++ are realised by means of reflection whereas we employ mixin composition.

The *Chisel* dynamic adaptation framework [7] of Keeney et al. addresses the problem of unanticipated policy-driven, context-aware dynamic adaptation of software systems. The authors promote the use of a meta-object protocol (MOP) to accomplish unanticipated dynamic adaptations. More concretely, they developed a so-called *meta-level adaptation manager* that performs run-time switches between metatypes of (base level) service objects in response to events. The decisions of this adaptation manager are guided by the Chisel policy language. The latter associates metatypes of service objects with run-time events (i.e. relevant context changes) using logical rules. This approach suffers from the following issues. First, the computational cost of continuously polling, reasoning and switching between metatypes at run-time is unclear. Consider the case in which Chisel continuously switches the metatype of a seldom used network protocol service object. In our approach, we limit the context acquisition and reasoning to well-specified places in the source code. Second, the policy language has no disciplined way to control the scope of the adaptations. For example, with regards to our cellular phone illustration, once the decision has been made of how to respond to a phone call, the system should be loyal to that decision until the phone call finishes. Such functionality cannot be implemented in the Chisel framework without workarounds like e.g. manually adding locks. In contrast, we confine adaptations to dynamic scopes in the program execution. Finally, there is no explicit support to express relationships between metatypes

like e.g. coexistence or mutual exclusion.

10. POSITION STATEMENT

This paper envisions context-aware scenarios in terms of context-dependent adaptations that can be applied to the application core logic of a system. Since traditional implementation techniques like design patterns and conditional statements easily lead to cumbersome designs, we advocate that context-aware systems require more dedicated language constructs. To this end, we introduce mixin layers as a suitable candidate to implement context-dependent adaptations while keeping the overall design lucid.

We have several reasons to believe that mixin layers provide better means in comparison with traditional approaches to develop context-aware systems. First, mixin layers go beyond the modularisation capabilities of design patterns since they support cross-cutting modularisation to a certain degree. Second, mixin layers are singletons and their application to objects do not require object instantiation. Third, mixin layers are not confined to a static inheritance hierarchy because the mixins have no fixed superclasses. All these characteristics enable the developer to focus on the implementation of the application logic without suffering from integrating "bookkeeping code" in the design to deal with context-aware issues.

Another important property of our approach is that context information is explicitly available in the system as declarative rules. In this way, the mixin layer selection and composition can be computed efficiently using a reasoning system. Moreover, the centralised context information considerably supports program comprehension and extensibility. In addition, we propose a more advanced layer selection and composition mechanism, called lazy layer activation, that reduces computational overhead of the reasoning system.

11. REFERENCES

[1] N. Amano and T. Watanabe. Lead++: An object-oriented reflective language for dynamically adaptable software model. *IEICE Trans. Fundamentals*, E82-A(6):1009–1016, 1999.

[2] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. 1996.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. *Transactions on AOSD I, LNCS*, 3880:135 – 173, 2006.

[4] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference*, volume 2789, pages 122–131. Springer-Verlag, 2003.

[5] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming - an overview of contextl. *Dynamic Languages Symposium, co-located with OOPSLA*, 2005.

[6] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the Joint Modular Languages Conference (to appear)*. Springer LNCS, 2006.

[7] J. Keeney and V. Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 3–14, 2003.

[8] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[9] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May 12–19 2001.

[10] A.-Y. Turhan, T. Springer, and M. Berger. Pushing doors for modeling contexts with owl dl a case study. In *PERCOMW '06: Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, page 13, Washington, DC, USA, 2006. IEEE Computer Society.

[11] D. Wagelaar and V. Jonckers. Explicit platform models for mda. In *MoDELS*, pages 367–381, 2005.

[12] D. E. Young. Lisp-based intelligent software agents `http://lisa.sourceforge.net`, 2006.

Demonstration Abstracts

Made in MAKAO

Bram Adams

Bram.Adams@UGent.be
http://users.ugent.be/~badams/makao/
Ghislain Hoffman Software Engineering Lab
INTEC, Ghent University, Belgium

ABSTRACT

Software systems do not solely consist of source code: various other types of artifacts play a role, notably the build system. Although nothing stands as close to the source code or lends itself better as a starting point to explore a system's high-level architecture, few techniques or tools recognize or exploit this. Still, as each considerable change to a software system potentially demands modifications of build-related files, maintainers and developers alike need to find their way around quickly. We present MAKAO (Makefile Architecture Kernel for Aspect Orientation), a re(verse)-engineering framework for build systems, as a means to extract as much knowledge from the build system as possible and to help solving typical build-related problems. MAKAO offers a DAG (Directed Acyclic Graph)-based view of a build system, supporting visualization, querying and (dynamic) modification.

1. BUILD SYSTEMS

Until 1977, ad hoc build and install scripts were used to automate the build process of software systems. Everything changed when Feldman presented "make" [1], the most influential software build tool ever. He proposed a declarative way of specifying the dependencies between targets (executables, object files, libraries, source files, etc.), whereas the recipe to build a target was written as an imperative list of commands and macros. The "make" interpreter relied on the observation that a target only needs to be (re)built by its recipe if at least one of its dependencies is newer. This greatly improved incremental compilation of software projects.

Later on, portability of software required configurability of both source code and build scripts. Figure 1 illustrates this. Configuration scripts written in e.g. GBS (GNU Build System)[1] describe the build process from a high-level perspective, abstracting away from platform-specific configuration issues. Afterwards, they generate the actual build scripts that will perform the ground work. The combination of both is called the build system.

Build systems play a crucial role, as various stakeholders interact with it, each with their own interests and problems:

developers Assess the effects of their code or, if the build did not succeed, try to find out what error was the culprit. When adding new source code, they want to find out where they need to change something.

maintainers Want to learn the inner mechanics of a new system, check if there is dead code, profile things, etc.

(power) users Try to find out what library dependencies they need to compile and run the software.

QA division Want to add feature and regression tests and run them as quickly as possible.

[1] `http://sources.redhat.com/autobook/`

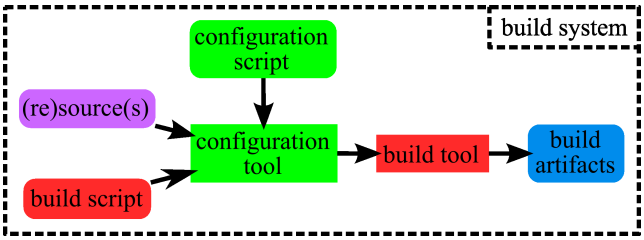

Figure 1: High-level view of build systems.

researchers Try to uninvasively integrate experimental tools.

As a consequence, the build process implicitly contains valuable information about all facets of the software itself. We would like to extract this inherent build system knowledge and make it available in an explicit form to all stakeholders. This way, they can deal more succinctly with their problems and learn new things about the software system's architecture. This enhances the data gained by existing reverse-engineering techniques for source code.

In the remainder of this abstract, we will present MAKAO (section 2) and apply it on the case study of [3] (section 3).

2. MAKAO

The observations of the previous section made us think about an extensible visualisation and re(verse)-engineering framework for build systems, which we named the Makefile Architecture Kernel for Aspect Orientation (MAKAO).

2.1 Design

MAKAO's philosophy is to make the implicit explicit. The declarative nature of dependency specifications is a good thing, but these are typically spread over hundreds of files and composed in some unclear way (e.g. recursively [2]). A visual representation combined with a powerful querying facility would be able to clarify this. However, more invasive problems like the addition of new tools (see section 3) also require re-engineering of the build system. This involves e.g. introducing new build targets, adding extra dependencies to existing targets or modifying targets' recipes. The design of MAKAO takes both the reverse as well as the re-engineering functionality into account.

Inspired by the ideas of aspect-oriented programming (AOP), MAKAO is composed of the following four components:

Explorer (Visually) Explore a representation of the build system.

Finder Query for targets and commands based on properties.

Adviser Write modifications for targets' dependencies and recipes.

Weaver Apply modifications both logically (in-memory representation) and physically (build and configuration scripts).

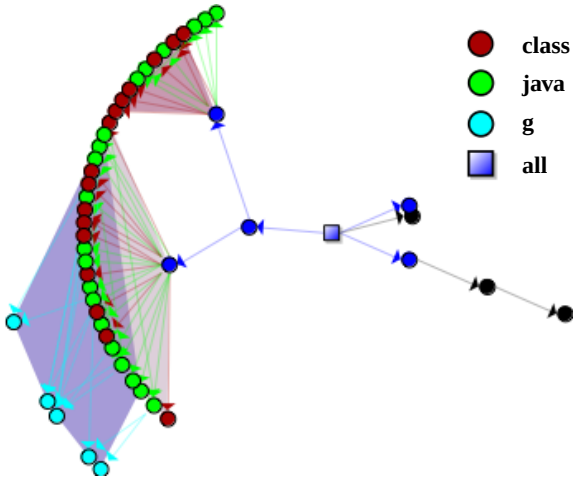

Figure 2: Dependency graph of build of Aspicere.

2.2 Data model

We opted for a Directed Acyclic Graph (DAG) as the underlying data model of a build run, based on the following observations:

- DAGs form the underlying model of "make" [1], and hence of most of its successors.
- Graphs have a natural visual representation and can be easily modified.
- Configuration scripts define in fact templates for source code and build scripts.
- Stakeholders, however, are mostly confronted with instantiated, platform-specific build scripts and code.

Providing only static views of a build system would not be useful, as the templates are too general and hence too hard to navigate or understand. Instead we chose, influenced by the fourth observation, to represent dynamic traces of concrete builds, but with links back to the static build data. Basically, while performing a typical build the build tool's internally constructed dependency graph is extracted and fed into MAKAO. In practice, this can be obtained by using a modified "make" like "remake"[2] or e.g. by capturing and post-processing the debug output produced by the build tool.

2.3 Implementation

Fig. 2 shows the dependency graph of a full build of Aspicere [3], as seen in MAKAO's main panel. We implemented MAKAO on top of GUESS[3], a graph exploration tool with an embedded Jython-based scripting language (called Gython). Targets (nodes) have directed edges to their dependencies. Each build script is shown as a colored convex hull around all of its targets, unless the hull degenerates to e.g. one node.

An additional "legend" panel, pasted on the graph here for the reader's convenience, outlines the various known build concerns in use (".java", ".o", etc.) and assigns a color to them. Unknown concerns are colored black by default, while the dark blue targets really represent the same concern as the square (starting) node's one. Nodes, edges and hulls are simply objects and can be easily queried and manipulated in the scripting console (not shown).

3. EXAMPLE

We will now show MAKAO's use by applying it on an issue we encountered during a reverse-engineering experiment [3] using Aspicere, our aspect language for C. To weave a tracing aspect into the code base, we had to run Aspicere's source preprocessing weaver right before each file's compilation. Doing this by replacing the relevant commands by a wrapper script around the aspect weaver did not resolve subtle issues like two or more wrappers invoking each other. Manually modifying the makefiles seemed inevitable back then, but we will now revisit this case with MAKAO.

First, we build the unaltered system while we extract the constructed dependency graph. After loading this graph into MAKAO, the Explorer-component allows us to verify that there are indeed C source files and that they apparently reside in a small number of directories. We are interested in those targets T directly depending on (i.e. processing) C files and we want to insert calls to Aspicere's weaver in T's recipe before the (sole) source-processing command. We write the following queries in the Finder-component:

```
1 T_list=[e.getNode1() for n in (concern=="c")
                       for e in n.getInEdges()   ]
3 base=[(command,tool) for T in T_list
                       for command in commands[T]
5                      for tool in ["gcc","esql"]
                       if command.find(tool)!=-1 ]
```

On line 1, we select all source nodes of edges leading to C targets, presuming that there are no duplicates. Then we try to find each T's sole source-processing command in its recipe using a Gython list comprehension. The tools we mention in this query could have been discovered earlier on by querying too. Now, we can write advice using the Adviser:

```
  before_advice=
7     ["\n".join([c.replace(t,t+" -E -o ${<}"),
                  "aspicere.sh ${<}"])
9           for (c,t) in base                  ]
```

The `join` function concatenates invocations of the selected tool in preprocessing mode and of Aspicere's weaver. Finally, the Weaver should weave all these concatenated commands in the recipes of the T targets as before advice, both in MAKAO's memory representation as in the proper build and/or configuration scripts:

```
9 weave_before(T_list,[c for (c,t) in base],
               before_advice)
```

One can now run the modified build scripts, check the scripts themselves or issue some new queries on MAKAO's memory model to see whether any relevant targets were skipped.

4. CONCLUSION

Build systems are inherently part of and tied to software systems. They offer valuable architectural information to various stakeholders. To facilitate quick understanding and clever modifications, MAKAO offers visualisation and flexible manipulation of both build structure and behavior inspired by aspect-oriented techniques.

Acknowledgements

The author wants to thank Kris De Schutter, Andy Zaidman and Herman Tromp for their support and (cumulative) wisdom.

5. BIBLIOGRAPHY

[1] S. I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 1979.

[2] P. Miller. Recursive make considered harmful, 1997.

[3] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006.

[2] `http://bashdb.sourceforge.net/remake/`
[3] `http://graphexploration.cond.org/`