FOAL 2003 Proceedings

Foundations of Aspect-Oriented Langauges Workshop at AOSD 2003

Gary T. Leavens and Curtis Clifton (editors)

TR #03-05
March 2003

Keywords: Aspect-oriented programming, modular reasoning, alternating transition systems, composition, inter-aspect dependencies, AspectJ, aspect classification, aspect reuse, composition graphs, modeling, modular transition systems, aspect inheritance, TinyC^2, code instrumentation, compilers, interpreters, denotational semantics, superimposition, model checking, Bandera, verification aspects, superimposition validation, binding interference

2000 CR Categories: D.1.m [*Programming Techniques*] Miscellaneous—aspect-oriented programming, reflection; D.2.1 [*Software Engineering*] Requirements/Specifications—languages, methodology, theory, tools; D.2.4 [*Software Engineering*] Software/Program Verification—class invariants, correctness proofs, formal methods, programming by contract, reliability, validation; D.3.1 [*Programming Languages*] Formal Definitions and Theory—semantics; D.3.3 [*Programming Languages*] Language Constructs and Features—control, data types and structures; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs—assertions, logics of programs, pre- and post-conditions, specification techniques; F.3.m [*Logics and Meaning of Programs*] Miscellaneous—reasoning about performance.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Table of Contents

Preface ii

Composition Graphs, a Foundation for Reasoning about Aspect-Oriented Composition 1
István Nagy, *University of Twente*
Mehmet Aksit, *University of Twente*
Lodewijk Bergmans, *University of Twente*

A Formal Model for Cross-cutting Modular Transition Systems 9
Henny B. Sipma, *Stanford University*

On Composition and Reuse of Aspects 17
Jörg Kienzle, *McGill University*
Yang Yu, *McGill University*
Jie Xiong, *McGill University*

TinyC^2: Towards Building a Dynamic Weaving Aspect Language for C 25
Charles Zhang, *University of Toronto*
Hans-Arno Jacobsen, *University of Toronto*

Interference Analysis for AspectJ 35
Maximilian Störzer, *University of Passau*
Jens Krinke, *University of Passau*

Compositional Reasoning About Aspects Using Alternating-time Logic 45
Benet Devereux, *University of Toronto*

Model Checking Applications of Aspects and Superimpositions 51
Marcelo Sihman, *Technion–Israel Institute of Technology*
Shmuel Katz, *Technion–Israel Institute of Technology*

Understanding AOP through the Study of Interpreters 61
Robert E. Filman, *NASA Ames Research Center*

Adding Superimposition to a Language Semantics 65
Ralf Lämmel, *CWI and Vrije Universiteit*

Preface

Aspect-oriented programming is an emerging paradigm in software engineering and programming languages that promises better support for separation of concerns. The second Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the 2nd International Conference on Aspect-Oriented Software Development in Boston, Massachusetts, on March 17, 2003. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including, but not limited to: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, theory of aspect translation (compilation) and rewriting, and applications of such theories in practice (such as language design studies). The call for papers welcomed all theoretical and foundational studies of this topic.

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest in the programming language theory communities concerning aspects and aspect- oriented programming languages.
- Foster interest in the formal methods community concerning aspects and aspect-oriented programming.

In addition, we hoped that the workshop would produce an outline of collaborative research topics and a list of areas for further exploration.

The papers at the workshop, which are included in the proceedings, were selected from papers submitted by researchers worldwide. Due to time limitations at the workshop, not all of the submitted papers were selected for presentation.

The workshop was organized by Gary T. Leavens (Iowa State University) and Curtis Clifton (Iowa State University). The program committee that selected papers consisted of Leavens and James H. Andrews (U. Western Ontario), William Cook (Allegis), Tzilla Elrad (Illinois Inst. of Technology), Ralf Lämmel (CWI and Vrije Universiteit), Oscar Nierstrasz (U. of Berne), Jens Palsberg (Purdue U.), Kris De Volder (U. of British Columbia), and Mitch Wand (Northeastern University). We thank the organizers of AOSD 2003 for hosting the workshop.

Composition Graphs: a Foundation for Reasoning about Aspect-Oriented Composition

- Position Paper -

István Nagy Mehmet Aksit Lodewijk Bergmans

TRESE Software Engineering group, Faculty of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
+31-53-489 3767

{ nagyist, aksit, bergmans }@cs.utwente.nl

ABSTRACT

Aspect-oriented languages offer new modularization concepts and composition approaches to provide more flexible solutions for the separation and integration of concerns. There are significant differences among aspect-oriented languages, due to the specific language constructs that they adopt. In this paper, we propose a common model, called Composition Graph, to represent different aspect-oriented approaches in a uniform way that can serve as a basis for the comparison of aspect-oriented languages. We also present a transformation language which can be used to model different weaving operations in our model.

1. INTRODUCTION

During the last several years, a considerable number of Aspect-Oriented Languages (AOLs) has been introduced. Some AOLs may be particularly suitable to program certain application categories. We think that in order to compare and evaluate AOLs, it is important to understand their underlying concepts.

An important characteristic of an AOL is its aspect composition mechanism. This is the mechanism to incorporate aspects with other aspects and/or with traditional programming abstractions.

In this paper, we focus on the aspect composition mechanisms of languages. To this aim, we introduce a generic model, called Composition Graph (CG), in which different aspect-oriented composition mechanisms can be expressed uniformly and can be compared with each other.

The structure of the paper is as follows. Section 2 presents a simple composition problem through an illustrative example. In section 3 we provide solutions to the problem in two different models, namely in AspectJ[1] and HyperJ[2]. Section 4 describes the approach. Section 5 outlines the notion of Composition Graphs exemplified by the solutions explained in the previous section. Section 6 demonstrates how the composition mechanisms can be represented by graph transformation rules. In section 7 we discuss some important related work. Finally, section 8 gives a conclusion and presents future work.

2. An Example Problem

AOLs use several composition techniques, such as *method composition*, *introductions*, *merging of different program elements*, etc. combined with new modularization concepts to cope with the phenomena of tangled code and crosscutting.

In this section, we introduce a method composition problem that we will use as an instructive example in the subsequent sections. This example is based on the Observer design pattern [3].

In Figure 1, class $Point_1$ implements a geometrical point with x and y coordinates as instance variables and get/set as methods. Class *Subject* is the part of the Observer pattern that maintains the list of observers for each subject, using the vector *observers*. This class is responsible for the notification of the observers by the method *Notify*.

```
class Point1 extends Object{
    private int _x, _y;

    void setX(int x){ _x=x; }
    int getX() { return _x; }

    void setY(int y){ _y=y; }
    int getY() { return _y; }
    ...
}

class Subject{
    private Vector observers;

    public Subject() { /* … */}

    public void attach(Observer o)
        { observers.add(o); }

    public void Notify()
        { /* foreach observer.update() */}
    ...
}
```

Figure 1. Definition of classes Point and Subject

Figure 2 displays a possible enhancement of class $Point_1$, labeled $Point_2$, to incorporate the subject role using inheritance. This class has the following responsibilities: a) After the execution of each method that changes the state of the object, the notification of the registered observers must take place. This is shown by the lines (2) and (3). b) This class inherits from class *Subject* to make the method *Notify* accessible for class *Point*[1]. As the source shows,

[1] Obviously, this is only one possible implementation of the Observer pattern.

```
class Point2 extends Subject{          (1)
    public void setX(int x)
        { _x=x; Notify();}             (2)
    public void setY(int y)
        { _y=y; Notify();}             (3)
    ...
}
```

Figure 2. Adaptation of $Point_1$ to support the Observer pattern

the adaptation of the subject role results in crosscutting code. To avoid this problem, other modularization and composition techniques should be used.

3. Aspect-Oriented Implementation of the Problem

In this section, we provide a simple aspect-oriented solution to the previous example both in AspectJ and HyperJ.

3.1 Composition in AspectJ

Figure 3 displays a possible implementation of the composition of class *$Point_1$* with class *Subject* in AspectJ.

Line (1) implements the language construct *introduction*. Here, the superclass of class *$Point_1$* is changed from the root class *Object* to the pattern defined class *Subject*. The pointcut specification shown in line (2) designates the methods *setX* and *setY*. In line (3) an *after advice* is bound to this pointcut specification. This means that the code "s.Notify()" specified in the advice will be performed after the execution of the designated methods.

```
aspect Notification{
    declare parents:
        Point1 extends Subject;            (1)

    pointcut stateChange(Subject s):
        this(s) &&
        execution(void Point.set*(..)); (2)

    after(Subject s): stateChange(s){  (3)
            s.Notify();
    }
}
```

Figure 3. Definition of the aspect Notification

This problem could be solved using more sophisticated features of AspectJ, such as abstract pointcuts [4]. For the sake of simplicity, however, we consider this solution adequate to explain the problem.

3.2 Composition in HyperJ

Figure 4 displays a HyperJ control file that implements an extension of class *$Point_1$* to integrate the subject role of the Observer pattern.

In line (1) we list the classes to be incorporated. The lines between (2) and (3) represent the *concern mapping*, where program entities are assigned to different hyperslices[2]. Here, class *Point* is assigned to the hyperslice *Feature.Kernel*, while class *Subject* is assigned to the hyperslice *Feature.Observing*. The hypermodule specification in line (3) consists of two important parts: identification of the hyperslices (4) that are to be integrated, and integration relationships (5). These specify the details of the desired composition. The line marked by (6) shows the general integration strategy that has to be specified. Finally, the operation *bracket* selects the methods to be composed from class *Point* (7) and specifies that the method *Notify* has to be performed after the execution of these methods (8).

```
-hyperspace
    hyperspace DemoHyperspace
    composable class test.*;              (1)
-concerns                                 (2)
    class Point1 : Feature.Kernel
    class Subject : Feature.Observing
-hypermodules                             (3)
  hypermodule ObserverDemo
    hyperslices:                          (4)
        Feature.Kernel,
        Feature.Observing;

    relationships:                        (5)
        mergeByName;                      (6)

        bracket "Point1"."set*"           (7)
        after                             (8)
         Feature.Observing.Subject.Notify();

  end hypermodule;
```

Figure 4. HyperJ control file

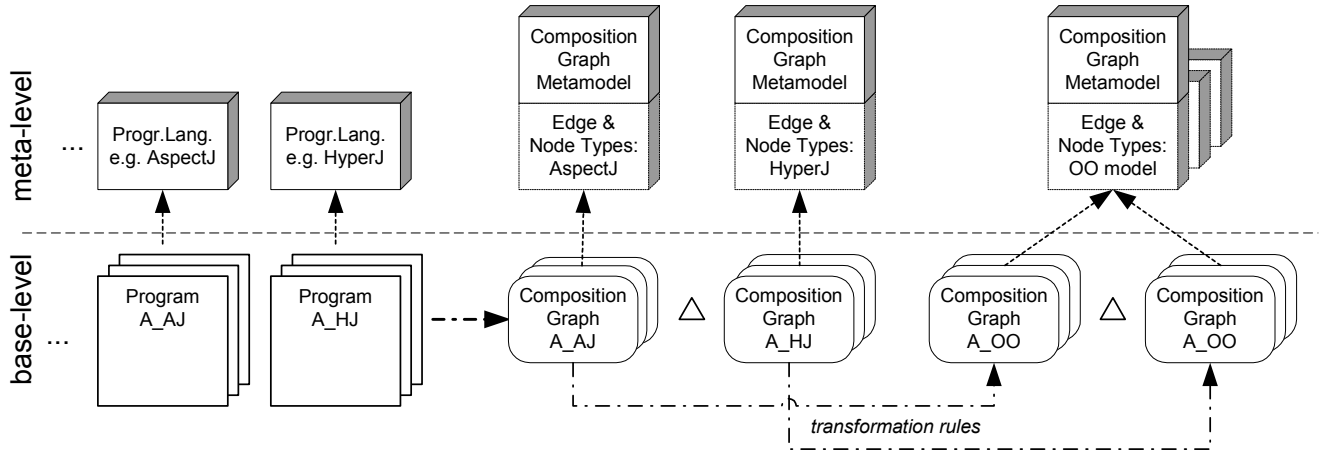
4. Our Approach

We explain our approach using the figure at the top of the next page. In this figure we can distinguish the lower base level and the meta level; the models at the base level are expressed in terms of the metamodels. We will discuss the picture from left to right, roughly corresponding to the general process of creating and transforming CGs.

On the left side, at the base level a number of boxes is shown which represent actual programs. Typically these programs can be represented by source code, byte code or an exchange format such as XML. Each individual program follows the rule of its programming language metamodel. The figure shows two example programming language metamodels: AspectJ and HyperJ. Our goal is to reason about the semantics of the programming languages, in particular their composition mechanisms. However, we choose to do so by considering the semantics and compositions of actual programs as well, rather than staying at the meta-level only.

Our approach is based on the application of a single metamodel which is capable of representing programs from a wide range of programming languages and paradigms: this is the Composition

[2] A more detailed specification of HyperJ can be found in [2].



Graph metamodel (the box appears repeatedly at the top right of the picture).

For example, imagine two versions of the same program A, each written in a different programming language (such as AspectJ and HyperJ): by translating these two programs into Composition Graph representations (these are the boxes in the middle of the bottom row of the figure), we can start to compare the structure of these programs, since they are represented in the same universal format. The differences between the programming languages are further visible through the different types of edges and nodes in each CG.

We expect a number of benefits from these representations of programs using CGs:

- Since CGs emphasize the (composition) structure and dependencies of programs, we may use them to reason about properties such as degrees of coupling and cohesion, e.g. by defining metrics.
- Since programs in different programming languages can be easily compared, we may be able to infer properties of the programming languages (in the form of "programming language 1 can express problem/program A with less coupling than programming language 2"). Note that making general assumptions based on one or a few concrete examples must be done with great care.
- We believe that the process of representing programs in the universal format, requiring one to define the composition structure of the programming language as types of nodes and edges, will yield increased insight in the workings and essence of aspect-oriented approaches, perhaps leading to new or generalized composition mechanisms.

A further step in defining and understanding the semantics of the composition mechanisms can be made by translating the program representations into CGs for a generic model: this could be a 'traditional' model such as the OO model, or alternatives such as a generic AOP model. Specifying the translation has several advantages:

1. It provides us insight into the ability to actually express a particular functionality, and how composition mechanisms really work.
2. If the resulting CGs are different, it will be fairly straightforward to see whether they are equivalent 'refactorings' of the same program, or in fact programs with (slightly) different semantics.
3. Defining general transformation rules, which can transform any CG in language A towards a CG in language B, is a way to define the precise semantics of the programming language[3].
4. Hence, the essential differences in composition mechanisms can be observed by looking at the differences between the transformation rules.

The remainder of this paper will focus on the concept and representation of composition graphs and transformation rules, exemplified by the example that we introduced in section 2 and 3.

5. Composition Graphs

Composition Graphs (CGs) are used to represent certain aspects of programs. They are especially useful to represent the structure of programs and reason about composition mechanisms.

CGs, like abstract syntax trees (ASTs), denote structural dependencies between different program units represented in the program. However, CGs are different from ASTs in several ways; they do not necessarily represent the full syntax of languages: certain parts of programs can be compressed into one node of the graph. CGs can also be used to explicitly represent certain composition relationships between various program units, such as classes, methods, advices, hyperslices, etc.

5.1 Structure of Composition Graphs

A Composition Graph consists of a set of nodes, labeled edges and attributes. Nodes represent the program units, which may be affected or used by the aspect weaving mechanism of the language considered. A node can refer to other nodes or attributes through labeled edges. An attribute refers only to its parent node and contains information about it.

Figure 5 depicts a part of the CG of class $Point_1$ which was shown in figure 1. Nodes are illustrated by small circles. The left uppermost node (1) denotes the whole class. Three attributes – illustrated by ovals - are connected to this node through the edges

[3] Note that the precision of this semantic specification depends on the level of abstraction of the target language.

name, *visibility* and *meta*. In corresponding order the first two attributes are the name and visibility of the class, while the third one is a meta-attribute. Each node can have a special edge called *meta* that holds meta-information about the type of the node.

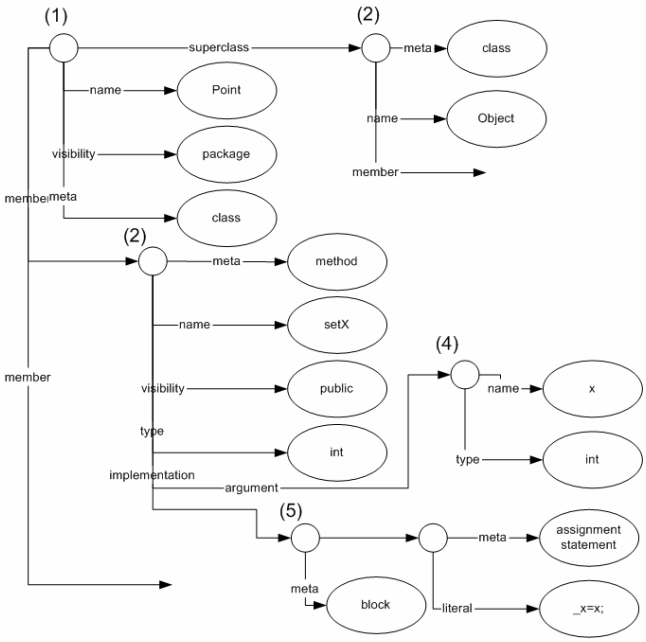


Figure 5. Part of class Point represented as a Composition Graph

The node marked by (1) has two edges that are connected to two other nodes. The edge with the label *member* refers to the node (2), which represents the method *setX* of class *Point*. This node has also some attributes (*meta*, *name*, *visibility*, *type of return value*) and relations with the two other nodes: the upper one (4) corresponds to the argument of the method, while the node marked by (5) denotes the implementation (body) of the method. This latter node has a meta attribute and an edge, which is connected to an assignment statement. This is the only statement of the method. The edge called *superclass* refers to a node (3) that denotes class *Object*, the superclass of class *Point*. Due to lack of space, we have not unfolded this node completely; This node is in fact a subgraph that has similar structure to the subgraph denoted by (1).

Note that figure 5 shows only the part of the Composition Graph of class *Point*. Other methods are represented like the method *setX*, but are not shown.

The same type of representation can be applied for aspect-oriented languages. Figure 6 illustrates a part of the aspect *Notification* as a CG. The node marked by (1) corresponds to the *introduction* statement in figure 3. Here, the introduction statement is represented as a literal. This is a way to hide the details if necessary. In fact, this node could have been expanded to several nodes as it is illustrated by the node marked by (4). The node at (2) illustrates the pointcut specification and also shown in a compressed form. The node marked by (3) illustrates the advice, which was shown in line (3) of figure 3.

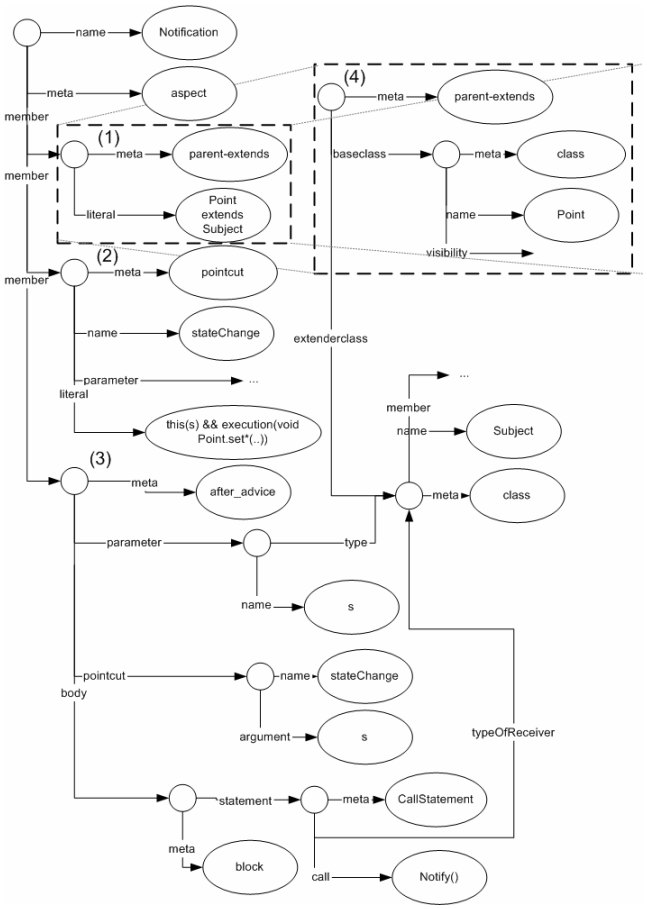


Figure 6. Part of the aspect *Notification* shown as a CG

5.2 Setting up Composition Graphs

Composition Graphs can be derived from various software artifacts, such as programs expressed in different languages (Java, AspectJ, HyperJ), XML documents and UML models.

As a first step, the files that contain the source code have to be parsed to build up their syntax tree.

In the next step, the syntax-tree is transformed into an initial CG by adding the cross-reference relationships as edges where necessary. For instance, in figure 5 the edge *superclass* is a typical cross-reference relationship. In a syntax-tree, the name of the superclass is an identifier, whereas in the CG, the relation *superclass* denotes to the actual representation of that class (see figure 5). In other words, in CGs every program unit which is relevant from the point of view of weaving is uniquely represented.

The third important step is the resolution of the nodes that contain composition (weaving) specifications. These are represented in CGs through additional edges and/or nodes. Figure 7 illustrates the aspect *Notification* in this way. Three new edges – illustrated by the broken arrows - are shown in figure 7. Edges marked by (1) and (2) represent the combination of the *after advice* with the methods *setX* and *setY*. The edge marked by (3) represents the *introduction*, which was shown in figure 3.

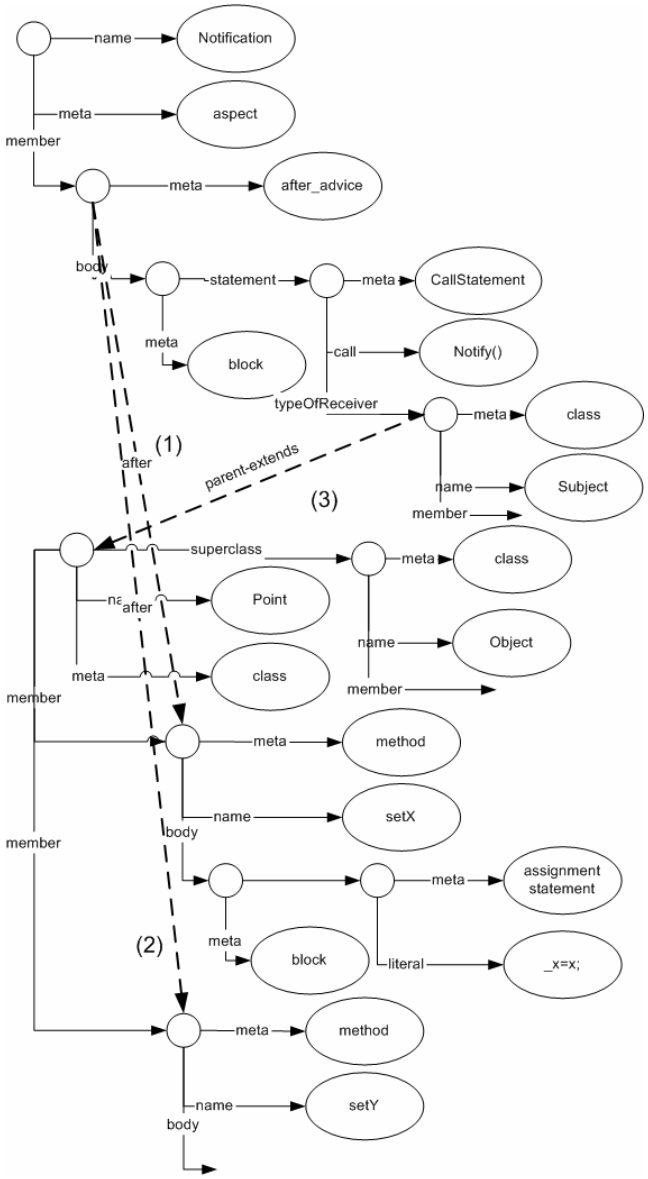


Figure 7. CG of the AspectJ program after the third step

Figure 8 shows the CG representation of the hypermodule *ObserverDemo*, which was described in figure 4. The edge marked by (1) is for the *mergeByName* relationship between the two hyperslices. The *bracket* relationship is represented by three new edges. The first two edges marked by (2) and (3) represent the combination of the methods *setX* and *setY* with the call *Notify()*. The third edge marked by (4) denotes the change[4] of the superclass of class *Point* from the root class *Object* to the class *Subject* of the Observer pattern.

[4] Looking at the AST description of the woven classes in HyperJ, we realized that the bracket relationship also changes the superclass of the class that contains the bracketed methods to the class of the 'bracketer' method if the two classes have not been equated previously. The weaver has to enforce this inheritance so that the method *Notify* can be accessed from the class *Point*.

For a given language specification, there is a closed set of types of edges and nodes. For example, in case of Java we define a fix set of edges and nodes, which represent the conventional object-oriented relationships. In case of AspectJ or HyperJ, we define nodes and edges, which represent the modules and composition constructs of these languages.

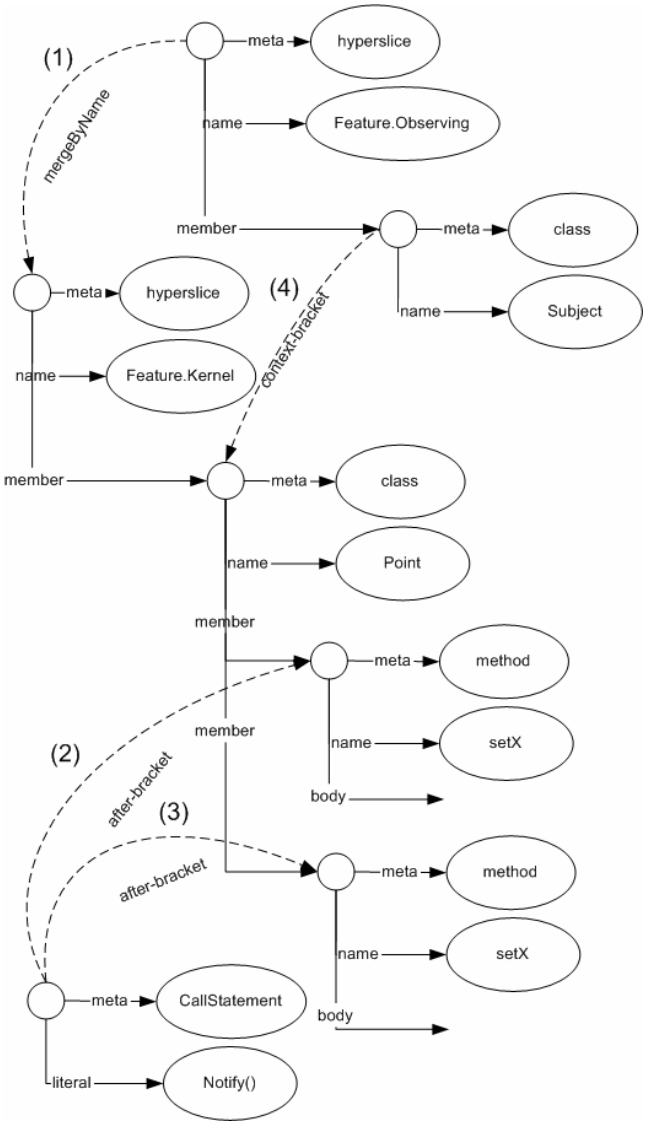


Figure 8. CG of the HyperJ program after the third step

Although languages may require specific kinds of nodes and edges, they are all expressed using the same CG notation. This is the key property in evaluating and comparing different AOLs.

We would like to uniformly interpret the CGs representing programs expressed in different languages. For this purpose, we transform the CGs that represent the aspect-oriented programs, to the CGs that represent the object-oriented implementations of these programs. We therefore transform every AOL specific edge and node to the equivalent object-oriented edge and node.

Figure 9 illustrates after the transformation a part of the CG that represented the introduction statement at (3) in figure 7. As a result of the transformation, a new edge named *superclass* has

been created between the class Point and Subject, while the edge *parent-extends* and the original superclass edge have been deleted.

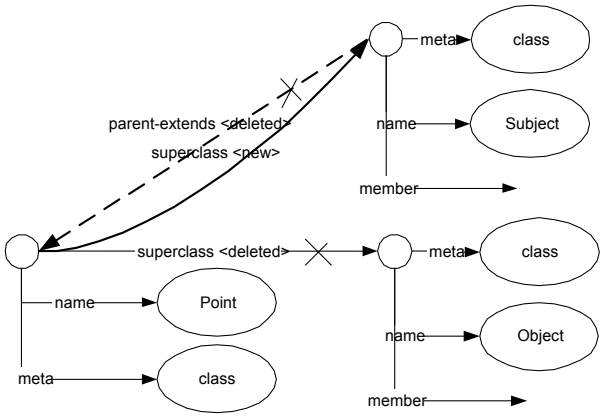


Figure 9. Transformation of the introduction statement

The result graph of the transformation of the *after* advice, illustrated at (1) and (2) in figure 7, is shown in figure 10. Only a new call statement has been attached to the body of the methods *setX* and *setY*. However, this method has no return value and we had to handle only one exit point inside the implementation of the methods. If an *after* advice is combined with an *execution* pointcut designator and the designated method has several return statements then we have to see after another solution that handles each exit point.

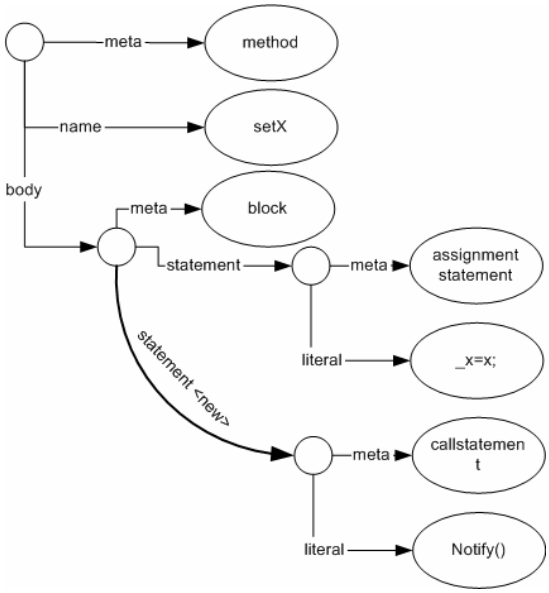


Figure 10. Method *setX* after the transformation

Note that the result graph of the transformation itself does not provide too much information for us. However, if we contrast the source graph of the transformation with the result graph in respect to the related edges and nodes we can see how the composition mechanisms of different languages differ from each other. For example, we can recognize that only one composition structure of an aspect-oriented language is able to implement a complex composition problem, which results in at least three or more standard object-oriented relationships, while another aspect-oriented language needs at least two or more composition structure in order to achieve the same realization.

We propose a transformation language to formulate the transformation processes that practically correspond to the weaving operations.

6. TRANSFORMATION LANGUAGE

In this section we outline a transformation language by which we can describe how the result graphs can be obtained from the source graphs.

6.1 Selecting Graph Fragments

To transform a set of edges and nodes of a graph into another set of edges and nodes first we have to be able to designate certain nodes and edges in the graph that serve as an input of the transformation. We experienced that aspect-oriented language abstractions are typically represented by multiple nodes and edges in Composition Graphs. Therefore, we initiate a query-based technique to select multiple nodes from CGs based on their relationships.

The queries employ formulas of predicate logic with free variables. We used set notation to highlight the free variables. The general form of a query expression, similarly to the tuple relational calculus, is

$$\{t \mid P(t)\}$$

where t is a free variable and P is a predicate. The variables can be quantified: ∃(there exist), ∀(for all). In our model predicates are parameterized propositions that formulate statements whether an edge between a node and an attribute (or between two nodes) exists or not in the CG. The skeletons of the propositions look like these: *node.edge=value* and *node.edge→node*. Predicates can be composed of other predicates by using logical connectives. The result of the query is a set of references to the nodes that satisfies the predicate if they are substituted with the free variables.

As a simple example, let us see the following query expression:

$$\{ X \mid X.meta = class \text{ AND } X.name = Point \}$$

This query will select each node that has a *meta* edge referring to the attribute *class* and a *name* edge referring to the attribute *Point* In other words, the result of this query is a set of references to such nodes that denote classes with the name *Point* (e.g. two classes with the same name can be placed in different packages or hyperslices).

A more complex example is the following:

$$\{Y \mid Y.name=setx \text{ AND } \exists X \exists Z(X.member \rightarrow Y \text{ AND } X.superclass \rightarrow Z \text{ AND } Z.name = Subject)\}$$

This query will designate each method with the name *setX* placed in a class that inherits from the class *Subject*.

By default, the query is executed against the whole graph. There are situations, however, where the scope of the query should be narrowed to only one or more subgraphs of the complete graph. For this purpose, we use scoping expressions that determine a set of subgraphs in order to narrow the scope of the query.

The general form of a scoping expression is

$<N_1, E_1>$ [on $<N_2, E_2>$ on … on $<N_n, E_n>$]

where N is a query expression and E is a set of labels of edges from the original graph. Nodes selected by N denote the root nodes of the subgraphs, while labels in E indicate those edges only which are allowed to connect the nodes in the subgraphs. Scoping expressions can be defined recursively on other scoping expressions.

As a simple example, let us see the following scoping expression:

<{ N | N.meta = method AND N.name = foo }, {statement}>

In this example the node that corresponds to the method *foo* will be the root node of the subgraph and the nodes in this subgraph can be connected through only one type of edge that has the label *statement*.

An application of this scoping expression is shown by the following example:

{RS | RS.meta = return-statement} on

<{ N | N.meta = method AND N.name = foo }, {statement}>

In this example a query expression is combined with the previous scoping expression that selects every return statement from each method called *foo* in the whole CG.

Based on the structure of CGs not only different types of program units but also program statements, such as calls, field reading/writing, etc. can be designated in an elegant manner.

6.2 Transformation Rules

The general form of a transformation rule is

{Identifying pattern}

{Context pattern} > Transformation Statement

where the identifying pattern and context pattern are query expressions, and the transformation statement is the application of a modification type on the nodes selected by the identifying and context pattern. Typical modification types are adding a node or edge to a graph, removing a node or edge from a graph, changing an edge to another one, etc. The identifying pattern identifies those edges that should be eliminated from the CG by the transformation. Sometimes, in the context of the identifying pattern, additional nodes and edges have to be used as input of the transformation. The context pattern designates these ones. The identifying pattern therefore can be regarded as a part of the context pattern.

The following example shows a simple transformation rule:

{X,Y | X.parent-extends→Y} *(1)*

{} > Change(Y.superclass→X) *(2)*

The query expression (1) designates a set of pairs of nodes which are connected via a *parent-extends* edge with each other. The transformation statement (2) changes the edge *parent-extends* between each pair of nodes to the edge *superclass*. Figure 9 illustrates the application of this transformation rule. We did not have to select additional nodes and edges for the transformation, thus, the place of the context pattern left empty.

The transformation rule which is intended to eliminate the *after* edges in figure 7, at (1) and (2) looks like this (the woven methods have only one exit point, no return value):

(1) *{X,Y | X.after→Y}*

(2) *{ MB, S | Y.body→MB AND ∃A∃B(X.member→A AND A.meta=advice AND A.body→B AND B.statement→S)}*

(3) > AppendAfter(MB.statement→S)

The identifying pattern (1) selects pairs of nodes connected through the edge *after*. The context pattern (2) selects the node that denotes the body of the method *(MB)*, and the nodes that denote the statements in the advice *(S)*. The transformation statement (3) appends these latter nodes to the former one.

Naturally, there may be nodes and edges that cannot be directly transformed into the desired form of graph in only one step. In this case a sequence of transformation rules has to be applied in order to achieve the CG with the proper characteristics. For example, merging two hyperslices typically requires the application of more than one transformation rule. On the top level the merge relationship is denoted by only one edge between the two hyperslices. In the first transformation step this edge is processed and a new merge edge is created between each pair of nodes that denote the units of these hyperslices. If some of these units are classes than the merge edge between those classes has to be processed again; in this way, the merge relationships are pushed down to the level of methods of those classes. This process ends up with the merging of methods.

This latter process is known as derivation sequence in the terminology of graph transformation systems. We actually found that this graph transformation language falls into the category of algebraic graph transformation approaches [5].

7. RELATED WORK

In [6], the authors propose a framework by which the core semantics of five aspect-oriented tools, namely AspectJ, DemeterJ, HyperJ, Open Classes, QJBrowser, can be modeled in terms of nine properties. These properties cover, among others, the language the input programs are written in, how the input languages indentify join points and how the input languages contribute to the semantics at the join points. The authors also provide a definition for the term *crosscutting* in terms of the model. However, they had difficulties to achieve a common weaver structure for all five models. Without a common representation the evaluation of AOP languages is difficult. In our approach we will try to provide a more generic model that can help to understand the composition mechanism of these languages.

Assman in [7] presents a GRS-based (Graph Rewrite System) aspect-oriented programming approach, in which aspects, joinpoints and weaving have well-defined and precise semantics in terms of graph-rewriting. In GRS-based aspect-oriented programming aspect composition operators correspond to graph rewrite rules, weavings are direct derivations, and weaved programs are normal forms of the rewrite systems. In accordance with this approach we use a common graph transformation system to model the different types of composition mechanisms of the

existing aspect-oriented languages in a uniform way. However, in our work we focus on the evaluation of the aspect-oriented languages and we regard the graph notation only as a means that helps to reason on the composition mechanisms.

QJBrowser [8] is a code exploration tool by which various program elements can be extracted from a source model and presented in a hierarchical view. A selection criterion determines what elements should be extracted from the program. This criterion is defined as a query in terms of first order predicates. The query is executed against the source model and results in the tuples of the selected properties. In our approach we use a similar technique to select certain nodes of the CGs.

Mens in [9] presents conditional graph rewriting as a domain-independent, formal approach for managing unanticipated software evolution. He proposes labeled typed nested graphs to represent complex software artifacts and graph rewriting to control the evolution of these artifacts. Similarly, we would like to use CGs as a domain independent formalism to model different program units and graph transformation as a formalism to describe weaving operations.

8. CONCLUSION & FUTURE WORK

In this paper, we have introduced the concept of Composition Graphs as a means for reasoning about (aspect-oriented) composition. We have illustrated how CGs can be used to represent a simple example program, expressed in Java, AspectJ and HyperJ, respectively. Subsequently, we demonstrated how composition (or weaving) mechanisms can be represented by transformation rules upon CGs.

This paper aims at laying the foundation for further work in reasoning about composition mechanisms.

- We may use CGs to reason about properties such as degrees of coupling and cohesion, e.g. by defining metrics.
- We believe that we can define the semantics of composition mechanisms effectively by specifying general transformation rules, which can transform any CG in language A towards a CG in language B. Hence, the essential differences in composition mechanisms can be observed by looking at the differences between the transformation rules.
- We expect that the application of CGs to represent a variety of programs in different AOLs will yield increased insight in the workings and essence of aspect-oriented approaches, perhaps leading to new or generalized composition mechanisms.

Although we have already gained some experience in modeling programs in different AOP languages as CGs, there are still several issues left to be addressed as future work. First of all, we have to refine the structure of the graphs in case of each language. In other words, we want to enrich the set of types of nodes and edges that represent AO composition structures. Besides, the transformation rules also have to be specified in order to reason about the corresponding composition mechanism.

Further issues that we plan to address shortly:

- Improving the representation/visualization of the CGs
- Address the ability to model both static composition and runtime composition.
- Define metrics to judge certain characteristics and quality attributes of programs represented as CGs.
- Analysis and comparison of existing composition mechanisms and identification of new composition mechanisms

We intend to explore the application of transformation rules to create code generators.

9. REFERENCES

[1] Kiczales, G. et al., *An overview of AspectJ*, in *Proceeding ECOOP 2001, LNCS 2072*, J.L. Knudsen, Editor. 2001, Springer-Verlag: Berlin. pp. 327-353.

[2] Ossher, H. and P. Tarr, *Hyper/J: Multi-dimensional separation of concerns for Java*, in *Proceeding 23rd International Conference on Software Engineering.* 2001, IEEE Computer Society. Pp. 729-730.

[3] Gamma, E. et al., *Design Patterns: elements of reusable object-oriented software*. 1995, Addison-Wesley.

[4] Hannemann, J. and G. Kiczales, *Design pattern implementation in Java and AspectJ*, in *Proceeding OOPSLA '02*. 2002, ACM SIGPLAN Notices.

[5] Rozenberg, G. (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 1., 1997, World Scientific.

[6] Masuhara, H. and G. Kiczales, *A Modeling Framework for Aspect-Oriented Mechanism*, in *Proceeding ECOOP '03*. 2003.

[7] Assman, U. and A. Ludwig, *Aspect Weaving by Graph Rewriting*, 1999, Generative Component-Based Software Engineering (GCSE), p. 24-36.

[8] Rajagolopan, R. and K.D. Volder, *QJBrowser: A Query-Based Approach to Explore Crosscuting Concerns.* 2002, submitted for publication.

[9] Mens, T., *Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution.* 2000, Lecture Notes in Computer Science, Springer-Verlag.

A Formal Model for Cross-cutting Modular Transition Systems

Henny B. Sipma*
Computer Science Department
Stanford University
Stanford, CA. 94305-9045
sipma@cs.stanford.edu

ABSTRACT

We define a notion of aspects in the framework of modular transition systems. In our model an aspect is viewed as a semantic transformation on transition systems. Our primary objective is to use the model as a basis for studying inheritance and imposition properties of aspect constructs currently in use in practical languages such as AspectJ. We show that our model is sufficiently expressive to represent many of the constructs in this language. However, the mechanism of aspect-orientation presented in this paper may also be of practical use for systems organized in a modular fashion.

1. INTRODUCTION

In recent years cross-cutting techniques have emerged as a useful programming technique orthogonal to object-oriented and modular programming methods [7, 6]. In this paper we propose a formal model of such cross-cutting techniques, also called aspects, in the framework of modular transition systems, an expressive, first-order representation of reactive systems. Our aim is to use the model as a basis to study properties and capabilities of such techniques, including inheritance properties: which system properties are preserved across the application of aspects, and imposition properties: which systems are guaranteed to satisfy the property imposed by the aspect. Although not set in an object-oriented framework we expect our analysis results to give insight into the general application of aspect-oriented techniques.

In our model aspects are viewed as semantic transformations on modular systems. Aspects can introduce global and local state into modules, introduce additional statements, and modify existing statements, including altering the program flow. The system modifications are modeled by a combination of abstraction to add newly desired program behaviors, and restriction to remove unwanted program behaviors. Analysis of the effect of aspect application can thus make use of the well-known analysis methods for abstraction and restriction.

Although our main objective is to use the model as a basis for analysis, the application of aspects to modular transition systems presented here can also be useful in practical construction of reactive programs. Because of the semantic basis of our method, potentially more constructs are enabled than with current methods in which new code can be introduced only at so-called join points in the call graph of the program.

The paper is organized as follows. In the next section we introduce the computational model of transition systems and present a simple programming language to describe systems. In section 3 we define the representation and semantics of an aspect, and in section 4 we illustrate some typical aspect constructs commonly provided by aspect-oriented languages. In section 5 a preliminary outline is given of the types of analysis we expect to do based on the model presented here, and section 6 concludes with a discussion of some shortcomings of our model and plans for future work.

2. PRELIMINARIES

2.1 Computational Model: Transition Systems

Our basic computational model is that of a *transition system* [9] (TS), $\mathcal{S} = \langle V, \Theta_{\mathcal{S}}, \mathcal{T} \rangle$, where $V \subset \mathcal{V}$ is a finite set of typed variables taken from a universal set of variables $\mathcal{V}$, $\Theta_{\mathcal{S}}$ is an assertion (first-order formula) characterizing the initial states, and $\mathcal{T}$ is a finite set of transitions. A *state* s is an interpretation of $\mathcal{V}$, which assigns to each variable $v \in \mathcal{V}$ a value $s[v]$ over its domain; Σ denotes the set of all states. A transition $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \mapsto 2^{\Sigma}$, and each state in $\tau(s)$ is called a τ-successor of s. We say that a transition τ is *enabled* on s if $\tau(s) \neq \emptyset$, otherwise τ is *disabled* on s.[1]

*This research was supported in part by NSF grants CCR-99-00984-001 and CCR-0121403, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.

[1]Note that the set of transitions can equally well be represented by a single transition. In the original definition of [9] separate transitions are identified to support the definition of fairness. Although we do not handle fairness in this paper, it is the intention to include it in the future. In addition, identification of separate transitions allows finer-grain control of aspect applicability.

Each transition $\tau \in \mathcal{T}$ can be described by a first-order formula $\rho_\tau(V, V')$, called the *transition relation*, expressing the relation between a state s and any of its τ-successors $s' \in \tau(s)$. In $\rho_\tau(V, V')$ the unprimed versions of the variables refer to values in s and the primed versions refer to values in s'. For example, the formula $x' = x + 1$ represents the transition function in which the set of τ-successors of a state s with $s[x] = c$ contains all states s' such that $s'[x] = c + 1$ for some constant c.

A *run* $\sigma : s_0, s_1, \ldots$ of a transition system $\mathcal{S}$ is an infinite sequence of states such that the following two conditions hold

- **Initiation:** the first state is initial, that is $s_0 \models \Theta_\mathcal{S}$;
- **Consecution:** for each $i \geq 0$, the state s_{i+1} is a τ-successor of s_i for some $\tau \in \mathcal{T}$.

The behavior of a system $\mathcal{S}$ is identified with its set of runs, denoted by $\mathcal{L}(\mathcal{S})$.

2.2 Modular transition systems

Modular transition systems organize a transition system into transition modules that can be composed into larger modules by means of module expressions [3]. In this paper we do not use the full power of module expressions, but restrict ourselves to modular transition systems consisting of n basic modules composed in parallel, where a basic module consists of an *interface* $I : \langle V_{input}, V_{output}, V_{shared} \rangle$ declaring the input variables V_{input}, which can be modified by other modules, but not by the module itself, the output variables V_{output}, whose value can be observed by the other modules, but can only be modified by the module itself, and the shared variables V_{shared}, which can be observed and modified by all modules, and a *body* $B : \langle V_M, \Theta_M, \mathcal{T}_M \rangle$ containing the set of variables local to the module, an initial condition, and a set of transitions. For the precise semantics of the parallel composition operator we refer to [3], as it is not directly relevant to the remainder of this paper.

2.3 SPL programs

Although systems can be described directly as modular transition systems, using first-order formulas to describe the initial condition and transition relations, it is usually more convenient to represent a system as a program in some structured programming language with a well-defined semantics in terms of transition systems. We will use SPL (Simple Programming Language) a simple imperative programming language to describe modular transition systems [9].

Example Figure 1 shows program BAKERY, a program that implements Lamport's *Bakery algorithm* for mutual exclusion [8], consisting of two SPL modules, P_1 and P_2. This program can be translated into a modular transition system with modules M_1 and M_2. The two modules have interfaces

$$\begin{array}{rcl} I_1 & = & \langle \{y_2\}, \{y_1\}, \emptyset \rangle \\ I_2 & = & \langle \{y_1\}, \{y_2\}, \emptyset \rangle \end{array}$$

respectively, reflecting that y_2 is observed by P_1, but not modified by P_1, and y_1 is modified by P_1, but not modified by the other module, and the other way around for module P_2. The bodies of the two modules are given by

$$\begin{array}{rcl} B_1 & = & \langle \{\pi_1\}, y_1 = 0 \wedge \pi_1 = \ell_0, \mathcal{T}_1 \rangle \\ B_2 & = & \langle \{\pi_2\}, y_2 = 0 \wedge \pi_2 = m_0, \mathcal{T}_2 \rangle \end{array}$$

where $\mathcal{T}_1$ and $\mathcal{T}_2$ contain the transitions corresponding to the statements in each module, as explained below.

Each statement in an SPL module is associated with a *prelocation*, identified by the label of the statement, and a set of *post locations*. To each module M_i a control variable π_i is added, ranging over the set of locations in the module, and initialized to the prelocation of the first statement in the module. Each statement corresponds with a transition according to the transition semantics of each statement. For example, the statement labeled by ℓ_2 is represented by a transition with transition relation

$$\pi_1 = \ell_2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge \pi_1' = \ell_3 \wedge y_1' = y_1$$

while the statement labeled by ℓ_0 can be represented by

$$\left(\begin{array}{c} \pi_1 = \ell_0 \\ \wedge \\ ((\mathit{true} \wedge \pi_1' = \ell_1) \vee (\mathit{false} \wedge \pi_1' = \ell_6)) \\ \wedge \\ (y_1' = y_1 \wedge y_2' = y_2) \end{array} \right)$$

In the remainder of the paper we will usually omit the conjuncts like $y_1' = y_1$, stating that a variable is preserved, and assume that all system variables not mentioned in the transition relation are preserved.

Thus, $\mathcal{T}_1$ and $\mathcal{T}_2$ each contain six transitions, one for each statement.

For a detailed description of the semantics of SPLin terms of transition systems, the reader is referred to [9].

$$P_1 : \left[\begin{array}{l} \textbf{out } y_1 : \textbf{integer where } y_1 = 0 \\ \textbf{in } y_2 : \textbf{integer} \\ \\ \ell_0 : \textbf{ while } \mathrm{T} \textbf{ do} \\ \quad \left[\begin{array}{l} \ell_1 : \textbf{ noncritical} \\ \ell_2 : \ y_1 := y_2 + 1 \\ \ell_3 : \textbf{ await } (y_2 = 0 \vee y_1 \leq y_2) \\ \ell_4 : \textbf{ critical} \\ \ell_5 : \ y_1 := 0 \end{array} \right] \\ \ell_6 : \end{array} \right]$$

$$\Big\| \qquad P_2 : \left[\begin{array}{l} \textbf{out } y_2 : \textbf{integer where } y_2 = 0 \\ \textbf{in } y_1 : \textbf{integer} \\ \\ m_0 : \textbf{ while } \mathrm{T} \textbf{ do} \\ \quad \left[\begin{array}{l} m_1 : \textbf{ noncritical} \\ m_2 : \ y_2 := y_1 + 1 \\ m_3 : \textbf{ await } (y_1 = 0 \vee y_2 < y_1) \\ m_4 : \textbf{ critical} \\ m_5 : \ y_2 := 0 \end{array} \right] \\ m_6 : \end{array} \right]$$

Figure 1: Program BAKERY

In the examples in Section 4 we will use the functions *preloc* and *postloc* that assign to each statement its prelocation and a set of post locations, respectively, where the set of post locations is determined by the control flow of the statement, as defined by the transition semantics. For example, $preloc(\ell_1 : y_1 := y_2 + 1) = \ell_1$, and $postloc(\ell_0 : y_1 := 0) = \{\ell_1, \ell_6\}$, even though location ℓ_6 is not reachable via statement ℓ_0. Again these functions are fully defined in [9].

3. ASPECTS

An aspect is defined as a transformation that maps modular transition systems into modular transition systems. The modifications an aspect may make to a modular transition system include the addition of global and local state to the system and modules, respectively, the introduction of new transitions, and the modification of existing transitions[2]. An aspect is described by a set of global variables, possibly with an initial condition, and a list of aspect facets, one for each module constituting the modular system, specifying the additional state and modifications to each module.

Modification of transition relations is achieved by a combination of conditional abstraction and restriction. Transition relations that imply an abstraction condition are abstracted by projecting out a set of variables associated with the condition. Abstraction is followed by restriction: the transition relations of those transitions whose original transition relation implies a restriction condition are conjoined with the associated restriction assertion.

The approach of transformation is illustrated in Figure 2. The original set of behaviors of the system is enlarged by means of abstraction. This set is then reduced by restriction, possibly eliminating some or all of the original system behaviors.

Our definition of an aspect was inspired by the class extensions presented in [4], which are code facets that are added to a base system based on predefined entry and exit conditions in the base system.

3.1 Definition

An *aspect* $\mathcal{A} : \langle V_A, \Theta_A, \alpha_1, \ldots, \alpha_n \rangle$ is a transformation defined on modular transition systems $\mathcal{M} : \langle M_1, \ldots, M_n \rangle$ consisting of n modules, with

$$M_i : \langle \langle V_{i,input}, V_{i,output}, V_{i,shared} \rangle, \langle V_i, \Theta_i, \mathcal{T}_i \rangle \rangle$$

The aspect consists of the following components:

- V_A: a finite set of global aspect variables, disjoint from the variables in $\mathcal{M}$;
- Θ: the aspect initial condition, an assertion over V_A and the shared variables in $\mathcal{M}$;
- $\alpha_1, \ldots, \alpha_n$: a list of aspect facets.

[2]At this stage we do not allow aspects to modify the modular structure; it is a straightforward extension to allow aspects to introduce new modules.

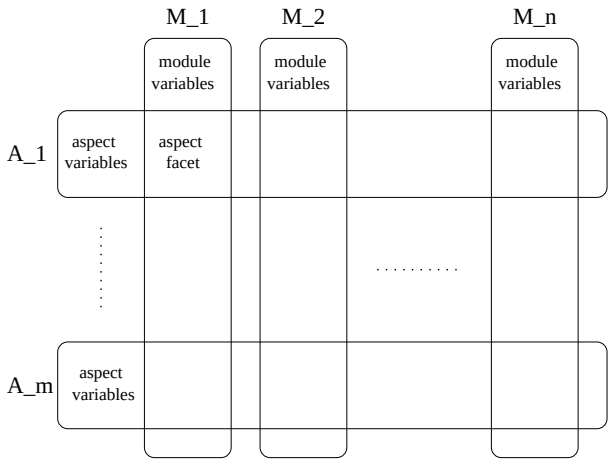


Figure 3: Schematic representation of aspect application

An aspect facet is defined for each module. It describes how the behavior of the module is modified. Figure 3 gives a schematic representation of the relationship between modules and aspects and the position of aspect facets.

An aspect facet $\alpha_i : \langle V_{\alpha i}, \Theta_{\alpha i}, \mathcal{T}_{\alpha i}, \beta_i, \epsilon_i \rangle$ consists of the following components:

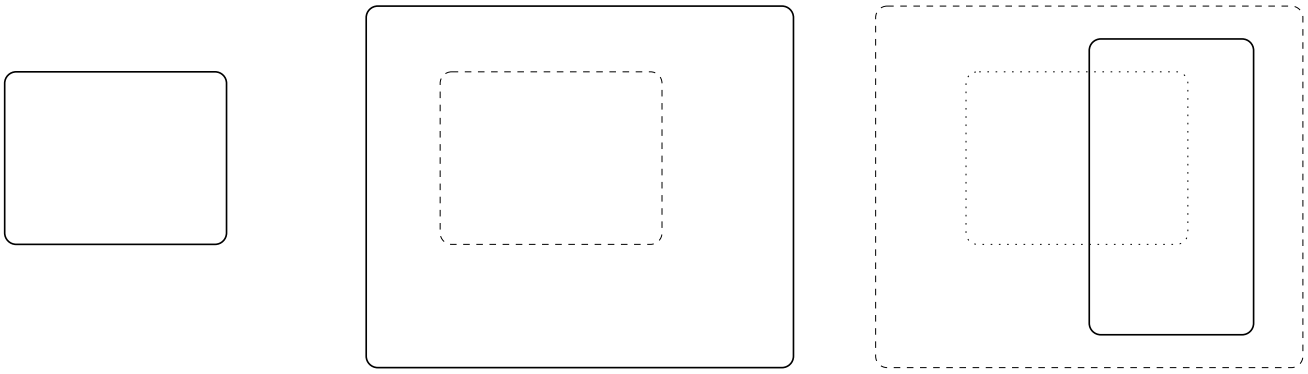
- $V_{\alpha i}$: a set of variables local to the facet, disjoint from the variables in $\mathcal{M}$; these variables are used to perform local bookkeeping and are not visible by other aspect facets.
- $\Theta_{\alpha i}$: the aspect facet initial condition, an assertion over $V_{\alpha i} \cup V_i \cup V_A$, used to initialize the local facet variables.
- $\mathcal{T}_{\alpha i}$: a finite set of transitions that may modify variables in $V_A \cup V_{\alpha i} \cup V_{i,output} \cup V_{i,shared} \cup V_i$, and in addition, may depend on the values of variables in $V_{i,input}$.
- β_i: a finite set of aspect facet abstraction instructions, where each instruction $\beta_{ij} = (\varphi_{ij}, W_{ij})$ is a pair consisting of an assertion φ_{ij} that governs the applicability of the instruction, and a set of variables W_{ij} to be projected out of the transition in case the transition relation implies the applicability condition;
- ϵ_i: a finite set of aspect facet restriction instructions, where each instruction $\epsilon_{ij} = (\psi_{ij}, \chi_{ij})$ is a pair consisting of an assertion ψ_{ij} governing the applicability of the instruction, and an assertion χ_{ij} to be conjoined with the transition relation in case the transition relation implies the applicability condition.

3.2 Semantics

Given an aspect $\mathcal{A} : \langle V_A, \Theta_A, \alpha_1, \ldots, \alpha_n \rangle$ and a modular transition system $\mathcal{M} : \langle M_1, \ldots, M_n \rangle$ the application of $\mathcal{A}$ to $\mathcal{M}$, written $\mathcal{A}(\mathcal{M})$, defines the modular transition system $\mathcal{M}^* : \langle M_1^*, \ldots, M_n^* \rangle$ with

$$M_i^* : \langle \langle V_{i,input}^*, V_{i,output}^*, V_{i,shared}^* \rangle, \langle V_i^*, \Theta_i^*, \mathcal{T}_i^* \rangle \rangle$$

with the following values



a) original set of behaviors b) set of behaviors after abstraction c) set of behaviors after restriction

Figure 2: An aspect as a combination of abstraction and restriction

- The input and output variables of all modules remain unchanged:

$$V^*_{i,input} = V_{i,input} \quad \text{and} \quad V^*_{i,output} = V_{i,output}$$

- The global aspect variables are included in the shared variables of each module, making these variables visible to all modules:

$$V^*_{i,shared} = V_{i,shared} \cup V_A$$

- The local variables of each aspect facet are included in the local variables of the corresponding module:

$$V^*_i = V_i \cup V_{\alpha i}$$

- The global aspect initial condition and the aspect facet initial condition are both conjoined with the module initial condition:

$$\Theta^*_i = \Theta_i \wedge \Theta_A \wedge \Theta_{\alpha i}$$

- The new set of transitions $\mathcal{T}^*_i$ consists of the module transitions, possibly modified by the facet abstraction and restriction instructions, and the facet transitions.

  Let $\tau \in \mathcal{T}_i$ be a module transition with transition relation ρ_τ, and let $\beta = \{(\phi_1, W_1), \ldots (\phi_k, W_k)\}$ be the set of abstraction instructions of aspect facet α_i. Then the abstracted transition τ_β has transition relation

$$\exists X \,.\, \rho_\tau$$

  where

$$X = \bigcup_{j \in I} W_j$$

  where I is the index set containing the indices of the abstraction instructions whose condition is implied by the transition relation, that is,

$$I = \{j \mid 1 \le j \le k \text{ and } \rho_\tau \rightarrow \phi_j\}$$

  Thus the variables associated with those abstraction instructions whose condition is implied by the transition relation are projected out of the transition relation.

  Similarly, let $\epsilon = \{(\psi_1, \chi_1), \ldots, (\psi_k, \chi_k)\}$ be the set of restriction instructions for aspect facet α_i. Then the abstracted and restricted transition $\tau_{\beta\epsilon}$ has transition relation

$$\exists X \,.\, \rho_\tau \wedge \bigwedge_{j \in I} \chi_j$$

  where

$$I = \{j \mid 1 \le j \le k \text{ and } \rho_\tau \rightarrow \psi_j\}$$

  is the index set containing the indices of the restriction conditions implied by the transition relation.

  For example, consider a transition τ with transition relation

$$\rho_\tau : \ \pi = \ell_1 \wedge x > 0 \wedge \pi' = \ell_2$$

  and abstraction and restriction instructions

$$\begin{array}{ll} \beta: & \{(\pi = \ell_1, \{\pi'\}), (\pi = \ell_4, \{x\})\} \\ \epsilon: & \{(\pi = \ell_1, \pi' = \ell_5)\} \end{array}$$

  Clearly ρ_τ implies the first abstraction condition in β, but not the second, and thus only π' is projected out, resulting in an abstracted transition τ_β with transition relation

$$\rho_{\tau_\beta} : \ \pi = \ell_1 \wedge x > 0$$

  Subsequent application of the restriction condition then results in the transition $\tau_{\beta\epsilon}$ with transition relation

$$\rho_{\tau_{\beta\epsilon}} : \ \pi = \ell_1 \wedge x > 0 \wedge \pi' = \ell_5$$

  Thus, the effect of β and ϵ on τ is to redirect the control flow of τ to a new location.

  Using the above notation, the new set of transitions $\mathcal{T}^*_i$ can now be given as

$$\mathcal{T}^*_i = \{\tau_{\beta_i \epsilon_i} \mid \tau \in \mathcal{T}_i\} \ \cup \ \mathcal{T}_{\alpha i}$$

4. ASPECT EXAMPLES

The aspect model introduced in the previous section is sufficiently expressive to represent many of the aspect constructs used in practical languages such as AspectJ [6]. In this section we present some examples of how these constructs can be represented in our model.

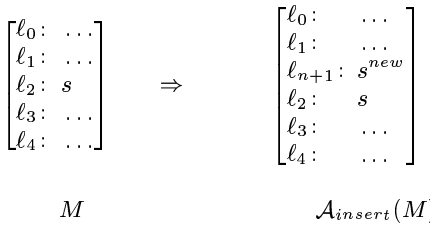


Figure 4: Inserting a statement before a given statement s

Insert Before

The most common aspect action is to introduce one or more statements before (or after) a given statement. Given a sequential program represented by the modular system $\mathcal{M} : \langle M \rangle$ with control variable π with range $\ell_0 \ldots \ell_n$, the following aspect inserts transition τ^{new} with (unspecified) transition relation ρ^{new} just before a given statement s.

$$\mathcal{A}_{insert} = \langle \emptyset, true, \alpha \rangle$$

The set of global variables is empty as no global state needs to be introduced, and there are no restrictions to the global initial condition. There is one aspect facet,

$$\alpha = \langle \emptyset, true, \{\tau^{new}\}, \beta, \epsilon \rangle$$

The set of local variables is empty and there is no change to the initial condition.[3] The facet transitions include the transition to be introduced, τ^{new}. The abstraction condition β needs to abstract the transition relation of the statement before s to enable redirection of control to the new transition:

$$\beta = \left\{ (\pi' = preloc(s), \{\pi'\}) \right\}$$

that is, variable π' is projected out from transition relations in which the control variable is set to the prelocation of statement s. Finally ϵ restricts the same transitions to direct control to a new program location ℓ_{n+1}, not occurring in the program

$$\epsilon = \left\{ (\pi' = preloc(s), \pi' = \ell_{n+1}) \right\}$$

The transition relation of the new transition now becomes

$$\rho_{\tau^{new}} : \rho^{new} \wedge \pi = \ell_{n+1} \wedge \pi' = preloc(s)$$

that is, the new transition is associated with the new program location and its post location is the prelocation of the given statement s.

Figure 4 shows the effect of the aspect in SPL program notation. Note that the above aspect is only one way of representing this construct. Others are possible and may be convenient in different situations (for example, if one does not want to modify the range of the control variable).

Logging

Given a system $\mathcal{M} : \langle M_1, M_2 \rangle$, the following aspect counts the changes to system variable x:

$$\mathcal{A}_{logging} = \langle \{N_x\}, N_x = 0, \alpha_1, \alpha_2 \rangle$$

[3] For simplicity we assume that s is not the first statement of M; if it is we do need to modify the initial condition.

It introduces a global aspect variable N_x, initialized to 0, to record the number of changes and includes two aspect facets $\alpha_{1,2}$:

$$\alpha_{1,2} : \{\emptyset, true, \emptyset, \emptyset, \epsilon\}$$

whose only significant component is the restriction instruction

$$\epsilon = \left\{ (true, (x' \neq x \rightarrow N'_x = N_x + 1) \wedge (x' = x \rightarrow N'_x = N_x)) \right\}$$

whose effect is to add the above conjunct to all transitions in $M_{1,2}$, incrementing N_x when x is modified, and leaving it unchanged otherwise.

An alternative restriction instruction is

$$\epsilon = \left\{ (x' \neq x, N'_x = N_x + 1), (x' = x, N'_x = N_x) \right\}$$

which, when used in actual program construction, would lead to a more efficient program. However, it relies on our ability to decide whether a transition relation implies the restriction conditions, which, in general, may not be decidable.

Adding synchronization

Consider a system $\mathcal{M} : \langle M_1, M_2 \rangle$ with two parallel processes with control variables $\pi_{1,2}$ ranging over program locations $\ell_0 \ldots \ell_n$ and $m_0 \ldots m_k$ respectively, that uses a resource S that requires mutually exclusive access. The following aspect adds protection of the resource using the *Bakery* synchronization mechanism shown in Section 2.

$$\mathcal{A}_{exclusion} = \langle \{y_1, y_2\}, y_1 = 0 \wedge y_2 = 0, \alpha_1, \alpha_2 \rangle$$

Two global variables are introduced, both initialized to 0. The two aspect facets are defined as follows:

$$\alpha_1 : \langle \emptyset, true, \{\tau_{11}, \tau_{12}, \tau_{13}\}, \beta_1, \epsilon_1 \rangle$$

with

$$\begin{array}{ll} \rho_{\tau_{11}} : & \pi_1 = \ell_{n+1} \wedge \pi'_1 = \ell_{n+2} \wedge y'_1 = y_2 + 1 \\ \rho_{\tau_{12}} : & \pi_1 = \ell_{n+2} \wedge \pi'_1 = preloc(S_1) \wedge (y_2 = 0 \vee y_1 \leq y_2) \\ \rho_{\tau_{13}} : & \pi_1 = \ell_{n+3} \wedge \pi'_1 \in postloc(S_1) \wedge y'_1 = 0 \end{array}$$

where $\ell_{n+1} \ldots \ell_{n+3}$ are new program locations not occurring in the program. The abstraction instruction is given by

$$\beta_1 = \left\{ (\pi_1 = preloc(S_1) \vee \pi'_1 = preloc(S_1), \{\pi'_1\}) \right\}$$

eliminating the control flow from the statement using the resource, and those preceding the statement that uses the resource. The restriction instruction restores the control flow:

$$\epsilon_1 = \left\{ \begin{array}{l} (\pi'_1 = preloc(S_1), \pi'_1 = \ell_{n+1}), \\ (\pi_1 = preloc(S_1) \wedge \pi'_1 \in postloc(S_1), \pi'_1 = \ell_{n+3}) \end{array} \right\}$$

Similarly,

$$\alpha_2 = \langle \emptyset, true, \{\tau_{21}, \tau_{22}, \tau_{23}\}, \beta_2, \epsilon_2 \rangle$$

with

$$\begin{array}{ll} \rho_{\tau_{21}} : & \pi_2 = m_{k+1} \wedge \pi'_2 = m_{k+2} \wedge y'_2 = y_1 + 1 \\ \rho_{\tau_{22}} : & \pi_2 = m_{k+2} \wedge \pi'_2 = preloc(S_2) \wedge (y_1 = 0 \vee y_2 < y_1) \\ \rho_{\tau_{23}} : & \pi_2 = m_{k+3} \wedge \pi'_2 \in postloc(S_2) \wedge y'_2 = 0 \end{array}$$

and similar definitions for β_2 and ϵ_2 as above.

Figure 5 shows the effect of the aspect in SPL program notation.

$$\left[\begin{array}{l}\ell_0\colon\ \ldots\\ \ell_1\colon\ S_1(uses\ S)\\ \ell_2\colon\ \ldots\end{array}\right] \ \|\ \left[\begin{array}{l}m_0\colon\ \ldots\\ m_1\colon\ S_2(uses\ S)\\ m_2\colon\ \ldots\end{array}\right]$$

(a) $\mathcal{M} :< M_1, M_2 >$ using shared resource S

$$\left[\begin{array}{l}\textbf{shared } y_1, y_2 : \textbf{integer}\\ \ \textbf{where } y_1 = y_2 = 0\\ \ell_0\colon\ \ldots\\ \ell_{n+1}\colon\ y_1 := y_2 + 1\\ \ell_{n+2}\colon\ \textbf{await } (y_2 = 0 \vee y_1 \leq y_2)\\ \ell_1\colon\ S_1(uses\ S)\\ \ell_{n+3}\colon\ y_1 := 0\\ \ell_2\colon\ \ldots\end{array}\right] \ \|\ \left[\begin{array}{l}\textbf{shared } y_1, y_2 : \textbf{integer}\\ \ \textbf{where } y_1 = y_2 = 0\\ m_0\colon\ \ldots\\ m_{k+1}\colon\ y_2 := y_1 + 1\\ m_{k+2}\colon\ \textbf{await } (y_1 = 0 \vee y_2 < y_1)\\ m_1\colon\ S_2(uses\ S)\\ m_{k+3}\colon\ y_2 := 0\\ m_2\colon\ \ldots\end{array}\right]$$

(b) $\mathcal{A}_{exclusion}(\mathcal{M})$ ensuring mutually exclusive access to S

Figure 5: Adding synchronization

$$\left[\begin{array}{l}\ell_0\colon\ i := 2\\ \ell_1\colon\ \textbf{while } i < N \textbf{ do}\\ \quad\left[\begin{array}{l}\ell_2\colon\ a[i] := f(a[i-1], a[i], a[i+1])\\ \ell_3\colon\ i := i + 1\end{array}\right]\\ \ell_4\colon\ i := 1\\ \ell_5\colon\ \textbf{while } i < N - 1 \textbf{ do}\\ \quad\left[\begin{array}{l}\ell_6\colon\ a[i] := g(a[i])\\ \ell_7\colon\ i := i + 1\end{array}\right]\\ \ell_8\colon\end{array}\right]$$

Figure 6: Program SERIES

$$\left[\begin{array}{l}\ell_0\colon\ i := 2\\ \ell_1\colon\ \textbf{while } i < N \textbf{ do}\\ \quad\left[\begin{array}{l}\ell_2\colon\ a[i] := f(a[i-1], a[i], a[i+1])\\ \ell_9\colon i := i - 1\\ \ell_6\colon a[i] := g(a[i])\\ \ell_7\colon i := i + 1\\ \ell_3\colon i := i + 1\end{array}\right]\\ \ell_8\colon\end{array}\right]$$

Figure 7: $\mathcal{A}_{fusion}$(SERIES)

Loop Fusion

Our model of aspects is sufficiently expressive to represent loop fusion, a transformation often useful in image processing to optimize cache performance. Consider program SERIES shown in Figure 6. Assume the program has control variable π with range 0..8, such that $\pi = i$ when control is at location ℓ_i.

The following aspect merges the two loops starting at ℓ_1 and ℓ_5:

$$\mathcal{A}_{fusion} = \langle \emptyset, true, \alpha \rangle$$

with

$$\alpha = \langle \emptyset, true, \{\tau\}, \beta, \epsilon \rangle$$

with abstraction instruction

$$\beta = \left\{(\pi = 1 \vee \pi = 2 \vee \pi = 7, \{\pi'\})\right\}$$

and restriction instruction

$$\epsilon = \left\{ \begin{array}{l} (\pi = 1, (i < N \wedge \pi' = 2) \vee (i \geq N \wedge \pi' = 8)) \\ (\pi = 2, \pi' = 9) \\ (\pi = 7, \pi' = 3) \end{array} \right\}$$

The transition relation of the newly introduced transition is

$$\rho_\tau :\ \pi = 9\ \wedge\ \pi' = 6\ \wedge\ i' = i - 1$$

The result of applying $\mathcal{A}_{fusion}$ to program SERIES is shown in Figure 7.

5. ANALYSIS OF ASPECTS

Properties of reactive systems are often specified in some form of temporal logic. For example, the property that in all program behaviors the value of system variable x does not exceed 10 can be specified by the linear temporal logic (LTL) formula

$$\Box(x \leq 10)$$

where the $\Box$-operator means "always". Other temporal operators are provided that state that some condition p must eventually be fulfilled ($\Diamond p$), or that some condition p holds until another condition q holds ($p \mathcal{U} q$). The precise semantics of LTL can be found in [9].

Verifying that a system $\mathcal{S}$ satisfies a specification expressed as an LTL formula ϕ, usually written

$$\mathcal{S} \models \phi$$

consists of proving that all program behaviors of $\mathcal{S}$ satisfy ϕ, that is

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\phi)$$

where $\mathcal{L}(\phi)$ is the set of all behaviors (infinite sequences of states) that satisfy ϕ.

Property Inheritance

The definition of a formal semantics of aspects now allows us to study questions of the type

$$\frac{\mathcal{M} \models \varphi, \ldots}{\mathcal{A}(\mathcal{M}) \models \varphi}$$

for some temporal property φ; that is, what restrictions on aspects guarantee preservation of system properties. For example, it is easy to see that for a given safety property φ

$$\frac{\mathcal{M} \models \varphi, \beta_{\mathcal{A}} = \emptyset, \mathcal{T}_{\mathcal{A}} = \emptyset}{\mathcal{A}(\mathcal{M}) \models \varphi}$$

where $\beta_{\mathcal{A}} = \emptyset$ stands for $\beta_i = \emptyset$ for all aspect fragments α_i in $\mathcal{A}$, and similarly, $\mathcal{T}_{\mathcal{A}} = \emptyset$ stands for $\mathcal{T}_i = \emptyset$ for all aspect fragments. Clearly all aspects in which no abstraction is applied, that is, no program behaviors are added to the system, and no new transitions are introduced, should preserve any safety property. Note that liveness properties may not be preserved as restriction may disable transitions necessary to achieve some goal.

As was suggested in an early work on superimposition [5], aspects can be classified by their inheritance properties. For example, one can distinguish *monitoring* aspects, which perform pure augmentation and therefore preserve all temporal properties, *regulatory* aspects, such as the one given above, which may turn unfair computations into fair ones, and thus cause liveness properties to be violated, and all other aspects, which cannot make any guarantees. We expect to be able to make a finer classification for this last class.

Property SuperImposition

Similarly, one may develop a notion of aspects satisfying certain properties and determining what restrictions on module systems are required to ensure that the resulting system satisfies the property, that is, under what conditions can we guarantee

$$\frac{\mathcal{A} \models \varphi, \ldots}{\mathcal{A}(\mathcal{M}) \models \varphi}$$

Aspect Interaction

The third question that can be studied in this framework is aspect interaction, that is, under what conditions does the following hold:

$$\mathcal{L}(\mathcal{A}_1(\mathcal{A}_2(\mathcal{M}))) \;=\; \mathcal{L}(\mathcal{A}_2(\mathcal{A}_1(\mathcal{M})))$$

That is, is the order of applying aspects significant?

6. DISCUSSION AND FUTURE WORK

Extensions

The model presented in this paper is the basic model upon which we intend to build several extensions. A first desirable extension is parameterization. In our current aspect description language, variables, predicates and transitions must be specified literally. It is a straightforward extension to the aspect description language to allow aspects to be parameterized by the components appearing in the aspect fragments, such as β, ϵ and $\mathcal{T}$. A more challenging extension is to add the capability of capture of context as, for example, provided in AspectJ.

The current model relies on rather coarse abstraction, potentially causing the set of behaviors to grow larger than necessary, thus making it harder to prove property preservation. We are currently investigating whether a more fine-grained abstraction such as assertion-based abstraction [2] would allow to increase the accuracy of the abstraction in a useful way.

For ease of exposition we have omitted fairness in this paper. To investigate inheritance and superimposition of liveness properties, fairness properties will have to be included.

Constructing systems

The model as presented cannot be used directly to construct new systems from aspects and base systems. Abstraction and restriction depend on our ability to decide the validity of the implications governing their applicability. In all examples shown these implications were decidable and easy to check. It would be interesting to determine which constructs can be expressed by decidable conditions. We are currently implementing a construction method in STeP (Stanford Temporal Prover) [1], using decision procedures to decide applicability, to experiment with the various constructs.

7. REFERENCES

[1] Bjørner, N. S., Browne, A., Colón, M., Finkbeiner, B., Manna, Z., Sipma, H. B., and Uribe, T. E. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design 16*, 3 (June 2000), 227–270.

[2] Colón, M. A., and Uribe, T. E. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. 10^{th} Intl. Conference on Computer Aided Verification* (July 1998), A. J. Hu and M. Y. Vardi, Eds., vol. 1427 of *LNCS*, Springer-Verlag, pp. 293–304.

[3] Finkbeiner, B., Manna, Z., and Sipma, H. B. Deductive verification of modular systems. In *Compositionality: The Significant Difference, COMPOS'97* (Dec. 1998), W.-P. de Roever, H. Langmaack, and A. Pnueli, Eds., vol. 1536 of *LNCS*, Springer-Verlag, pp. 239–275.

[4] Fisler, K., and Krishnamurthi, S. Modular verification of collaboration-based software design. In *International Conference on Foundations of Software Engineering* (2001).

[5] Katz, S. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Sys. 15*, 2 (April 1993), 337–356.

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. Getting started with AspectJ. *Communications of the ACM 44*, 10 (October 2001), 59–65.

[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (1997), vol. 1241 of *LNCS*, Springer-Verlag.

[8] Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (1974), 435–455.

[9] Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

On Composition and Reuse of Aspects

Jörg Kienzle, Yang Yu, Jie Xiong
School of Computer Science
McGill University
Montreal, QC H3A 2A7
Canada
contact: Joerg.Kienzle@mcgill.ca

Abstract

This position paper investigates the possibilities of separation, modularization and reuse offered by aspect-orientation, concentrating not on the technical or syntactic problems, but on the inherent issues resulting from inter-aspect dependencies. An aspect is defined based on the services it provides, on the services it requires and on the services it removes from other aspects. A classification of aspects is established based on the way they interact with each other and on the way their functionality is triggered. Composition rules and the weavability criteria are defined based on this classification. Moreover, the impact of the dependencies of aspects on the level of achievable reuse is analyzed. Finally, the paper shows how the general ideas apply to the aspect-oriented programming environment AspectJ.

1 Introduction

Separation of concerns is a fundamental principle of software engineering that in its most general form refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose. The benefits of a successful modularization of concerns during the implementation phase are obvious: simpler code structure resulting in improved readability of program code, program code that is easier to customize and adapt to new situations, increased possibilities for reuse.

In order to help developers, software development methods, e.g. the Unified Process [1], define a step-by-step process that leads the development team from an initial requirements document through to the final implementation [2]. Most approaches start by analyzing the system requirements based on use cases, which capture the expectations that the final users of the software may have. In a sense they focus on the different concerns of the end-users. During the design phase however, most approaches concentrate on elaborating an object-oriented design, i.e. decomposing the system into objects, each of them providing a well-defined part of the main functionality of the system. As a result, secondary functionality, e.g. distribution support, is often poorly encapsulated. This phenomenon is known as the "tyranny of the dominant decomposition" [3], and aspect-orientation [4] might be a possible way to counter it.

Of course, the notion of main functionality is relative. It has never been precisely defined, but is usually used to denote what is particular to a certain piece of software. These days, general mechanisms, for instance mechanisms that deal with concurrency and failures, distribution, or security, are considered secondary functionality.

This classification into main and secondary functionality, often also referred to as *functional* and *non-functional aspects*, is very unfortunate. It somehow conveys the feeling that there are important concerns and less important concerns during the development of a piece of software, which often leads to the mistake that only the functional part of an application is developed following software engineering principles, and the non-functional part, e.g. fault tolerance, is added later on.

It is our firm belief that there are no such things like non-functional aspects. Every concern of a certain piece of software is important, is part of its functionality. During design all concerns must be considered and integrated in order to obtain an elegant solution.

What are called non-functional aspects are actually concerns that are more general, i.e. they are likely to be present in other applications as well. Of course, it is tempting to separate these aspects from the other functionalities of the application and make then "generic", meaning that modularize them in such a way that they can be easily reused in other contexts and applications. The idea is very legitimate and it does not take long to convince any sensible programmer that such a separation would be great. Aspect-orientation might just be the right way to achieve this kind of separation.

This paper investigates the possibilities of separation and reuse offered by aspect-orientation, concentrating not on the technical or syntactic problems, but on the inherent issues resulting from inter-aspect dependencies. Section 2 defines the essence of an aspect based on the services it

provides and on the services it requires from other aspects. Section 3 classifies aspects according to the way they interact with each other. Based on this classification, Section 4 provides composition rules for aspects and Section 5 examines reusability issues. Section 6 illustrates how the presented ideas apply to one of the main-stream aspect-oriented development environments, AspectJ. Section 7 takes a closer look at circular dependencies. Section 8 presents recommendations for aspect developers, and the last section summarizes the results of this work.

2 Aspects

For the subsequent discussion, it is important to specify clearly what we mean when we talk about an aspect.

From our understanding, an aspect at the design and implementation level is a main abstraction that encapsulates that part of the design solution that addresses a certain concern expressed at the analysis level. On on hand, the aspect provides a certain functionality: it implements the concern. We'll designate the set of the services it provides *P*. Services can be seen as the entry points or interface offered to the rest of the system. On the other hand, the aspect may depend on functionality offered by other aspects. The set of services it depends on is named *D*. Optionally, an aspect might remove functionality of other aspects. The set of services it removes is named *R*. Obviously, *R* is a subset of *D*. An aspect is therefore categorized by the three sets *P*, *D*, and *R*.

What is needed to accurately describe a *service* is intentionally left open. On one end, specifying the complete semantics of an aspect is a challenging task, and out of the scope of this paper. One the other end, object-oriented programming languages often just use method signatures to specify their interface to the outside world. Applying this idea to aspects would mean specifying the signatures of *P*, *D*, and *R*.

If we want to use UML to depict an aspect, we might be tempted to use the representation of a class or interface. Unfortunately, these constructs only show what a component provides to the environment, and not what it requires from others. UML stereotypes make it possible to extend the base UML concepts and add additional meaning to them. Fig. 1 shows an `<<aspect>>` stereotype with three new compartments: *Provides*, *Depends On* and *Removes*. It is not clear if an aspect should be seen as an extension of the UML class, of the UML package or rather an extension of the UML collaboration. Discussions are still in progress [5, 6].

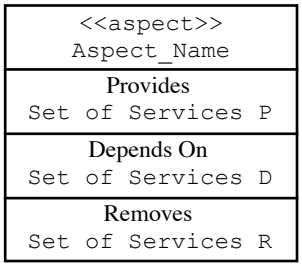


Fig. 1: UML Representation of an Aspect

3 Aspect Interaction

In this section we attempt to classify aspects according to the way they interact with each other. We have established two classification criteria: the *activation mechanism* and the *dependencies*.

3.1 Activation Mechanism

The activation mechanism of an aspect is determined by analyzing *when* the aspect delivers its functionality. There are two different kinds: *autonomous* and *triggered*.

Autonomous: An aspect is autonomous if it can act on its own, i.e. it does not need to be stimulated to deliver its functionality. It typically performs its duties continuously or periodically.

Triggered: Initially, a triggered aspect is passive. It waits for some other part of the application to activate it, and only then it delivers its service.

This classification is similar to the one found in object-orientation, where one distinguishes active and passive objects [7]. Active objects act autonomously, whereas passive object must be triggered, i.e. only execute methods when they are called from the outside.

Of course this classification is not absolute. An aspect may provide autonomous services and triggered ones, similar to the Time-Triggered Message-Triggered Objects presented in [8], which provide periodically executing services as well as services triggered by messages. For the sake of clarity, however, such mixing of activation mechanisms is discouraged.

3.2 Dependencies

We essentially distinguish three different kinds of dependencies: *orthogonal*, *uni-directional* and *circular*.

The functionality that an *orthogonal* aspect provides to an application is completely independent from the other functionalities of the application. The only thing it might depend on is activation time (see Section 3.1 above), or general application-independent information provided by the run-time environment, e.g. information on the virtual machine, current method name, etc.

Unfortunately, such aspects are not very common. An example of an autonomous orthogonal aspect is a clock counter. Every second, the counter is increased by one. No explicit triggering is needed (probably the counter is implemented using an independent thread or interrupts), and there are no shared data structures between the clock aspect and others. Measuring the time elapsed between two events can be seen as a triggered orthogonal aspect. One might think that there is a semantic dependency of such a timing aspect on the part of the application it actually performs timing on. This is, however, not true. The dependency is only on the fact that the aspect has to be triggered twice: once to start the timing, and a second time to stop it.

One of the most popular orthogonal triggered aspects is logging. For debugging purposes, a logging aspect can be applied to various places in an application to print out stack traces, etc.

A *uni-directional* aspect depends on some functionality (service or data) offered by other aspects in the application. Without this functionality, it can not deliver its services. Among uni-directional aspects, we can further distinguish between *uni-directional preserving* and *uni-directional modifying* ones.

Uni-directional preserving aspects provide new services based on services of other aspects, but do not alter or hide the other services in any way. The properties and functionalities of the other aspects are preserved.

[9] presents an example of two triggered uni-directional preserving aspects. It describes an aspect-oriented implementation of a telecom application that handles phone calls. In order to set up correct billing, the elapsed time of long distance phone calls must be measured. The long distance timing aspect uni-directionally depends on the call aspect, adding timing information to the calls. It is not orthogonal, because it has to associate timing with calls, and therefore depends on the existence of the call aspect. The billing aspect in turn depends on the call and the timing aspect, for it has to know the calls source and destination city, and the elapsed time in order to calculate the total cost. In the same context one can imagine an aspect that periodically collects statistical information on long distance calls, e.g. the average length of calls. This is an example of an autonomous uni-directional preserving aspect that depends on the call and the timing aspect.

A *uni-directional modifying* aspect replaces or modifies functionality of some other part of an application, but it does this transparently; the other aspect is not aware of this, and therefore does not have to behave differently. In a sense, a uni-directional modifying aspect wraps around or encapsulates some services provided by other aspects. As a result, some of the original services might not be provided anymore.

As an example of a uni-directional modifying aspect, imagine a typical banking application. Some banks (at least Swiss banks) allow good clients to overdraw their account. Clients with a bad credit history on the other hand are not be allowed to do this. The desired effect can be achieved by encapsulating in one aspect the account behavior, and design an additional aspect that denies withdraw requests in case of insufficient funds. The additional aspect removes the withdraw service from the account aspect.

Circular dependency is the strongest form of dependency. It occurs when several aspects are mutually dependent. The simplest form is encountered when two aspects depend on each other, i.e. the first aspect requires some service provided by a second aspect, which, in turn, can only deliver its service with the help of the first one. Another way of looking at this from the perspective of an aspect that you are adding to an application is the following: if in order to make the overall application work with the new aspect it is necessary to modify other aspects, then there is circular dependency.

An example of a circular aspect has been presented in [10]. In this example, a transaction aspect is added to a previously non-transactional application, allowing the application to deal with concurrency and failures. The aspect itself provides the run-time support for transactions, making it possible to execute methods transactionally. However, the application must state which method calls it wants to make transactional, and what actions should be taken in case a transaction aborts due to a failure.

Circular-dependent aspects are so tightly coupled that one might argue that it makes no sense to consider each aspect separately. This first impression will be confirmed when considering composition and reuse later on. It is often simpler to treat them as a single aspect. The set of services the single aspect provides is the union of the services the individual aspects provide, and likewise for the set of services it depends on and the set of services it removes. In the following sections we do not consider circular-dependent aspects, they will be revisited in section 7.

The following table summarizes the classification established in this section:

Class of Aspect	Restriction
Orthogonal	D = ø
Uni-directional preserving	R = ø
Uni-directional modifying	no restriction

Table 1: Classification of Aspects

4 Composition Rules

In AOP, the so-called *aspect weaver* composes the different aspects to form the final application. This composi-

tion can be done statically, i.e. at compile-time, or even dynamically during the execution of the application. Implementing such an aspect weaver is far from trivial, and there are lots of technical issues that need to be addressed when composing aspects. In this section, however, we will concentrate on the more fundamental problems of aspect composition. Even though a set of aspects might be technically composable, it might be conceptually impossible.

In order to simplify the discussion, we introduce the notion of an *aspect group*. Aspects in an aspect group all have some dependency relationship. An executable application consists of at least one non-empty aspect group, containing at least one autonomous aspect. Initially, the set of aspect groups that forms the final application is empty. Step by step, additional aspects are added. The set of aspect groups that form the final application is called a *configuration*.

If we represent aspects as nodes, and dependencies as directed edges, the representation of a configuration takes the form of a directed acyclic graph (short DAG) as shown in Fig. 2. We'll call it the *configuration dependency graph*. Each *component* of the dependency graph forms an aspect group.

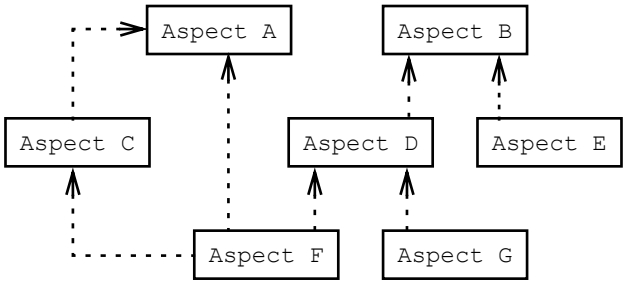


Fig. 2: A Configuration Dependency Graph

The composition rules for aspects in this section is based on the classification presented in the previous section.

Orthogonal aspects are very flexible — due to their orthogonality there are no restrictions on composing such aspects with others. When adding an orthogonal aspect to the final application, a new aspect group is created, i.e. a new component is added to the graph.

Uni-directional aspects must be added to an already existing aspect group. The set of services that the aspect requires must be provided by the aspects that are already in the group. It is also possible to combine aspect groups, i.e. join previously separated components of the graph, in order to obtain the required set of services.

4.1 Weavability

An interesting problem is the *weavability problem*, i.e. determining if a given set of aspects can be composed in such a way that all service requirements are fulfilled.

In graph theory, this is equivalent to solving a multi-commodity flow feasibility problem [11] with additional node constraints. The graph to be analyzed contains one node for each aspect and is fully connected. Every type of service will be considered a separate flow. An aspect that provides a certain service is a source for the flow (provides one flow unit). An aspect that removes the service is a sink (consumes one flow unit). Aspects that depend on the service are mandatory transshipment nodes for the corresponding flow. They can be modeled by an additional constraint that states that the sum of incoming flows for this node must be equal to one.

If and only if there exists a feasible flow, then the application is weavable. By calculating the flow distribution that uses the lowest number of arcs, and then inversing all arcs, we obtain the dependency graph of the final application.

5 Making Aspects Reusable

One of the major encouragements for using AOP is reuse. After having identified a certain concern, the idea is that AOP should allow one to modularize and implement this concern in an aspect. Later on, this aspect should be usable in every application that exhibits the need for the concern. Again, there are technical issues that must be solved in order to make aspects reusable, e.g. how to specify the required, provided and removed services in a concise way. In this section, however, we will concentrate on the obstacles introduced by aspect dependencies.

An even stronger form of reusability is *genericity*. What we want to achieve in this case is to write an aspect in such a way that it can be added to an application without disturbing the already existing structure. In other words we want to add support for a certain concern to an application just by adding the aspect that implements the concern to the configuration.

The difficulty of providing such a form of reusability increases depending on the class of aspect.

Orthogonal aspects can be reused in any context. They are generic per se. They do not depend on any other aspects, and therefore do not remove any existing services. They do not disturb any existing aspect group configuration, since they always start a new group. In the dependency graph, orthogonal aspects will show up as sinks. In Fig. 2, `Aspect A` and `Aspect B` are orthogonal aspects.

Uni-directional preserving aspects also make good candidates. Since they do not remove any services, they can be added to any aspect group that provides the required ser-

vices. Of course, when moving a uni-directional preserving aspect from one configuration into a new one, any aspects it depends on must be either moved as well, or equivalent services must already be available in the new configuration. In the dependency graph, new uni-directional aspects shows up as a source nodes. For instance, in Fig. 2, the uni-directional aspects `Aspect E`, `Aspect F` or `Aspect G` might just have been added to the configuration.

Uni-directional modifying aspects are hard to reuse, since they modify the services of aspects they depend on. They can only be added to an aspect group if it remains weavable, i.e. the new aspect does not remove services that are needed by other aspects.

6 AOP Mechanisms

This section analyses the support of the concepts presented above provided by AspectJ [12], one of the mainstream aspect-oriented programming environments.

6.1 Interface Specification

Somehow, aspect-oriented programming environments must provide a means for specifying what services an aspect provides, what services it depends on, and what services it removes. This has been an area of research for a long time, and elegant solutions to this problem still have to be found.

AspectJ takes the Java approach. The services provided by a class or aspect are determined based on Java visibility rules. Inside visible code, all potential joinpoints are advisable, meaning that they can be used as triggers or points of extension for adding additional behavior.

There is no special part where dependencies are specified. An aspect potentially depends on all other modules that are visible or that it imports. By looking closely at the code, the services it actually uses can be determined.

The services that an aspect modifies or removes are very hard to determine. Potential candidates are the destinations of `around` advice, but also `before` and `after` advice that modify the behavior of the class or aspect they are advising.

6.2 Activation Mechanism

Just as conventional object-oriented environments, aspect-oriented development environments support autonomous aspects. The autonomy of aspects is typically implemented by the underlying operating system. Autonomous aspects are either separate processes, or implemented based on threads.

AOP is however particularly well suited for implementing triggered aspects. They are usually activated by the aspect-oriented run-time, which in turn is stimulated by intercepting some specific event.

In AspectJ, for example, pointcut designators allow a developer to specify when an aspect is to be activated. For instance, it is possible to intercept calls to / and execution of methods, throwing and handling of exceptions, and reading and writing of fields. It is also possible to activate aspects based on control flow information.

6.3 Aspect Semantics

There are several mechanisms that allow an AspectJ programmer to write uni-directional aspects.

First of all, aspects are subject to the same visibility rules as normal Java classes. They can call methods, or read from / write to fields depending on their respective mode (public, protected, private) and the package they belong to. As soon as an aspect makes an explicit reference to some other class or aspect, a dependency is created.

Next, aspects can use static *introduction* to add new fields or methods to classes or aspects at compile-time. If explicit names are used, then again a dependency is created. However, the introduction mechanism allows a programmer to use pattern matching rules to defer the destination of the introduction to weave-time.

Finally, aspects can add code *before* or *after* any joinpoint defined in the code they are advising. It is even possible to wrap code *around* a joinpoint, optionally replacing the code that would have been executed at this point.

6.4 Composition

At some point, AOP environments must perform the weaving, i.e. composing all aspects of an application to yield the final application. Logically, the composition ordering is determined by the configuration dependency graph. In order to obtain a possible sequential composition order, topographical sort [13], also known as linear extension, can be applied to the configuration dependency graph.

In current aspect-oriented environments, the dependency graph information is in general encoded by the developer in a separate configuration language, or in the aspect language itself.

The latter is true for AspectJ. The pointcut designators in an aspect specify the set of joinpoints to which the advice must be applied. If several advice apply to the same joinpoint, then the developer can specify an ordering among them by using the *dominates* primitive. If some aspect A is specified to dominate some aspect B, then advice in A take precedence over advice in B. In a sense, A wraps around B (or B is nested in A). In this case, the execution order of the advice is:

- before advice in A

- before advice in B
- original code at joinpoint
- after advice in B
- after advice in A

If around advice are used, the ordering takes the following form:

- around advice in A
  (optional around advice in B
  (optional original code at joinpoint))

6.5 Drawing the Line for Dependencies

As we have seen in the previous section, in order to achieve high reusability or even genericity, a developer of an aspect should strive for low dependencies. As strange as it might seem, dependency does not only depend on the nature of the problem, but also on the power of the weaving mechanism. Surprisingly, some dependencies can be replaced by exploiting the activation mechanism in a clever way.

To illustrate this idea, consider a typical bank account, implemented as an aspect. In addition, there is a security policy that states that the account balance should not drop below zero. This policy is implemented in a separate security aspect. At first one might think that the security aspect is uni-directionally dependent on the account aspect, for it must monitor all changes to the state of the balance of the account.

It turns out that this is not necessarily true. The security aspect can be turned into a orthogonal one that prevents any numerical value to drop below zero. It is the weaving mechanism that links it to the account, i.e. activates the security aspect on every change of state of the account balance.

In AspectJ, for instance, the account would be implemented as a normal Java class with a balance field. The security aspect would be implemented as an aspect containing a before advice that verifies that the field value is higher than the amount passed to the withdraw method. The dependencies on class fields can be removed by using AspectJ run-time information. The triggering of the aspect, i.e. intercepting every write access to the balance attribute of the account, is done in the pointcut definition.

7 Circular Aspects Revisited

After having examined composition, reuse and support mechanisms we can now reexamine circular-dependent aspects. Several reasons push to believe that they should be considered a single aspect:

- Composition: When composing an application, a set of circular-dependent aspects must be added to an configuration as a whole. Moreover, it is only possible if the configuration provides the union of the services needed by each circular-dependent aspect.
- A set of circular-dependent aspects can only be reused as a group. The added services are the union of all services provided by the circular-dependent aspects.
- During the weaving process, a set of circular-dependent aspects must conceptually be woven at the same time. This may actually lead to implementation problems, similar to the problems encountered when compiling mutually dependent source files.

However, in certain situations it might make sense to consider them separately, e.g. if one aspect of the group is fairly generic. This is the case in the previously mentioned example [10], where transaction support has been implemented as a separate aspect in AspectJ. Transactions are a generic concept that can be applied to parts of an already existing application. However, the application wants to be aware of this, especially when a transaction aborts due to some underlying failure. In this case, the application might want to try the same transaction again, or decide to perform some alternative computation, and / or inform the user of the failure, etc. As a result, the application and the transaction aspect are tightly coupled.

What we suggest in this case is to try and extract that part of the application aspect that deals with transactions and make it a separate aspect. As a result, the two circular dependent aspects (application and transaction) can be transformed into an orthogonal and two uni-directional ones (the application without transaction handling, transactions, and the application-specific transaction handling part). This is illustrated in Fig. 3. The feasibility of such a transformation again depends heavily on the expressiveness and power of the weaving support. If this can be achieved for the transaction example using AspectJ remains to be explored.

Of course, programmers must be aware that the new application aspect and the application-specific transaction handling aspect have very tight semantic dependencies, although physically separated. Modifying the application aspect most probably also requires modifying the transaction handling one.

8 Discussion

As we have seen in the previous sections, the dependencies of an aspect have a profound impact on the ways it can be composed with others and on the possibilities of reuse.

Therefore, when developing an aspect that is to be made reusable or even generic, a programmer should first determine the semantic nature of the concern that is to be modu-

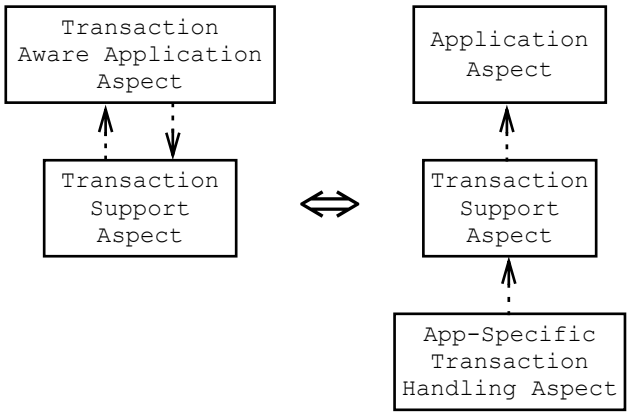


Fig. 3: Transforming Circular-dependent Aspects into Uni-directional Ones

larized. This will provide a hint on what degree of decoupling might be achievable. Next, the developer should try and classify the aspect according to the rules mentioned above.

Orthogonal aspects are most flexible, followed by uni-directional preserving ones. Uni-directional modifying aspects are not easily reusable, since adding them to an application may compromise weavability. Circular-dependent aspects should, if possible, be transformed into several uni-directional ones.

The weaving mechanism offered by the aspect-oriented environment has also an important impact on the dependencies. If it is not powerful enough, or the weaving language is not expressive enough, then additional dependencies might be artificially introduced into the system. On the other hand, exploiting the power of the aspect weaver and aspect run-time information might make it possible to remove dependencies. Imagine an aspect that monitors some data, and triggers some action if the data changes. Such an aspect would fall into the autonomous uni-directional category, since it is dependent on the data it monitors. However, if the aspect run-time allows activation of aspects based on data changes[1], then the dependency can be removed from the aspect. In a sense, the dependency is re-introduced later at weave-time, when the actual configuration is assembled. As a result, the monitoring aspect now is triggered and orthogonal, and hence can be reused in a straightforward way.

Based on these observations, we encourage designers of aspect-oriented programming environments to conduct further research in this direction. For instance, the decision for adding new features such as new pointcut designators to AspectJ should be based on whether or not such a new feature would make it possible to remove a certain kind of dependency.

1.AspectJ, for instance, allows triggering aspects when a field of a class is modified.

9 Conclusion

In this position paper we have investigated the possibilities of separation, modularization and reuse offered by aspect-orientation in general.

We have defined an aspect based on the services it provides, on the services it requires from other aspects and on the services it removes. Furthermore, a classification of aspects has been established. Aspects can be *autonomous* or *triggered*, depending on the activation mechanism. The dependencies lead to a categorization into *orthogonal*, *uni-directional preserving*, *uni-directional modifying*, and *circular-dependent* aspects. The influence of the power of the weaving mechanism on dependency has been highlighted.

Composition rules have been established based on these criteria, and the notion of weavability has been defined based on flow feasibility analysis. Likewise, the impact of the semantic nature of aspects on the level of achievable reuse has been analyzed.

Finally, we have presented how the general ideas of this paper apply to the aspect-oriented programming environment AspectJ, and made recommendations for determining the usefulness of new features.

10 Acknowledgments

The authors would like to thank the anonymous reviewers of the FOAL and SPLAT workshop committees for their detailed comments.

11 References

[1] Jacobson, I.; Booch, G.; Rumbaugh, J.: *The Unified Software Development Process*, Addison Wesley, Reading, MA, USA, 1999.

[2] Hutt, Andrew T. F.: *Object Analysis and Design – Description of Methods*. Object Management Group, John Wiley & Sons, Inc., 1994.

[3] Tarr, P. L., et al.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In *Proceedings of the 21st International Conference on Software Engineering (ICSE'1999)*, pp. 107-119, IEEE Computer Society Press / ACM Press, 1999.

[4] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: "Discussing Aspects of AOP". Communications of the ACM 44 (10), pp. 33 – 38, October 2001.

[5] First International Workshop on Aspect-Oriented Modeling with UML. Held at the *First International*

Conference on Aspect-Oriented Software Development, April 22-26, 2002, Enschede, The Netherlands.

[6] Second International Workshop on Aspect-Oriented Modeling with UML. Held at the *Fifth International Conference on the Unified Modeling Language - the Language and its Applications*, September 30 - October 4, 2002, Dresden, Germany.

[7] Briot, J.-P.; Guerraoui, R.; Lohr, K.-P.: "Concurrency and Distribution in Object-Oriented Programming", *ACM Computing Surveys **30**(3)*, September 1998, pp. 291 – 329.

[8] Kim, K.H.; Masaki, Ishida; Liu, Juqiang: "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation". In *Proceedings of the second IEEE CS International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'99)*, pp. 54 - 63, St. Malo, France, May 1999.

[9] The AspectJ Team: *The AspectJ Programming Guide*, Xerox Corporation, February 2002.

[10] Kienzle, J.; Guerraoui, R.: "AOP — Does it make sense? The case of concurrency and failures". In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, pp. 37 - 54, Malaga, Spain, June 2002, Lecture Notes in Computer Science **2374**, Springer Verlag, 2002.

[11] Cook, W. J.; Cunningham, W. H.; Pulleyblank, W. R.; Schrijver, A: *Combinatorial Optimization*. John Wiley and Sons, Inc. 1998.

[12] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersen, M.; Palm, J.; Griswold, W. G.: "An Overview of AspectJ". In *Proceedings of the 15th European Conference on Object–Oriented Programming (ECOOP 2001)*, pp. 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001, Lecture Notes in Computer Science **2072**, Springer Verlag, 2001.

[13] Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, USA, 1987.

TinyC^2: Towards building a dynamic weaving aspect language for C *

Charles Zhang and Hans-Arno Jacobsen
Department of Electrical and Computer Engineering
and Department of Computer Science
University of Toronto
10 King's College Circle
Toronto, Ontario, Canada
{czhang,jacobsen}@eecg.toronto.edu

ABSTRACT

The runtime behaviors of software systems are often subject to alteration or intervention after their development cycles for various reasons such as performance profiling, debugging, code specialization, and more. There are two separate domains related to the instrumentation of software systems, one being various performance measurement and instrumentation tools, the other the new aspect oriented programming (AOP) paradigm. This paper describes TinyC^2 language, a language approach which experiments with the idea of implementing an aspect oriented language based upon existing system instrumentation techniques. Like other aspect oriented languages, TinyC^2 uses new language constructs to allow programmers to intentionally compose systems in the dimensions of both components and aspects. In this paper, we discuss both the grammatical features and the compiler architecture of the TinyC^2 language. Through the TinyC^2 implementation, we demonstrate that a language approach can well bridge the gap between the AOP paradigm and the existing system instrumentation technologies. It greatly simplifies code instrumentation effort and provides runtime optimization at the application level.

Keywords

Aspect Oriented Programming, Compiler, Dynamic Instrumentation, Dynamic Weaving, Source-to-source translation Proceedings

1. INTRODUCTION

Programming methodologies have evolved from direct machine-level coding to object-oriented programming. Good modularization capability in the programming language design allows software architects to successfully tackle two issues: the ever growing complexity of software systems and the increasing diversity and volatility of the execution environment. Besides improving language designs, there has also been extensive work on finding better compiling techniques to provide effective adaptations for software systems and to efficiently support a wide spectrum of hardware platform and computing resources [6, 5] that change dynamically. However, compiler-based program adaptation and optimization techniques are powerful but limited if the optimization involves changing the functional behavior of the system. These optimization tasks include adaptations to many domain specific characteristics, such as state validation conditions, synchronization strategies, logging strategies, and many others. It is very difficult to build compilers to make such application level decisions flexibly.

To overcome this difficulty, it becomes necessary to perform post-development transformation to large software systems according to specific usage scenarios. The post-development transformation mainly includes modifications made to software systems after their development cycles. A major stream of manipulation techniques includes tools that provide source-code level instrumentation, as in SvPablo[1], and post-compilation instrumentation techniques as in jContractor[2] and Vulcan [4]. Dyninst [7] and the Paradyn[3] performance tools provide runtime instrumentation to C/C++ systems. Another stream of program manipulation techniques mainly belong to the aspect oriented programming paradigms [9], where "instrumentation" has the first-class status in the language design and can be used to compose system functionality. AOP advocates composing systems using different sets of models and leaving the integration work to the AOP compiler which is also referred to as the aspect weaver.

Code instrumentation techniques and aspect oriented programming are two fields that are developed independently. We think that those two domains are fundamentally com-

*In: Foundation of Aspect Oriented Languages Workshop in conjunction with 2nd AOSD Conference 2003, Boston, MA.

[1]SvPablo: A Graphical Source Code Browser for Performance Tuning and Visualization `http://www-pablo.cs.uiuc.edu/Project/SVPablo/SvPabloOverview.htm`
[2]Java Implementation of Design By Contract for the Java Language `http://jcontractor.sourceforge.net/`
[3]Paradyn. `http://www.cs.wisc.edu/~paradyn/`

patible as they both perform a certain type of after-the-fact transformation to the existing software systems. An aspect oriented language provides a more powerful approach in terms of methodology. We think that the various code instrumentation techniques can be treated as means to realizing the methodology in practice. The main motivation of our work is to experiment with such ideas by developing an aspect language using existing code instrumentation techniques. The advantage of using a hybrid language is two-fold. Firstly, a hybrid language design which decouples the language semantics from the backend implementation platform can increase the configurability and the adaptability of the aspect language. The compiler is able to readily take advantage of the advances in the code instrumentation domain by selecting different lower-level implementation strategies to instrument the system, i.e. to weave aspects, under different circumstances. Secondly, since a language provides a high level abstraction of the instrumentation semantics, it is easy to understand, to change, and to maintain the instrumentation code. This technique is also applied in [3] and [8].

The second motivation of our work is that for most of the AOP languages today, including AspectJ[4], Hyper/J[5], AspectC[6] and AspectC++[7], the transformation of programs is done statically either at the source-code level or at the bytecode level. To maximize the benefit of multi-dimensional programming, it is desirable to have the support for dynamic transformation since a lot of platform specific parameters are not available until runtime. HandiWrap [1] is a runtime weaving aspect language for Java. In the C/C++ programming domain, we are not aware of any previous work in aspect languages that provide dynamic weaving. The runtime weaving property is directly supported by the Dyninst library. We are interested to see how an aspect oriented language can take advantages of platforms like Dyninst in supporting dynamic adaptations.

We have developed the TinyC^2 language, which is a prototype aspect language. The language is designed to be an extension of the C language with new language constructs to enable the composition of aspect programs. This is also a common language design approach used in AspectJ and AspectC++. The compiler of TinyC^2 is essentially a source-to-source translator that translates C statements to the API instructions of the target instrumentation tool. We construct the compiler to be independent of any particular instrumentation techniques, thus, give the compiler the flexibility of switching to different instrumentation tools. Currently, we have implemented support for the Dyninst runtime instrumentation platform. Due to the runtime instrumentation nature of Dyninst, TinyC^2 can be treated as the runtime weaving aspect language.

The rest of the paper is organized as follows: Section 2 presents the related work regarding aspect oriented language designs. Section 3 presents a detailed description of the new language features of TinyC^2. The architecture of the compiler is also discussed in this section. Section 4 uses three case studies to demonstrate the effectiveness of the dynamic weaving nature of TinyC^2 in addressing runtime crosscutting concerns. Section 5 presents runtime characteristics of TinyC^2. Section 6 concludes the paper.

[4]AspectJ `http://www.aspectj.org`
[5]HyperJ `http://www.alphaworks.ibm.com/tech/hyperj`
[6]AspectC `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`
[7]AspectC++ `http://www.aspectc.org`

2. RELATED WORK

There are a number of aspect oriented programming languages in C and Java flavours. AspectJ adds an aspect oriented extension to the Java programming language. **Aspect**s are AspectJ's units of modularity. They are defined in terms of pointcuts, advice, and introductions. By adding these simple constructs, AspectJ enables the clean modularization of crosscutting concerns such as synchronization, context-sensitive behavior, and multi-object protocols.

Hyper/J is developed by IBM. It also supports multi-dimensional separation of concerns for Java. It provides the ability to identify concerns, specifies modules in terms of those concerns, and synthesizes systems and components by integrating those modules. It operates on standard Java class files, without need of source, and produces new class files to be used for execution.

AspectC++ is an application of the AspectJ approach to C++. It is a set of C++ language extensions to facilitate AOP with C++. It provides language features that allow a highly modular and thus easily configurable implementation of monitoring tasks and supports reuse of common implementations. AspectC++ offers virtual pointcuts and aspect inheritance to support the reuse of aspects. AspectC is an extension to the C language based on the AspectJ technologies. It is being developed concurrently with the a-kernel[8] project at UBC.

MDL [12] is a language built by the authors of Dyninst. It is specifically designed for performing runtime instrumentation using the Paradyn runtime code generation platform. The language is specialized for writing instrumentation requests in terms of performance metrics. The MDL code is parsed and translated to Paradyn instructions. Although the authors of MDL do not mention AOP, since their language can capture crosscutting concerns, we categorize it as one type of aspect language.

3. THE TINYC^2 LANGUAGE

The design goal of the TinyC^2 language is to provide a language perspective in terms of code instrumentation, and, at the same time, to establish a framework for implementing a post-compilation weaving aspect language that uses the C syntax and a third party instrumentation tool as the backend. The rest of the section describes the language in detail from both the syntactic point of view and the compiler architecture perspective.

3.1 Language Features

Using aspect oriented programming terms, the component programs of TinyC^2 can be composed in the C language. The aspect program is composed using TinyC^2. Similar to AspectJ, TinyC^2 implements standard C grammar rules

[8]a-Kernel `http://www.cs.ubc.ca/labs/spl/projects/a-kernel.html`

with the addition of a few new syntactic constructs. Programmer can use the regular C syntax to compose code blocks. However, the basic modularization units in TinyC^2 are not functions as in C but "snippet"s. A snippet is a unit of aspect implementation. It encapsulates a code block and defines the "weaving" points in the component program where the aspect code is inserted. Snippets are functionally equivalent to the "joinpoint" and "advice" concepts in an "aspect" module in AspectJ.

```
void trace(char *);                                              1
onentry Service(int size) : (int totalsize)                      2
{                                                                3
  trace("function service is called\n");                         4
  if(size>0)                                                     5
  {                                                              6
        totalsize=totalsize+size;                                7
  }                                                              8
                                                                 9
}                                                                10
onexit int retv Service(int size) : (int totalsize)              11
{                                                                12
  trace("function service is exiting\n");                        13
  if(retv<0)                                                     14
  {                                                              15
        totalsize=totalsize-size;                                16
  }                                                              17
                                                                 18
}                                                                19
                                                                 20
```

Figure 1: Snippet: onexit and onentry constructs

Let us look at the constructs of "snippet"s more closely through Figure 1. This code snippet illustrates how to implement the typical logging and tracing functionality as an aspect program in TinyC^2. This aspect program, like in regular C programs, first declares the prototype of the function `trace` (Line 1). The first section of the program (Line 2-10) traces the invocations of the function `Service` in the target system. That is, before the `Service` is executed, a message is logged (Line 4) and the size is added to a total size (Line 7) if the size is bigger than zero. More specifically, the `onentry` construct is defined as follows:

`onentry FunctionName` (*formals_list*) : (*formals_list*)

The construct binds the following identifiers in the component program: 1. function names and these formal parameters (arguments); 2. global variables in the component program designated by the formals after the ":".

The second code segment (Line 11 - 19) presents an example of the construct `onexit`. This snippet logs a message before the function `Service` returns. It also performs some post-invocation checking so that, if the `Service` function returns a negative value possibly meaning an error, the service size is subtracted from the total size. The `onexit` construct can be used to insert new behavior after a certain function finishes executing. We define the syntax of `onexit` as follows:

`onexit` *formal_list* `FunctionName` (*formals_list*) : (*formals_list*)

The difference of the `onexit` construct as comparing to `on entry` is that `onexit` allows us to bind to the return value of the function which is designated by the *formal* before the function name. The formal grammar definition of these two "snippet" constructs are defined using EBNF in Figure 2 and Figure 3.

```
onentry                                                          1
   :  TK_onentry ID LPAREN                                       2
      (formalParameter (COMMA formalParameter)*)?                3
      RPAREN COLN LPAREN                                         4
      (formalParameter (COMMA formalParameter)*)?                5
      RPAREN                                                     6
      block                                                      7
                                                                 8
```

Figure 2: Grammar definition for onentry

```
onexit                                                           1
   :  TK_onexit (formalParameter)? ID                            2
      LPAREN                                                     3
      (formalParameter (COMMA formalParameter)*)?                4
      RPAREN COLN LPAREN                                         5
      (formalParameter (COMMA formalParameter)*)?                6
      RPAREN                                                     7
      block                                                      8
```

Figure 3: Grammar definition for onexit

Currently, the TinyC^2 provides a simple pattern matching mechanism based on the prefix of the function names and their return types. In addition, the wildcard character "*" can be used to match all functions. The prefix-based matching can be extended to the regular-expression-based matching. The pattern can be defined using the "`group`" keyword as follows (the vertical line denotes an OR relationship):

`onexit | onentry` *formal_list* `group prefix_of_function | *` (*formals_list*) : (*formals_list*)

Currently, TinyC^2 supports integer and character computations. It supports conditional statements such as `if` and `else`. The `for` and `while` loops are also supported by the language.

3.2 Compiler Architecture

Generally speaking, the compiler of TinyC^2 is essentially a source-to-source translator built on top of the ANTLR[9] parser generator tool, formerly known as PTTCS. ANTLR uses a LL(K)-based language parsing scheme to parse a grammar file and generates the corresponding parser. The TinyC^2 compiler consists of three main components: the grammar file for the language, the lexer and parser generated from the grammar file, and the backend code translator and generator. Programs written in TinyC^2 language are translated by TinyC^2 compiler to a source file written according to

[9]ANTLR: ANother Tool for Language Recognition. http://www.antlr.org

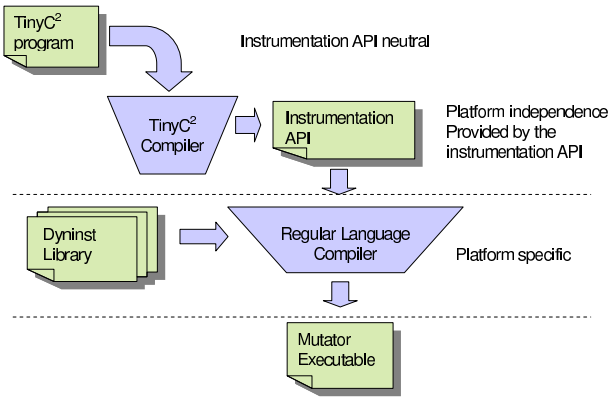


Figure 4: Compilation process of TinyC^2

the application programming interface of the target instrumentation platform. The generated source file then can be compiled again using the common language compiler of the runtime platform. It is the responsibility of the instrumentation platform to integrate the generated aspect system and the component program together. That process is illustrated by Figure 4.

The grammar understood by ANTLR is very similar to the extended BNF grammar rules with additional manipulation options that can be defined together with the grammar. Therefore, during the evaluation process of the grammar against a source code file, a large number of customized tasks can be carried out by ANTLR to perform specific analysis tasks regarding the target language, such as tree walking, code translation, and many others.

The TinyC^2 compiler is entirely composed in Java. The most fundamental component of the translator is the `Snippet` class which is the abstraction of the generated code for a particular language element in TinyC^2. The extended or subtypes of the abstract class `Snippet` provide concrete code translation for a specific code instrumentation platform. As the parser finishes parsing the entire source code file, a parse tree is built consisting of various levels of snippets. A hierarchy of snippet objects corresponds to the structure of the source program which is defined by a finite set of grammar rules. The creation of a snippet hierarchy is illustrated by the following example.

In TinyC^2, the following rule defines the conditional `if` statement.

```
statement: . . . . . .                                  1
    | TK_if LPAREN iexpr RPAREN statement                2
      (TK_else statement)?                               3
iexpr: ID  (GRT|LET)  expr                               4
```

Figure 5: Grammar definition for if statement

The rule in Figure 5 defines that an if statement consists of a token "if" followed by "(" (`LPAREN`), then by an inequality expression, the token ")" (`RPAREN`), and a statement. The product of the rule itself is also a statement. The `iexpr` rule defines that the inequality statement is in the form of an identifier followed by either ">" (`GRT`) or "<" (`LET`) symbol, and then by a compound expression. Although those rules are indifferent from the C grammar rules, ANTLR allows us to directly place program code to get executed when a matching of the rule occurs during parsing. The code is then placed verbatim in the generated parser code. Figure 6 is the same rule given above embellished with Java code.

```
statement                                                1
returns [Snippet s = null]                               2
{Snippet subexpr, ifexpr,elsexpr;}                       3
| TK_if LPAREN subexpr=iexpr                             4
       RPAREN ifexpr=statement                           5
       {                                                 6
          s = new DyninstSnippet("if");                  7
          s.addSnippet(subexpr);                         8
          s.addSnippet(ifexpr);                          9
       }                                                 10
  (TK_else elsexpr=statement                             11
      {s.addSnippet(elsexpr);}                           12
  )?                                                     13
```

Figure 6: Defining parsing behavior for if statement

To look at code example in Figure 6 more closely, line 2-3 instructs ANTLR to generate and to return a `Snippet` class for `statement` after finding a matching of the `statement` rule. Line 3 declares three sub-snippets that a snippet for the `if` statement consists of: the snippet for the condition statement, the snippet for the code block of the `if` branch, and the snippet for the code block of the `else` branch. Line 6-9 is the inserted code to actually create the snippet object of type `if` which knows how to generate the code for `if` statements. The three snippets representing the three parts of `if` blocks are inserted into the `if` snippet at line 8, 9 and 12. For example, to parse the statement: `if(a>b) b = b * a;`, a hierarchy of Snippet objects are built as illustrated in Figure 7. The left of the figure is the parse tree of the `if` statement. On the right is the image of the composition for the `Snippet` representation. Each box is the boundary of a Snippet object. The label denotes the type of the `Snippet`.

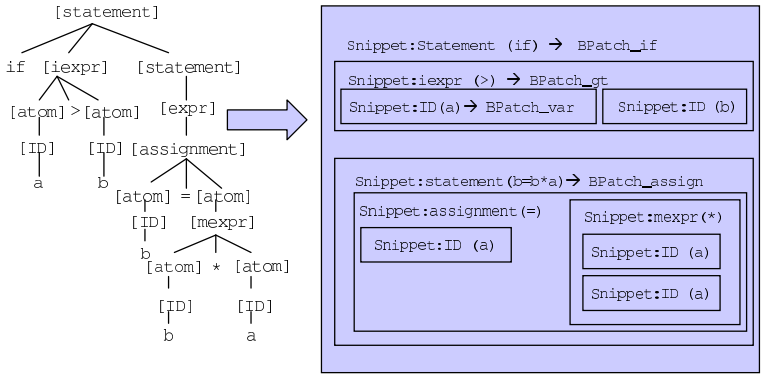


Figure 7: Snippet construction

And the symbol in the bracket represents the token(s) that the corresponding `Snippet` is responsible to translate into the target language. The labels following the arrows are the actual class types in the target instrumentation API that each corresponding snippet is translated to.

The design of the `Snippet` follows the Composite [11] architectural pattern since a complex grammar rule can be treated as a composite of the basic rules. To provide specific code translation functionality, extensions to the abstract `Snippet` class need to be defined. A concrete `Snippet` class extends the abstract method `getCode` to generate the actual target code for the corresponding language segment. In our current implementation, the translation from the TinyC^2 code to Dyninst API is carried out by the DyninstSnippet class. The code generation is initiated by invoking `getCode` at the top of the `Snippet` hierarchy which is the outer most box in Figure 7. The invocation then recursively traverses through all snippet classes after the parsing is finished. Although the instances of the DyninstSnippet class are created directly in the parser, it is easy to decouple the generated parser from any knowledge of the concrete Snippet class by using a Factory [11]. Therefore, the TinyC^2 compiler can be made backend independent by seamlessly switching to any other specialized Snippet class at class loading time.

3.3 Dynamic Weaving Mechanism

The current implementation of the backend code generator is targeted at the Dyninst runtime instrumentation platform. Therefore, the TinyC^2 code is firstly translated into C++ code in the Dyninst library API. The translated code is then compiled by a regular C++ compiler to generate a binary executable which is linked to the Dyninst instrumentation library. The executable is started with the process information of the target running system. The Dyninst library is responsible for properly inserting the code into the address space of the target program. The insertion mechanism is based on system services used by debuggers. Detailed information on how Dyninst works can be found in [7].

Leveraging the dynamic instrumentation capability of Dyninst, TinyC^2 can be classified as a dynamic weaving aspect language. The language can be used to perform traditional non-functional activities such as tracing and performance analysis. Moreover, benefiting from the modularization capability of the language, it is convenient to develop, to maintain, and to evolve sophisticated aspect programs to intentionally change the runtime behavior of the system in a systematic manner. To understand the applicability of the dynamic weaving aspect languages, we present three case studies of the language in the following section.

4. CASE STUDIES OF TINYC^2

We can use dynamic-weaving aspect languages to increase the portability, the adaptability and the reusability of shared libraries. The reusability and the portability of libraries can be greatly improved by maintaining the properties of domain-independence and platform-neutrality. However, in practice, domain specific or platform specific constraints always require adaptations in either the library code or the application layer. Some of these constraints require changing the code in a crosscutting fashion and, thus, can be modeled by aspects. The problem with statically composed libraries including those built on static-weaving aspect languages is that it is not possible to pre-configure different versions of the shared code for every application domain or platform. And it is not safe to assume that the domain or platform specific runtime constraints are always properly addressed by the applications. Thus, runtime adaptation of libraries can be a very attractive feature especially for migrating code and dynamically configured systems. In this section, we present three case studies to illustrate such adaptations and the kind of problems these adaptations solve.

4.1 State Validation

Libraries are often shared among different application domains at runtime. Same results computed by the library might subject to different interpretations and different definitions of validity depending on the domain-specific computing requirement. We use an example to illustrate this scenario. Suppose that we want to develop a math library that provides a collection of functions to perform various integer related mathematical computations. For example, in mathematical or scientific applications, there can be no limitation to the operating range of integer values. However, in some particular computing domains such as our hypothetical statistical application for populations, there can possibly be some constraints regarding the operating range of integers and, thus, a negative result should trigger an application error. Since the goal of library design is generality, one must

not hardcode the data validation logic into the library. One possible solution is to apply the validation code at every call site of the library functions that return integers. This causes the same checking code to scatter all over the places. The bloated code greatly degrades maintainability.

A more elegant and powerful solution is to compose the validation layer in TinyC^2 as aspect programs. This layer can be "woven" into the library dynamically in runtime as needed. This layer is unloaded when the library is linked into other applications. In TinyC^2, this runtime adaptation layer can be composed using the 7 lines of code in Figure 8.

```
onexit int retv group * :                                  1
(int errorno, char * errormsg)                             2
{                                                          3
 if( retv < 0 )                                            4
 {                                                         5
   errorno=ILLEGAL_RESULT;                                 6
   errormsg="Result cannot be negative";                   7
 }                                                         8
}                                                          9
```

Figure 8: Domain specific validation in TinyC^2

We use the `onexit` construct to apply the validation (line 1). The `onexit` construct binds all the functions in the target system that return integers by using the wild card ("*") matching capability of the `group` keyword. The variable `retv` binds the specific return value of these functions. Line 2 binds global variables `errorno` and `errormsg` in the target system assuming the target system supports system wide error code schemes similar to the `errorno` of Solaris. The body of the `onexit` construct is very straightforward. It sets the `errorno` to the error code `ILLEGAL_RESULT` and assigns the error message in the target system.

If we save the file in t.c, we can invoke the compiler as `java tc t.c > Mutator.cpp`. The output Mutator.cpp is displayed in Figure 9. Lines 1-11 attach to the running process identified by its process name and process ID. Lines 12-14 invoke the `findGroupProcedurePoints` method to obtain the instrumentation points for all the functions that return integers. All the instrumentation points are collected in an object of type `BPatch_pointgroup`. Lines 20-25 create three variables to hold two global variables and the variable for the return value of the function. Lines 26-43 contain a `while` loop which iterates through every instrumentation points in the collection and inserts the `if` statements at these points in the address space of the target program.

This example shows that, although Dyninst API can be used directly by programmers, it is tedious to implement even a simple functionality. The program in Dyninst is considerably more complex and lengthy (24 lines) than our aspect program (7 lines) in TinyC^2. More importantly, the TinyC^2 program greatly improves the reusability and the adaptability of library code since no changes are made to both the math library and the application code.

```
#include "BPatch.h"                                                1
int main(int argc, char** argv)                                    2
{                                                                  3
  BPatch bpatch;                                                   4
  char* name = argv[1];                                            5
  int pid = atoi(argv[2]);                                         6
  printf("Attaching to %s pid %d\\n", name, pid);                  7
  BPatch_thread * appThread =                                      8
  bpatch.attachProcess(name, pid);                                 9
  appThread->continueExecution();                                  10
  BPatch_image *appImage = appThread->getImage();                  11
  BPatch_pointgroup                                                12
  *star_exit=appImage->                                            13
  findGroupProcedurePoints("*","int",BPatch_exit);                 14
  if ( !star_exit || (*star_exit).size() == 0)                     15
  {                                                                16
   printf("Unable to find exit point to \"*\"");                   17
   exit(1);                                                        18
  }                                                                19
  BPatch_variableExpr *errorno =                                   20
  appImage->findVariable("errorno");                               21
  BPatch_variableExpr *errormsg =                                  22
  appImage->findVariable("errormsg");                              23
  BPatch_variableExpr *retv =                                      24
  appThread->malloc(*appImage->findType("int"));                   25
  while((BPatch_Vector<BPatch_point*> *point=                      26
   star_exit->getNextPoint())!=NULL)                               27
  {                                                                28
     appThread->insertSnippet(BPatch_arithExpr(                    29
     BPatch_assign, *retv, BPatch_retExpr()),                      30
     point);                                                       31
     appThread->insertSnippet(BPatch_ifExpr                        32
     (BPatch_boolExpr (BPatch_lt, *retv,                           33
     BPatch_constExpr(0)),                                         34
     BPatch_arithExpr(BPatch_assign,                               35
     *errorno, BPatch_constExpr(1))),*point);                      36
     appThread->insertSnippet(BPatch_ifExpr                        37
     (BPatch_boolExpr (BPatch_lt, *retv,                           38
      BPatch_constExpr(0)),BPatch_arithExpr                        39
      (BPatch_assign, *errormsg,                                   40
      BPatch_constExpr("Result cannot be negative")                41
     )),*point);                                                   42
  }                                                                43
  exit(1);                                                         44
}                                                                  45
```

Figure 9: Mutator.cpp:A mutator program in full Dyninst API

4.2 Adaptive Character Encoding

The bit format for representing characters has evolved from ASCII-based single-byte encoding to multi-byte character encoding such as Unicode. For legacy systems built on the single-byte character encoding, processing information encoded by multi-byte character sets can produce erroneous results. There exist several solutions to support different character encodings in legacy code. One solution aims at providing a translation layer in between applications and the legacy code. Microsoft introduces MSLU[10] to handle the encoding translation between Unicode windows applications and windows 9X operating systems which do not support Unicode. A second solution relies on smart compilers to convert the character encoding. It requires re-

[10] `http://msdn.microsoft.com/msdnmag/issues/01/10/MSLU/default.aspx`

compilation of the system. For example, gcc[11] users can use the `-fshort-wchar` switch to generate 16-bit characters rather than the default 4-byte characters.

In a dynamic setting, both solutions fall short because they require the prior knowledge of the target platform and the pre-configuration of the system before the application can run. During runtime, a library could possibly be dynamically linked into several multi-byte applications, some use one type of encoding and some use another type. It is not possible to know what type of encoding to deal with until the application is running. In these situations, we can use TinyC^2 to compose the translation layer on top of the legacy code. This translation layer can be inserted into the library dynamically at run-time when it is needed. For illustration purposes, suppose in our hypothetical library, which only supports ASCII encoding, there is a group of functions which are responsible for maintaining a global message buffer. To prevent unpredicted results, our adaptation layer should first convert the characters in the buffer from a foreign encoding to the native encoding before the buffer is processed. After the buffer is processed, the adaptation layer should convert the buffer back to its original encoding. This pre/post processing logic can be implemented by the `onentry` and `onexit` constructs of TinyC^2. Figure 10 shows the TinyC^2 code.

```
onentry group buffer_ : (char * buffer)                1
{                                                      2
  convert_encoding(buffer);                            3
}                                                      4
                                                       5
onexit group buffer_ : (char * buffer)                 6
{                                                      7
  restore_encoding(buffer);                            8
}                                                      9
```

Figure 10: Encoding adaptation layer

This TinyC^2 code uses the `group` keyword to match all functions prefixed by `buffer_`. The `onentry` block (lines 2-4) invokes an external function `convert_encoding` which is responsible for converting the buffer into the native encoding. The `onexit` block (line 8) calls another external function `restore_encoding` to restore the original encoding. The TinyC^2 compiler generates the following code in Dyninst API.

```
#include "BPatch.h"                                                    1
int main(int argc, char** argv)                                        2
{                                                                      3
  BPatch_thread * appThread =                                          4
  bpatch.attachProcess(name, pid);                                     5
  appThread->continueExecution();                                      6
  BPatch_image *appImage = appThread->getImage();                      7
  BPatch_pointgroup                                                    8
  *buffer__entry=appImage->                                            9
  findGroupProcedurePoints("buffer_","void",                           10
  BPatch_entry);                                                       11
  BPatch_pointgroup                                                    12
      *buffer__exit=appImage->                                         13
  findGroupProcedurePoints("buffer_","void",                           14
  BPatch_exit);                                                        15
  BPatch_variableExpr *buffer =                                        16
  appImage->findVariable("buffer");                                    17
  BPatch_function *convert_encodingptr =                               18
  appImage->findFunction("convert_encoding");                          19
  BPatch_Vector<BPatch_snippet *>                                      20
  convert_encoding_args;                                               21
  convert_encoding_args.push_back(buffer);                             22
  BPatch_funcCallExpr convert_encoding                                 23
  (*convert_encodingptr, convert_encoding_args);                       24
  BPatch_function *restore_encodingptr =                               25
  appImage->findFunction("restore_encoding");                          26
  BPatch_Vector<BPatch_snippet *>                                      27
  restore_encoding_args;                                               28
  restore_encoding_args.push_back(buffer);                             29
  BPatch_funcCallExpr restore_encoding                                 30
  (*restore_encodingptr, restore_encoding_args);                       31
  while((BPatch_Vector<BPatch_point*> *point=                          32
   buffer__entry->getNextPoint())!=NULL)                               33
  {                                                                    34
     appThread->                                                       35
     insertSnippet(convert_encoding,*point);                           36
  }                                                                    37
  while((BPatch_Vector<BPatch_point*> *point=                          38
   buffer__exit->getNextPoint())!=NULL)                                39
  {                                                                    40
     appThread->                                                       41
     insertSnippet(convert_exit,*point);                               42
  }                                                                    43
  exit(1);                                                             44
}                                                                      45
```

Generated encoding adaptation layer in Dyninst API

In the generated code, lines 4-15 attach to the running process and obtain two groups of instrumentation points, one being the entry points of all function prefixed by `buffer_`, the other their exit points. Lines 16-31 bind to the global message buffer and set up the function calls to `convert_encoding` and `restore_encoding`. The `onentry` and `onexit` constructs in Figure 10 are translated to two loops which insert the function calls at corresponding instrumentation points of every function in the group (lines 32-45).

4.3 Adaptive Systematic Behavior

A dynamic weaving aspect language allows us to modularize systematic properties and to build systems that are more adaptive and more efficient for specific runtime conditions. For example, middleware systems are software substrates that provide abstractions for the distributed computing entities. In a environment such as mobile computing where the platform resources and computation requirements change dynamically, it is highly desirable to configure a right set of middleware characteristics during runtime. Such high level of configurability and adaptability is hard to achieve due to non-modularized systematic properties. A typical systematic property is *Thread Safeness*. It is important for middleware systems to ensure the accesses to shared data are synchronized. However, synchronization is not always necessary for a smaller platform such as handheld devices where the underlying OS might only support a single-thread execution model due to power and memory constraints. Some middleware implementations such as TAO uses techniques

[11] `http://gcc.gnu.org/`

such as strategic locking [2] to allow fine tuning of locking schemes. These implementations suffer from performance overhead of redundant locking and unlocking if deployed on small platforms where the contention of resources should be minimized or avoided. The dynamic behaviors of applications such as the migration of services require middleware to load and unload properties such as Thread Safeness during runtime. A dynamic weaving aspect language such as TinyC^2 can help us achieve these goals.

To illustrate the TinyC^2 approach, suppose that the function `Service` is responsible for sending a buffer of characters to a remote entity. To ensure a valid read, the function acquires the buffer lock by invoking `lock_buffer` function before sending. It releases the lock by invoking the `release_buffer` function. Figure 11 presents the simple implementation in C.

```
int Service(char **buffer, int size)                    1
{                                                       2
   int ret = 0;                                         3
   lock_buffer();                                       4
   ret=network_send(socketfd,buffer, size);             5
   release_buffer();                                    6
   return ret;                                          7
}                                                       8
```

Figure 11: A synchronized buffer send

As we have discussed, statically configured systems including statically weaving aspect implementations incur runtime overhead if locking is not necessary. We now provide the TinyC^2 implementation using the `onentry` and the `onexit` constructs in Figure 12.

```
onentry Service(char** buffer, int size)                1
{                                                       2
   lock_buffer();                                       3
   //perform other operations such as checking          4
   //the buffer size                                    5
}                                                       6
                                                        7
onexit Service(char ** buffer, int size)                8
{                                                       9
   lock_release();                                      10
   //perform necessary post invocation checkings        11
}                                                       12
```

Figure 12: TinyC^2 approach to thread safeness

The TinyC^2 compiler generates Dyninst API code in Figure 13. Similar to the previous example, lines 4-8 attach to the target process. Lines 9-14 locate the entry point and the exit point of the function `Service`. Lines 15-19 locate the function `lock_buffer` insert the function to the entry point of `Service`. Lines 20-25 load the function `release_buffer` and insert it to the exit point of `Service`. TinyC^2 does not require the functions used in the aspect program such as `lock_buffer` also defined in the component program. These functions can be compiled into a dynamically shared library and linked at runtime.

```
#include "BPatch.h"                                                    1
int main(int argc, char** argv)                                        2
{                                                                      3
  BPatch bpatch;                                                       4
  BPatch_thread * appThread =                                          5
     bpatch.attachProcess(name, pid);                                  6
  appThread->continueExecution();                                      7
  BPatch_image *appImage = appThread->getImage();                      8
  BPatch_Vector<BPatch_point*> *Service_entry=                         9
  appImage->                                                           10
  findProcedurePoint("Service",BPatch_entry);                          11
  BPatch_Vector<BPatch_point*> *Service_exit=                          12
  appImage->                                                           13
  findProcedurePoint("Service",BPatch_exit);                           14
  BPatch_function *lock_bufferptr =                                    15
  appImage->findFunction("lock_buffer");                               16
  BPatch_funcCallExpr lock_buffer(*lock_bufferptr);                    17
  appThread->insertSnippet(lock_buffer,                                18
  *Service_entry);                                                     19
  BPatch_function *release_bufferptr =                                 20
  appImage->findFunction("lock_release");                              21
  BPatch_funcCallExpr release_buffer                                   22
  (*release_bufferptr);                                                23
  appThread->insertSnippet(release_buffer,                             24
  *Service_exit);                                                      25
}                                                                      26
                                                                       27
```

Figure 13: TinyC^2 approach to thread safeness

Again, our TinyC^2 implementation achieves considerable code reduction from 25 lines to 8 lines. More importantly, the synchronization facilities can be dynamically plugged in and out depending on the runtime requirements. Saving redundant locking and unlocking greatly improves the efficiency of the system.

5. RUNTIME CHARACTERISTICS OF ASPECT PROGRAMS USING DYNINST API

In this section, we examine the runtime characteristics of the application and aspect programs using addition instructions as an experiment. We are interested in two types of behaviors: 1. the "weaving" cost which is the time taken to insert the aspect code into the component program; 2. the runtime cost which is the time of computation in the dynamically inserted aspect program versus a statically written component program. We first measure the code patching cost of Dyninst. It is measured as the time taken to insert a number of "add" instructions in the target program. To measure the runtime execution overhead, we first measure the execution time of executing an increasing number of addition instructions in the component program. We then measure the same computation in the inserted aspect program. The data is collected on a Pentium IV 2GHz Linux workstation.

5.1 Code Patching Cost

Figure 14 shows the time to insert the snippet versus the number of additions in the snippet. As the size of the snippet increases, the weaving time of snippet increases rapidly. Dyninst uses the same operating system services such as

ptrace and /proc file system to communicate between the application process and the mutator process. The instrumentation code is stored in large arrays which are loaded into the application process. The arrays are used for dynamically allocating small regions of memory: one is used for instrumentation variables; the other is to hold instrumentation code. A bigger snippet occupies a larger space in the array in the application memory space. It takes longer to fetch data from a larger memory space.

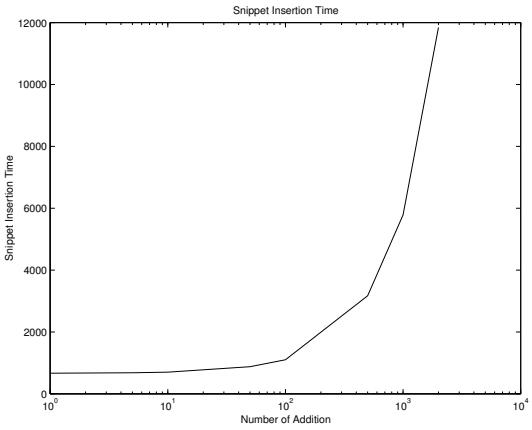


Figure 14: Code patching cost

5.2 Runtime Cost

Another important factor for dynamic weaving aspect language is the execution overhead of the aspect language as compared to carrying the same computation task in the component program. Figure 15 plots the runtime cost of performing additions in the regular C programs and in the inserted TinyC^2 code.

The running time for the same number of additions in the aspect program is significantly longer than in the component program. This can be explained by the runtime instrumentation mechanism of Dyninst. The original code in the application process branches into newly generated code through use of `trampolines` [7]. Trampolines are short sections of code that provide a way of getting from the point to the newly generated snippet. Several steps are involved here. Firstly, one or more instructions at the instrumentation point are replaced with a branch to the start of a base trampoline. Then the base trampoline code branches to a mini-trampoline. The mini-trampoline saves the current machine state and contains the code for a single snippet. At the end of the single snippet, code is placed to restore the machine state and to branch back to the base trampoline. The base trampoline executes the original instruction(s) in the application code. Therefore, there is significant management overhead for executing the aspect program in the case of Dyninst. Another reason is that since the aspect code is inserted during runtime, the code misses the static compiler optimization stage and, therefore, produces un-optimized code.

5.3 Limitations and Open Questions

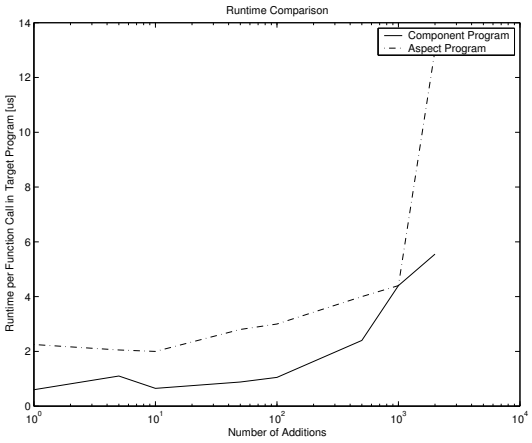


Figure 15: Runtime cost of TinyC^2 code versus regular C code

There are many limitations of the current implementation of the TinyC^2 language. Firstly, the language is being implemented as a prototype. We hope to demonstrate its capability of implementing large scale and complex aspect oriented systems by our continuous extension of the language. The second limitation comes from the limitations of Dyninst. The API of Dyninst was not designed to support aspect languages. Features such as modifying function arguments and their return values are not yet possible to implement using Dyninst. We have added a number of APIs to Dyninst to support the "group" language construct.

There are also many challenges regarding implementing dynamic weaving aspect oriented systems in general. The first category of challenges is comprised of performance related issues of dynamically woven AOP systems. Our experimental data show that the cost of computing in dynamically inserted code is considerably high. One reason is that dynamically inserted code misses the optimization stage in the compilation process which leads to un-optimized code. Intuitively, advanced compiler techniques such as dynamic optimization techniques [10] can be used to further optimized the mutated code during runtime. However, there are several issues regarding dynamic optimization. Firstly, from a compiler point of view, the newly patched code might disturb any optimization strategy that the compiler has chosen for the code. Runtime code patching can also trigger subsequent runtime optimization, which adds a considerable overhead to the overall runtime cost. Secondly, it is not clear to us if the runtime optimized code still allows us to detach the inserted aspect code on the fly as part of the dynamic adaptation. A third prominent issue is that current aspect language designs require preservations of weaving points, e.g. function identifiers in the context of the TinyC^2 language, in order for weaving to work. This is a trivial concern for static-weaving languages. However, these identifiers in the source code might disappear in the runtime code due to compiler optimization techniques such as code specialization, function inlining, and many others. Certain identifiers or symbols must be made available to aspect weavers at all time. But does the preservation of symbols decrease the optimization gain? Is there a measure of such trade-offs?

The second category of challenges concerns designing dynamic weaving languages is that whether there should be language facilities to take advantage of its dynamic nature. For example, Dyninst gives us some degree of control over the running state of the target program during the code patching process. Should the design of a dynamic weaving language gives first-status concerns to issues such as controlling the state of the target program, runtime information of the platform, optimization related tasks, and many others?

The third category of challenges includes issues regarding the security of dynamic weaving languages. That is the dynamically inserted code must comply with the security policies of the target platform. These policies could include execution privileges and copyright protections.

6. CONCLUSION

In this paper, we presented the work of TinyC^2, an aspect oriented language that is designed to syntactically extend the C programming language and to use existing code instrumentation platforms as the backend. A prototype of the language compiler is developed to support a subset of the standard C language features with a couple of additional language constructs. The backend instrumentation platform is provided by Dyninst runtime instrumentation platform.

Through this work, we demonstrate the possibility of supporting certain aspect oriented language semantics by using code instrumentation platforms. We prove the concept that code instrumentation techniques and the aspect oriented design goals are fundamentally compatible as one can be used to express the other. A language approach in bridging the two domains is viable because, as illustrated in the case study, we are able to express higher-level programming concerns in the form of TinyC^2 language and to realize those concerns through the form of code instrumentation.

It is currently not possible to have a complete evaluation of the language approach presented in this paper, since the full aspect language features are still needed to be developed. We also need to experiment with a different instrumentation tool to verify if the consistency of the language semantics can be maintained. Finally, from the experience of this work, we have encountered several issues regarding the viability of the runtime weaving aspect language design. These issues are mainly concerned with the cost of dynamically changing the runtime behavior of the system. We expect further research on advanced AOP compilers will develop solutions to these problems.

Acknowledgements

We are very grateful to Michael J. Voss who pointed out to us the similarities between runtime instrumentation techniques and aspect oriented mechanisms. The initial performance analysis and the graphs in this paper are prepared by Yiqian Ying.

7. REFERENCES

[1] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, 2002.

[2] Douglas Schmidt Michael Stal Hans Rohnert Frank Bushmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, Ltd, 1 edition, 1999.

[3] Morgan Deters Ron K. Cytron. Introduction of Program Instrumentation using Aspects. *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, pages 131–147, 2001.

[4] A. Srivastava A. Edwards and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report Technical Report MSR-TR2001 -50, Microsoft Research, One Microsoft Way, Redmond,WA, April 2001.

[5] B. Grant M. Philipose M. Mock C. Chambers S.J. Eggers. An Evaluation of Staged Run-time Optimizations in DyC. *Conference on Programming Language Design and Implementation*, May 1999.

[6] M. Arnold S. Fink D. Grove M. Hind and P.F. Sweeney. Adaptive Optimization in the Jalapeno JVM. *Object-Oriented Programming Systems, Languages and Applications*, 2000.

[7] Bryan Buck Jeffrey K. Hollingsworth. An API for runtime code patching. *Journal of Supercomputing Applications and High Performance Computing*.

[8] Charles Zhang Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. *International Conference of Aspect Oriented Software and Development*, pages 130–139, 2003.

[9] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[10] Bala Vasanth Duesterwald Evelyn Banerjia Sanjeev. Transparent dynamic optimization. Technical Report HPL-1999-77, Hewlett Packard, 1999.

[11] Erich Gamma Richard Helm Ralph Johnson John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[12] Jeffrey K. Hollingsworth Barton P. Miller Marcelo J. R. Goncalves Oscar Naim Zhichen Xu and Ling Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. *International Conference on Parallel Architectures and Compilation Techniques*, 1997.

Interference Analysis for AspectJ

Maximilian Störzer, Jens Krinke
Universität Passau
Passau, Germany
{stoerzer, krinke}@fmi.uni-passau.de

March 1, 2003

Abstract

AspectJ is a language implementing aspect-oriented programming on top of Java. Besides modification of program flow and state using *advice*, AspectJ offers language elements to statically modify existing classes by changing their position in the inheritance hierarchy or introducing new members. This can lead to binding interference, i.e. the dynamic lookup of method calls not affected directly by the aspect might change.

This paper presents methods allowing programmers to automatically check the impact of introductions and hierarchy modifications on existing programs.

1 Motivation

Aspect oriented programming (AOP) is a new paradigm in programming, extending traditional programming techniques, first introduced in [5]. Its basic idea is to encapsulate concerns which influence many modules of a given software system, so called *crosscutting concerns*, in a new module called *aspect*.

This encapsulation improves separation of concerns and can avoid invasive changes of a program if crosscutting concerns are affected by system evolution. The functionality defined in the aspect is *woven* into the base system with a so called *aspect weaver*, at compile time, load time, or even run time of the program. Here *AspectJ*—an aspect-oriented language extending Java—is considered. Main features of AspectJ are introduction, modification of class hierarchies and advice. This paper will concentrate on the first two points which are designed to statically change a given system by introducing new members in classes or modifying the structure of an inheritance hierarchy.

AOP is a very powerful technique but includes new risks, too. Changes introduced with AspectJ are not visible *directly* in the source code of the base system. Aspects are a new modularization unit usually stored in separate files. The effect of this code can influence semantics of the whole system. Tool support is necessary to reveal the impact of aspect application. To motivate this necessity, this paper presents problems related to AspectJ language constructs which might be avoided by modifying the AspectJ language itself. However, impact on language design is not in the scope of this paper.

To achieve this support, methods to determine the impact of aspect application have to be developed. As a first step, a method to decide *if* an aspect modifies base system behavior is presented. This analysis will be extended to perform an impact analysis to show *where* system behavior is influenced by an aspect.

Throughout this paper, the simple class hierarchy defined by program 1.1 will be used as an example to demonstrate aspect influence. This hierarchy will be modified using introduction and hierarchy modification and some of the classes will be declared to implement interface `I`.

This paper describes the problem emerging from these transformations, presents an algorithm to detect their effects and suggests how this information can be used to reduce flaws in a software system. Organization is as follows: Each section takes a look at a AspectJ language construct, starting with interface introduction in section 2. Section 3 presents an algorithm to detect binding interference for class introduction, section 4 for hierarchy modification. Section 5 shows how these results can be used for impact analysis. Section 6 presents an example application of this analysis for a given hierarchy. Section 7 briefly summarizes the preliminary implementation and outlines future work. Sections 8 concludes and gives an overview of related work.

Program 1.1 Example Hierarchy

```
class A { void n() {
  print("A.n()"); }}
class B extends A {
  void m() { print("B.m()"); }}
class C extends B {
  public void x() { print("C.x()"); }}
class D extends B {
  public void y() { print("D.y()"); }
  public void x() { print("D.x()"); }}
class E extends C {}
class F extends D {
  void n() { print("F.n()"); }}
class G extends B {
  void n() { print("G.n()"); }}
interface I {
  void x(); void y();
}
```

2 Interface Introduction

Introduction is an AspectJ language construct to add new members to existing classes or interfaces. The purpose of interface introduction is to provide *default implementations* of interface methods which can be used to reduce necessary work for implementation. However, if no multiple inheritance is needed an abstract superclass can often be used instead .

Usage of this feature can result in 'forgotten' implementations which may introduce flaws into a program. The compiler no longer issues an error message if a class implements an interface but does not (re)define all default implementations. To avoid flaws by 'forgotten' redefinitions a compiler warning should be given when a class uses a default method implementation provided by the interface.

A simple analysis of interface introductions can provide the necessary information. Given a class hierarchy and an aspect A, an analysis could be performed in three steps:

1. The set of interfaces for which aspect A provides default implementations has to be determined by scanning A's introductions. Let I_{def} be the set of these interfaces. For $\mathtt{I} \in I_{def}$ let *methods*(I) be the set of methods for which default implementations are given.

2. The set of classes implementing an interface $\mathtt{I} \in I_{def}$ has to be identified. Let $\mathcal{C}_{I_{def}}$ be the set of these classes.

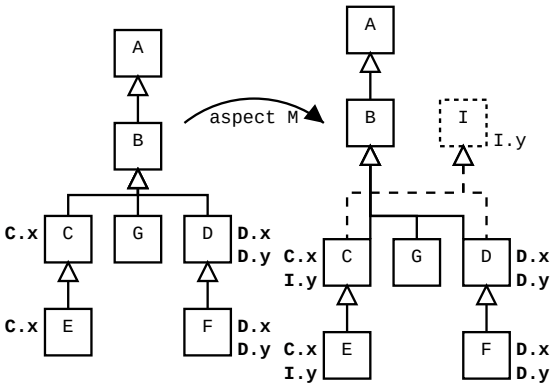


Figure 1: Using default implementations.

3. The set of classes $\mathcal{C}_{di}$ which do not provide an implementation of all interface methods (i.e. which use the default implementations) has to be determined. Let *methods*(C) be the set of all methods defined in Class C. Then

$$\mathcal{C}_{di} = \{\mathtt{C} \in \mathcal{C}_{I_{def}} \mid \exists \mathtt{I} \in I_{def} : \mathtt{C} \text{ implements } \mathtt{I} \\ \wedge \mathit{methods}(\mathtt{I}) - \mathit{methods}(\mathtt{C}) \neq \emptyset\}$$

It is sufficient to check weather all methods in *methods*(I) are implemented as other missing methods are detected by the java compiler. Note that any subclass of an affected class is influenced as well, unless it implements the necessary method and thus overrides the default implementation.

The programmer must examine affected classes to check whether the default implementation given by the interface is appropriate.

As an example consider aspect M given by program 2.1, which declares that classes C and D implement interface I and introduces a default implementation of method y to the interface.

Program 2.1 Adding interface implementation.

```
aspect M {
  declare parents: C implements I;
  declare parents: D implements I;

  public void I.y() { print("I.y()"); }
}
```

Figure 1 presents the effects of this modifications. Note that classes C and E—maybe unexpectedly—use the im-

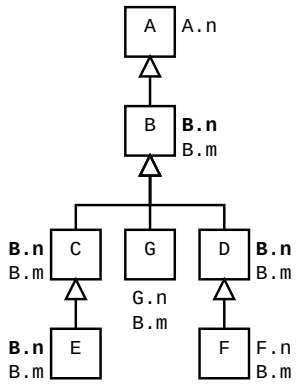


Figure 2: Example hierarchy, effects of introduction.

plementation given by `I.y`. This fact is reported by the proposed analysis.

3 Noninterference Criterion for AspectJ Introduction

In contrast to interface introduction, class introduction is more complex as program semantics may change without modifying any class directly. These effects are described in the following.

3.1 Impact of Class Introduction

Introducing members to classes can result in changes of dynamic lookup if the introduced method redefines a method of a superclass, called *dynamic interference* in [10]. However, as the term dynamic is misleading, the term *binding interference* is preferred. Consider the example hierarchy defined by program 1.1 and aspect `N` to be applied:

```
aspect N {
  void B.n() { print("B.n()"); }
}
```

This aspect introduces a method `n` to class `B`, which is already defined in superclass `A` of `B`. Any (virtual) call e.g. from class `C` now results in call of `B.n()` and not in `A.n()` as before. So, the semantics of a call to `n` has possibly changed for any object of class `B` or any subclass thereof without direct modification of these classes. Figure 2 indicates the changed lookups in bold.

The presented considerations abstract from Java access specifiers: All methods are considered `public`. Addition of access specifiers reduces the set of inherited methods (some might not be visible in the subclass), thus reducing binding interference.

If the introduced method `B.n()` redefines `A.n()` with respect to behavioral sub-typing [6], a (unknown) client of a subclass of `B` may still work as expected. However, neither Java nor AspectJ guarantees this kind of method redefinition. The described problem is a special case of the *fragile base class problem* [7]—subclasses change behavior because of changes in the superclass. Although tracking down bugs introduced by changing a base class is difficult, the problem is even worse with aspect languages as modifications of the base class are not visible if the code is viewed in isolation (i.e. without the applied aspect). To track bugs emerging from dynamic interference, impact analysis of aspect application should reveal method calls whose *dynamic lookup has changed*.

3.2 Detecting Semantical Changes

To detect semantical changes in the hierarchy, the interference criterion of [10]—informally stating that all virtual calls evaluate to the same target as before—is applied to aspects by reducing introduction to hierarchy composition. As a result, the correctness proof of the criterion can be applied to aspect introduction as well.

In contrast to Hyper/J, AspectJ is much more restrictive in the possible static modifications of the class hierarchy. Modification of system behavior is mainly achieved by using advice. However, introduction can be viewed as a hierarchy composition. Let a hierarchy $\mathcal{H}$ be defined as in [10]:

Definition 3.1 (Class Hierarchy) *A class hierarchy $\mathcal{H}$ is a set of classes and an inheritance relation: $\mathcal{H} = (\mathcal{C}, \leq)$. A class $\mathtt{C} \in \mathcal{H}$ has a name and contains a set of members. According to this definition, members$(\mathtt{C})$ does not contain inherited members that are declared in super-classes of* `C`.

To indicate the members of class `C` defined in hierarchy $\mathcal{H}$ we write *members*$_{\mathcal{H}}(\mathtt{C})$; $\mathtt{C}_{\mathcal{H}}$ references definition of class `C` in hierarchy $\mathcal{H}$.

Any AspectJ introduction can be viewed as a hierarchy composition by defining a new hierarchy induced by an aspect `A`.

Definition 3.2 (Hierarchy induced by Introduction) *Let $\mathcal{H} = (\mathcal{C}, \leq)$ be a hierarchy an aspect* `A` *is applied to. Let I be the set of introduction statements of this aspect. Elements of I have the form $(\mathtt{C},\mathtt{m})$. $\mathtt{C} \in \mathcal{C}$ indicates the class where the new member* `m` *should be introduced to. Then:*

1. $\forall \mathtt{C} \in \mathcal{H}$ *create a new empty class named* C*, add it to* $\mathcal{C}'$
2. $\forall(\mathtt{C},\mathtt{m}) \in I$ *add member* m *to the corresponding class* $\mathtt{C} \in \mathcal{C}'$ *created in (1)*
3. $(\leq') = (\leq)$ *(same inheritance relations as in* $\mathcal{H}$*)*

The hierarchy induced by I is $\mathcal{H}' = (\mathcal{C}', \leq')$.

Informally, the resulting hierarchy contains no members from the base hierarchy but any introduced member and mirrors the inheritance relations. Empty classes are possible.

As name clashes or *static interference* are considered an error by the AspectJ compiler *ajc*

$$\forall \mathtt{C} \in \mathcal{C}' : \forall \mathtt{m} \in \mathit{members}_{\mathcal{H}'}(\mathtt{C}) : \\ \mathtt{C} \in \mathcal{C} \wedge \mathtt{m} \notin \mathit{members}_{\mathcal{H}}(\mathtt{C})$$

always holds for syntactically correct AspectJ programs. AspectJ[1] does not allow overriding introductions. So only *basic compositions*, i.e. compositions without priority rules to choose from a set of possible method implementations, have to be considered.

The hierarchy induced by an aspect needs not to be syntactically correct as methods introduced by the aspect might reference methods not present in $\mathcal{H}'$ but only in $\mathcal{H}$. All these dangling references are bound after combination of the resulting hierarchies if the original AspectJ-program was correct.

The hierarchy $\mathcal{H}'$ induced by the introductions of an aspect A will now be composed with the hierarchy of the base system $\mathcal{H}$ by using a hierarchy composition operator $\oplus_s$. When working with arbitrary hierarchies, the inheritance relations of both hierarchies can be contradictory, e.g. if $(\mathtt{B},\mathtt{C}) \in \leq_1$ and $(\mathtt{C},\mathtt{B}) \in \leq_2$.

This is immpossible if a hierarchy induced by an aspect should be combined with the base hierarchy, as the resulting inheritance relation is always conflict free (here, they are identical), no collapsing of cycles is necessary and the general combination operator of [10] can simplified as follows:

Definition 3.3 (Simplified Hierarchy Composition) *Let* $\mathcal{H}_1 = (\mathcal{C}_1, \leq_1)$*,* $\mathcal{H}_2 = (\mathcal{C}_2, \leq_2)$ *be two class hierarchies with conflict free inheritance relations* $\leq_1$*,* $\leq_2$ *and no static interference. Then* $\mathcal{H}_1 \oplus_s \mathcal{H}_2 = (\mathcal{C}, \leq)$ *is defined as follows*[2]*:*

1. $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$
2. $(\leq) = (\leq_1 \cup \leq_2)$
3. $\forall \mathtt{C} \in \mathcal{C}_{\mathcal{H}} : \mathit{members}(\mathtt{C}_{\mathcal{H}}) = \mathit{members}(\mathtt{C}_{\mathcal{H}_1}) \cup \mathit{members}(\mathtt{C}_{\mathcal{H}_2})$ [3]

It is easy to see that the effect of composing $\mathcal{H}$ and $\mathcal{H}'$ using operator $\oplus_s$ has the same effects as the introductions of AspectJ: Both operations simply add the introduced members to the respective classes of the resulting hierarchy.

Following the analysis of [10], it is now possible to apply the stated noninterference criterion for AspectJ introduction as well, which informally states that all used virtual calls must evaluate to the same method as before.

3.3 Finding Changed Lookups

To test the interference criterion it has to be checked, whether the dynamic lookup for any possible call has changed. The analysis described below only needs the hierarchy and signature information as input; method bodies are *not* analyzed. This approach guarantees that the hierarchy preserves its behavior if *no* binding interference occurs at all.

For impact analysis, this information is insufficient as the set of changed lookups calculated by the subsequent analysis demands that behavior of *any* affected class together with its subclasses has to be considered as beeing changed. The reason is that methods defined in a class in $\mathcal{H}$ might transitively use a call with a changed lookup in their implementation.

To reduce the set of affected classes, a simple code scanning of an affected method for calls with changed lookup might be enough—methods only using unchanged calls in their implementation as well as calls evaluating to unaffected classes are guaranteed to work as before if only these methods are called. The call graph is an appropriate data structure to calculate all this information.

Note that newly introduced methods may very well change the state of objects, thus altering system behavior. Anyhow, introduced methods are never called by the original system as the system would not have been syntactically correct otherwise—the method did not exist in the original system[4].

[1] Referenced Version is 1.0.6.

[2] In this paper $\oplus$ will refer to $\oplus_s$.

[3] Here, $\mathit{members}(\mathtt{C}_{\mathcal{H}_j})$ indicates the set of members defined in class C in hierarchy $\mathcal{H}_j$. If $\mathtt{C} \notin \mathcal{H}_j$, then $\mathit{members}(\mathtt{C}_{\mathcal{H}_j}) = \emptyset$.

[4] Keep in mind, that advice is not considered here—advice code might call newly introduced methods.

The information necessary to check the interference criterion as well as for impact analysis is the set of changed lookups. In [10], calculation of changed lookups is more precise as only calls actually appearing in the hierarchy are examined (using points-to analysis). The method proposed here calculates any possible change in lookup due to aspect application. The loss of precision might be negligible as the set of changed lookups is much smaller (explicit introduction instead of arbitrary hierarchy combination). As an additional advantage, our algrithm is independent of a specific client, because all statically possible calls are examined.

This set of method calls can easily be calculated by a modified version of breadth first search, given by algorithm 3.1. Recall that a class hierarchy in Java (as well as in AspectJ) always defines a tree. Therefore, the inheritance relation $\leq$ always contains `java.lang.Object` as maximal element. For the algorithm let $\mathtt{C} \in \mathcal{C}$ be a class. Then $Ints(\mathtt{C})$ is the set of all methods introduced in class `C`. For the root object, define $\Delta lookup(father(root)) = \emptyset$.

Algorithm 3.1 Calculation of Changed Lookups

algorithm get-binding-interference
input: hierarchy $\mathcal{H} = (\mathcal{C}, \leq), \forall \mathtt{C} \in \mathcal{C} : Ints(\mathtt{C})$
output: $\forall \mathtt{C} \in \mathcal{C} : \Delta lookup(\mathtt{C})$

queue = $\{\max(\leq)\}$
while queue $\neq \emptyset$ do
  $\mathtt{C} = remove(queue)$
  $\Delta lookup(\mathtt{C}) = (\Delta lookup(father(\mathtt{C}))$
    $-members(\mathtt{C})) \cup Ints(\mathtt{C})$
  $\forall \mathtt{D} : \mathtt{D} \leq \mathtt{C}$ do: add `D` to queue

The changes in lookup are used as input for a subsequent impact analysis (refer to section 5). However, changes in lookup are not only due to introduction but can have a different reason: hierarchy modification. Its effects are examined in the next section.

4 Noninterference Criterion for Hierarchy Modification

Besides introduction, AspectJ allows structure modification of inheritance hierarchies, with the intention to move classes (together with all their subclasses) 'down' the inheritance hierarchy, so that original type relations still hold[5].

4.1 Impact of Changing the Inheritance Hierarchy

The impact of changes in the inheritance relations is demonstrated in figure 3. The changes presented in thie example are due to application of the following simple aspect:

```
aspect O {
    declare parents: D extends G;
}
```

At first sight any client using classes with a modified inheritance hierarchy should still work as any type relation is still correct. However, there are two problems. Let `d` be an object of type `D`:

instanceof: In example of figure 3, class `D` is moved down the inheritance hierarchy by aspect `O`. Any predicate `d instanceof G` now changed value—from *false* to *true*. More generally, the **type** of class `D` has changed. This allows additional up-casts (`(G)d`), which resulted in a `ClassCastException` before. These exceptions might have been caught and so control flow might have changed.

binding interference: Change of inheritance hierarchies might possibly change the method actually executed by a virtual call. Figure 3 gives an example of this situation with method call `d.n()`: Without application of the aspect, `A.n()` is called; with `O` applied, the virtual call evaluates to `G.n()`.

4.2 Hierarchy Modification as Hierarchy Composition

Modification of the inheritance hierarchy can again be viewed as a hierarchy combination. In this case, the hierarchy induced by `declare parents ...extends` statements contains an empty class for any class in the base hierarchy and an inheritance relation $\leq'$ modified by the aspect statement as follows:

Definition 4.1 (Induced Hierarchy) *Let $\mathcal{H} = (\mathcal{C}, \leq)$ be a hierarchy an aspect `A` is applied to. Let `D` be the set*

[5] It is not possible to move classes 'up' in the inheritance hierarchy (AspectJ accepts this declaration without effect).

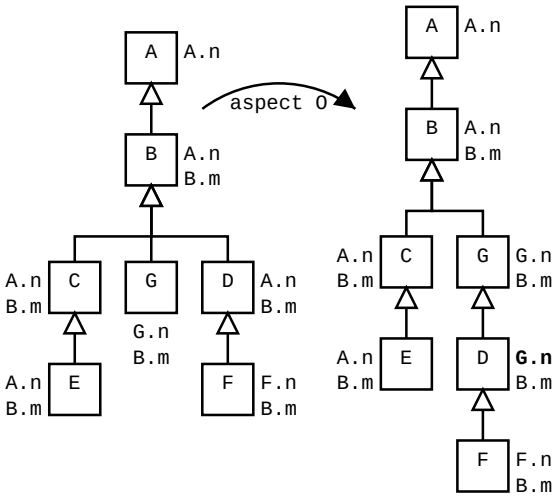


Figure 3: Effects of hierarchy modification.

of tuples derived from `declare parents ... extends` *statements of this aspect. Then* $\leq'$ *is defined as follows:*

$$(\leq') = (\leq \cup D)$$

The hierarchy defined by `A` *is* $\mathcal{H}' = (\mathcal{C}', \leq')$*, where* $\mathcal{C}' = \mathcal{C}$*,* $\forall \mathtt{C} \in \mathcal{C}'$ *:* $members(\mathtt{C}) = \emptyset$*.*

As hierarchy modifications in AspectJ are restricted—it is only allowed to declare that a class now is a subclass of a sibling (or a subclass thereof)[6]in the inheritance tree—the following always holds:

- $(\leq) \subseteq (\leq')$
- $(\mathtt{D},\mathtt{C}) \in (\leq') \Rightarrow (\mathtt{C},\mathtt{D}) \notin (\leq')$ (no conflicts in $\leq'$)

With this properties, the simplified hierarchy combination operator can be applied as no collapsing of equivalence classes due to conflicts is necessary. The resulting hierarchy is given by $\mathcal{H} = (\mathcal{C}, \leq')$.

4.3 Impact of Type Changes

To prove that any client still works as before, the interference criterion of [10] is a necessary but *not sufficient* condition. If a language contains statements for run time type identification (*RTTI*), control flow might change although the above noninterference criterion is met. Java contains such statements with the predicate `instanceof`, which allows to make control flow dependent of the type of an object.

To guarantee that behavior of a client is preserved, all `instanceof` statements have to evaluate to the same value. To calculate the value of such expressions, the type of each reference involved in an `instanceof` predicate has to be known. Approximations with points-to analysis are possible but precise points-to analysis is undecidable. Thus in general only a superset of the type of an object a reference points to can be calculated.

Preservation of behavior can only be guaranteed iff points-to sets of references involved in an instanceof-statement before and after the hierarchy modification evaluate to the *same single type*—a very rigid requirement. In general, when using static analysis, many predicates will evaluate to type-sets with a cardinality bigger than one. In this case, conservative approximation requires to assume that the behavior of the client has changed.

To check the impact of changes to any client of the modified hierarchy the noninterference criterion can be applied if RTTI is excluded. Finding the method calls with changed lookup is easy: Only calls to methods (re)defined in a class between (and including) the new and the former superclass can be influenced, if those methods are not redefined by the affected class itself.

4.4 Detection of Binding Interference due to Hierarchy Modifications

Detection of changes in lookup due to hierarchy modification can be achieved by a simple algorithm. The idea is that any method call has a changed target iff now the virtual call evaluates to a newly assigned superclass. This change in lookup again has to be propagated to any subclass not redefining the affected method.

Calculation of the necessary data can be performed in three steps:

1. Get the set of classes D affected by hierarchy modification.
2. $\forall \mathtt{d} \in D$ calculate the intermediate classes IC between this class and the newly assigned superclass.
3. For any method `m` known in `d`, check if a call now actually evaluates to a class $\mathtt{C} \in IC$. If this is the case, the behavior of the call to `m` possibly changed and `m` has to be added to $\Delta lookup(\mathtt{d})$.

Again, any (transitive) subclass of `d` which does not redefine `m` is affected by the change as well.

[6] If u, v are siblings $\Rightarrow (u,v) \notin (\leq^*) \wedge (v,u) \notin (\leq^*) \wedge \exists w \in \mathcal{C} : (u,w) \in (\leq^*) \wedge (v,w) \in (\leq^*)$, $(\leq^*)$ indicates the transitive closure of $(\leq)$.

5 Impact Analysis of Changes

In [9], a method to compute impact of system modifications on a set of given test drivers has been suggested. It breaks modifications down into atomic changes like *add method* (AM) and *add field* (AF). These atomic changes can be easily derived from the aspects; dependent changes like *change lookup* are calculated by the analysis presented in sections 3 and 4.

With the set of changed lookups at hand, impact analysis can be used to choose a set of test drivers which has to be rerun to check whether the system still works as intended. Only a short summary is presented here, for details refer to [9].

The classes of the hierarchy $\mathcal{H}$ under consideration are now associated with a set of test drivers $\mathcal{T} = \{t_1, \ldots, t_n\}$, where each $t \in \mathcal{T}$ calls a subset of methods defined by classes in $\mathcal{H}$. For each test driver t_i, impact analysis is performed using the call graph of t_i to determine if the test driver (or client) is affected. This is done by checking if t_i calls (maybe transitively) any method with changed lookup.

This check uses calculated information about changed lookups when traversing an edge in the call graph. If the call matches a call in the set of changed lookups $\Delta lookup(\mathtt{C})$ the test driver has to be rerun.

To create the call graph, the type of the calling object at runtime has to be determined for each method call to decide whether the call changed its behavior. This is the case if the object reference may have a type with changed behavior as indicated by the analysis presented above.

Unfortunately, calculation of the exact type at runtime is undecidable. However, points-to analysis can be used to calculate an approximation: the set of possible types for an object reference in the test driver. If a call of any type in this set is contained in the set of methods with changed semantics, conservative approximation demands that the semantics of this call have to be considered as changed. In this case, the test driver containing this method call has to be rerun. The results of this regression tests show if the program still works as intended.

So, the analysis proposed here can provide different results:

- A set of introductions and hierarchy modifications with no effect *on a given set* of test-drivers can be determined. These changes can be incorporated safely into the system as the semantics of the system are not changed.
- For atomic changes modifying system behavior, the subset of test cases which must be rerun can be determined. Impact of these changes can be checked by the results of these regression tests only.
- For the given hierarchy $\mathcal{H}$, impact of static features of aspect application on the semantics of the hierarchy can be determined.

This information can be used by the programmer to avoid unexpected changes and specifically examine results of intended changes.

6 An Example Analysis

To see how the proposed algorithms work, the analysis is applied to an example using all static modification features of AspectJ.

Program 6.1 Combined Aspect Applied to Hierarchy.

```
class Main{
  public static void main(String[] args) {
    print("A: "); A a = new A(); a.n();
    print("B: "); B b = new B(); b.n(); b.m();
    print("C: "); C c = new C(); c.n(); c.m();
    print("D: "); D d = new D(); d.n(); d.m();
    print("E: "); E e = new E(); e.n(); e.m();
    print("F: "); F f = new F(); f.n(); f.m();
    print("G: "); G g = new G(); g.n(); g.m();
    println();
  }
}

aspect MNO {
  // declare parent extends / implements
  declare parents: D extends G;
  declare parents: C implements I;
  declare parents: D implements I;

  // introductions
  public void I.y() { print("I.y()"); }
  void B.n() { print("B.n()"); }

  public static void main(String[] args) {
    print("A: "); A a = new A(); a.n();
    print("B: "); B b = new B(); b.n(); b.m();
    print("C: "); C c = new C(); c.n();
      c.m(); c.x(); c.y();
    print("D: "); D d = new D(); d.n();
      d.m(); d.x(); d.y();
    print("E: "); E e = new E(); e.n();
      e.m(); e.x(); e.y();
    print("F: "); F f = new F(); f.n();
      f.m(); f.x(); f.y();
    print("G: "); G g = new G(); g.n(); g.m();
    println();
  }
}
```

6.1 The System to Analyze

As a starting point, the class hierarchy defined by program 1.1 is given, together with aspect `MNO`, which combines

the effects of former aspects. It introduces a new method `n` to class `B`, changes the inheritance relation (`declare parents:D extends G`) and declares that classes `C` and `D` implement interface `I`. Methods are inserted to class interface `I`. Additionally, the aspect defines an own `main`-method which is necessary to test the results of interface declaration. Effects of aspect application are a changed structure as well as a changed lookup for some methods.

The classes of this example are quite simple: All methods only print their name and the class they are defined in, but this setting is already sufficient to show how the aspect affects the existing system. Figure 4 presents the output of the system. The figure contains three sections. The output of the original system without application of the aspect is marked with '(a)'. The effects of binding interference are visible in section '(b)', which shows the output of the original main method with aspect `MNO` applied to the system. The set of known methods is identical, but the dispatch has changed for classes `B`, `C`, `E`, and `D`. The first three classes are affected by the introduction of `n` to `B`, class `D` by the change of the hierarchy.

All effects of the aspect are visible in section '(c)', where the effects of the `declare parents: ...implements I` statements become visible. No 'old' base system code uses this effects as in the original hierarchy `C` and `D` did not implement `I`. So, for `C`, `D` and all their subclasses, methods `x` and `y` can be called. For class `C` only an implementation of `x` is provided, for `y` the default implementation of `I` is used—as is visible in the output.

6.2 Applying the Proposed Analysis

The analysis revealing classes only using the default implementation of an interface, like e.g. `E` does, is quite simple and not considered. The example concentrates on changes in lookup. Changes due to introduction can be found by applying algorithm 3.1. For the example hierarchy, table 5 summarizes the gathered information. The example application of the algorithm traverses all classes of a given hierarchy according to a bfs-order determined by the structure of the class hierarchy *after* applying hierarchy modifications of the aspect.

Step 7 is interesting as at this position the changed lookup results from the change of hierarchy structure, *not* from introduction (the father of `D` now is `G`, which has an own definition of method `m`; so introduction of `m` to `B` has no longer any effect on `D`). When calculating changes in lookup, these effects must be considered. The algorithm reproduces the results visible when comparing sections (a) and (b) of figure 4.

6.3 Using these Results—Impact Analysis

The calculated information about changed lookups can be used for impact analysis to determine whether a given test driver has to be rerun. For illustration consider the set of (quite simple) test drivers associated with the example hierarchy presented in program 6.3.

To decide if control flow has been changed by introductions, the call graph has to be constructed. Note that points-to analysis is necessary as the types of caller and callee of a virtual call has to be identified or at least re-

```
a) : original system
javac demo.java
java Main
A: A.n()
B: A.n()    B.m()
C: A.n()    B.m()
D: A.n()    B.m()
E: A.n()    B.m()
F: F.n()    B.m()
G: G.n()    B.m()

b) : changes due to dynamic
     interference
ajc demo.java demo.aj
java Main
A: A.n()
B: B.n()    B.m()
C: B.n()    B.m()
D: G.n()    B.m()
E: B.n()    B.m()
F: F.n()    B.m()
G: G.n()    B.m()

c) : including introduction
     to interface
ajc demo.java demo.aj
java M
A: A.n()
B: B.n()    B.m()
C: B.n()    B.m()   C.x()   I.y()
D: G.n()    B.m()   D.x()   D.y()
E: B.n()    B.m()   C.x()   I.y()
F: F.n()    B.m()   D.x()   D.y()
G: G.n()    B.m()
```

Figure 4: Example: Produced output.

Step	v	declared methods	members(v)	Intr(v)	Δ*lookup*(v)	queue
1	-	-	-	-	-	{A}
2	A	n	n	-	-	{B}
3	B	n, m	m	n	B.n	{C, G}
4	C	n, m	-	-	B.n	{G, E}
5	G	n, m	n	-	-	{E, D}
6	E	n, m	-	-	B.n	{D}
7	D	n, m	-	-	**G.n**	{F}
8	F	n, m	n	-	-	∅

Figure 5: Results produced by the algorithm (x, y omitted).

Program 6.2 Test Drivers for the Example Hierarchy.

```
class T1 {
    public static void main(String[] args) {
        F f = new F();
        f.m();   // calls B.m()
        f.n();   // calls F.n()
    }
}
class T2 {
    public static void main(String[] args) {
        B b = new B();
        b.n();   // calls B.n(), changed lookup
    }
}
class T3 {
    public static void main(String[] args) {
        G d;
        if (args.length != 0) d = new D();
        else d = new G();
        d.n();   // calls G.n(), caller: D or G
    }
}
```

stricted to a as-small-as-possible type set.

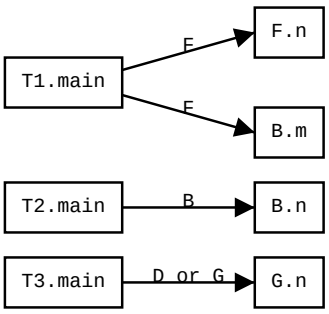


Figure 6: Call Graph of Simple Test drivers

To get a first impression how impact analysis works, consider test drivers T1 to T3 and their call graph. The edge labels of figure 6.3 indicate the type of the calling object. To evaluate the impact of an aspect using the call graph, we need the results of table 5.

Test driver T1 is obviously unaffected by changes due to aspect application as no lookup for an F-object changed. Test driver T2 calls n from a B-object. This lookup has changed from A.n() to B.n() due to introduction of method B.n. This test driver has to be rerun.

Test driver T3 is a little more complex as here the type of the calling object is statically unknown. Possible types are D and G. For a G-object, semantics would be preserved, but for a D-object, the call would evaluate to G.n() and not to A.n as in the original hierarchy. Conservative approximation demands to rerun test driver T3. Certainly this is a simple example, but there is no restriction to apply this analysis to real-world call graphs as it can be done by performing this simple check for every edge.

7 Preliminary Implementation and Future Work

A prototype of the analysis presented in sections 2 to 4 has been implemented and produces reasonable results for programs written in a subset of AspectJ, including the example of section 6 presented in this paper.

However, implementation of the impact analysis and extension of the set of analyzable programs still has to be done. A point of interest is the handling of Java import-statements as imported classes are necessary information to built up the hierarchy $\mathcal{H}$. For these classes, source code might not be available. To solve this problem, it is planned to reconstruct class information out of Java byte code using the BCEL API.

Evaluation of occurence of binding interference in 'real life' AspectJ programs is necessary to determine if this problem is actually relevant for AspectJ programmers. However, even if binding interference is not very frequent, the AspectJ compiler should issue a warning.

8 Conclusion and Related Work

This paper pointed out the problem of binding interference emerging from usage of the AspectJ features introduction and hierarchy modification. Definitions are given how AspectJ introduction and hierarchy modification can be interpreted as hierarchy combinations. With this definitions at hand, the noninterference criterion of [10] and the impact analysis of [9] can be applied to check if clients of the hierarchy under consideration possibly changed behavior. This analysis can help AspectJ programmers to examine the impact of aspects before application and avoids subtle flaws in their programs.

To improve separation of concerns, several different approaches besides aspect oriented programming have been suggested. Aksit et al. proposed composition filters [2, 1] to route incoing and outgoing messages through a filter queue, thus enabling similar functionality. Batory et al. proposed layered designs [4, 3].

Especially relevant for the approach presented here is [8]. Ossher and Tarr proposed multi-dimensional separation of concerns, leading to a separate implementation of different features and a composition of the resulting hierarchies according to user defined composition rules. Semantics of these compositions are a research topic addressed in [10].

Besides [10], very little work of program analysis for AOSD approaches is known, although impact analysis of [9] could be used for AOSD software as well.

Acknowledgements

Thanks to Silvia Breu for her valuable feedback.

References

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.

[2] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[3] D. Batory and Y. Smaragdakis. Building product-lines with mixin layers, 1999.

[4] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[5] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[6] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. 1994.

[7] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. *Lecture Notes in Computer Science*, 1445:355–382, 1998.

[8] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach, 2000. Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.

[9] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*, pages 46–53, 2001.

[10] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In *ECOOP*, page 562ff, 2002.

Compositional Reasoning About Aspects Using Alternating-time Logic

Benet Devereux
Department of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, Canada
M5S 3G4
benet@cs.toronto.edu

ABSTRACT

Aspect-oriented programming offers greater modularity to the programmer, but it is not yet clear how best to reason about an aspect-oriented program in a modular way. We propose a translation of aspect-oriented programs into alternating transition systems (ATSs), which provide a decidable formal specification language, alternating-time logic, that allows us to specify which component is responsible for enforcing certain properties. We develop rules for compositional reasoning using these translations.

1. INTRODUCTION

It is an observed problem [5] with aspect-oriented programming that, while aspects do provide additional modularity of development, it is not yet well established how to reason about aspect-based programs in a modular way. Such compositional reasoning allows aspects to be not only designed in isolation, but also formally verified. Such formal verification is useful for tractably proving correctness of large programs, since the task can be decomposed, and also for promoting re-use of aspects.

Compositional reasoning in concurrent programming is a well-understood (though difficult) problem. Modules of concurrent programs can be verified separately through *assume-guarantee* reasoning [7]. The question we pose at the outset is: how much of this knowledge can be reused for compositional reasoning about aspects?

To begin answering this question, we propose a semantics of aspect-oriented programs based on *alternating transition systems* (ATSs), a variety of state-machine model which explicitly represents how multiple components work together to change the system's state, each only having partial control. In our model, we treat aspects as concurrent components which have the authority, at certain points, to take control and modify program state, possibly returning at a different point. The possible points of return are constrained, but allow for an aspect either to return control where it took it, or to skip a statement. This semantics is a direct translation from code into a low-level state machine model, but it should be the same as a code-weaver-based semantics [8], where the aspect code is woven into the source code, and then translated; with the additional information of the allocation of responsibilities to components (system and aspect). We are not proposing any new constructs for aspect languages, but rather an approach to automated compositional analysis of existing languages.

Compositional reasoning for concurrent systems often proceeds as follows: there are two communicating components, P and Q. We show first that P placed in composition with a suitable abstraction of Q is correct; then, that Q in composition with an abstraction of P is also correct. In this way, we avoid constructing the full state-space of $P \parallel Q$, which may not be possible in the available memory; we pay the cost of having to construct and verify the abstractions.

We argue that this model is general enough to allow us to represent complicated interactions of components to determine state transitions, and yet remain amenable to the existing methods of analysis of alternating transition systems, such as model-checking [1] and refinement checking [2]; as well as any new techniques which aspect verification may make necessary.

The contribution of this paper is a translation from a simple aspect language into the ATS formalism, which allows assume-guarantee reasoning, and a discussion of how this translation may be used to show that an aspect modifies a program correctly. The aspect language is similar to the fragment of AspectJ [8] which only deals with advice to running code, and not with modifications of the class hierarchy. We explain the proposed technique on an example, giving definitions as needed, and suggest two compositional proof rules for analysis of aspect-oriented programs.

2. EXAMPLE

As an example aspect, we take precondition-checking. A program module makes use of a `Point` class; some class

(a)

```
p0:  while (!button);
p1:  pt.movePoint(x,y)
p2:  goto p0
```

(b)

```
pointcut mp(x, y):
 call (Point.movePoint)
   && args(x,y)
-------------------------
      over mp(x,y):
a1:   if (x>=0, y>=0)
a2:     continue
a3:   else
a4:     skipover
```

Figure 1: (a): code using `movePoint`, (b): code for the precondition-checking aspect.

methods have preconditions, and if they are called without the precondition holding, their behavior is undefined. We take `movePoint` as one such method, following [5]; its precondition is that the coordinates given are non-negative. An aspect is defined which inserts checks of this precondition before any call to `movePoint`. The aspect skips over the call if the precondition is not met. It is assumed that `movePoint` is provably correct: that is, if it is invoked under the right conditions, it terminates with its postconditions satisfied.

The program is shown in Figure 1(a). Its user interface contains a canvas, two numeric text fields in which the user can fill in x and y coordinates, and a button to move an image to the specified location on the canvas. The program only ever reads the text fields, it does not write to them; and so their contents are determined by the environment alone. All other state is determined by the program. For this aspect to be correct, the combination of the aspect with the program must have the following properties:

- if the program calls `movePoint` with the proper precondition, the postcondition will be satisfied at the first program point after the invocation of `movePoint`; this was true before aspect imposition, and should *still* be true after the aspect is added
- the aspect should prevent `movePoint` from being called with bad parameters

Additionally, the problem of aspect interaction should also be addressed: there may be other aspects which run before, or after, or even during the precondition-guarantee aspect. The problem of formally representing the assumptions on *other* aspects under which the aspect continues to behave as specified seems similar to the problem of representing the system assumptions under which the aspect of interest behaves properly, but considerably more subtle. Though interaction of aspects is an important consideration, this preliminary work does not as yet deal with it.

3. ALTERNATING TRANSITION SYSTEM SEMANTICS

In this section, we give a sketch of the proposed semantics, using the `movePoint` example to illustrate it. The semantics is in most respects a standard operational semantics for an imperative language [9], only enriched with information about how individual agents (the system, the environment, and the aspect) control the state transitions. We refer the reader to the stated references for a formal treatment, and illustrate all definitions using the example.

Informally, an alternating transition system is a state machine where multiple agents each have partial control over the transition relation. Thus, in a single state, each agent may not be able to definitively choose a successor, but rather a set of possible successors: the actual state chosen from that set is contingent upon how the other agents choose. Thus, in a sense, the agents play a game to control the behaviour of the system. We say that an agent has a *capability* if it has a strategy to keep the behaviour within a certain set where every possible execution has some desired property; these capabilities will be expressed in *alternating-time logic*, to be described more in Section 4.

DEFINITION 3.1 (ALTERNATING TRANSITION SYSTEMS). *An alternating transition system is a tuple* $A = (\Omega, S, P, L, R)$ *where:*

- Ω *is a finite set of agents;*
- S *is a set of states;*
- P *is a set of atomic propositions;*
- $L : S \to 2^P$ *is a function labelling states with sets of atomic propositions;*
- $R : S \times \Omega \to 2^{2^S}$ *is the transition relation;*

At each state s*, every agent* $a \in \Omega$ *chooses one set of possible successors* $T_a \in R(s, a)$*; the intersection* $\bigcap_{a \in \Omega} T_a$ *must be a singleton, which is the chosen successor.*

We look again at the program-aspect composition of Figure 1. At any point in execution, the environment has partial control over the evolution of system state: it may change the text fields and the button, but update of the program counter and the canvas is up to the system. Thus, when the environment moves at a state, it chooses a set of successors: for instance, it can choose the set of all states where x is 5 and y is 3 as successors, but it cannot choose one uniquely, because the system has control over the program counter and the canvas. Symmetrically, the system chooses a set determined by its choices; the intersection of these two (along with the aspect's actions) produces a unique successor.

To represent ATSs visually, we use state diagrams annotated with decision nodes between states. The states are labelled circles, and the decision nodes are small squares with the name of the agent making the decision. Note that the decision points between states are arranged in a sequence for the sake of visual clarity, but that in fact all decisions are made *simultaneously* by the agents, none having knowledge of what the other agents are choosing. A fragment of the ATS translation for the system/aspect composition of Figure 1 is shown in Figure 3. Solid circles are states where the program is active, dashed circles are states where the aspect is active. A starred transition between two states indicates that it is reachable by collaboration of all three agents.

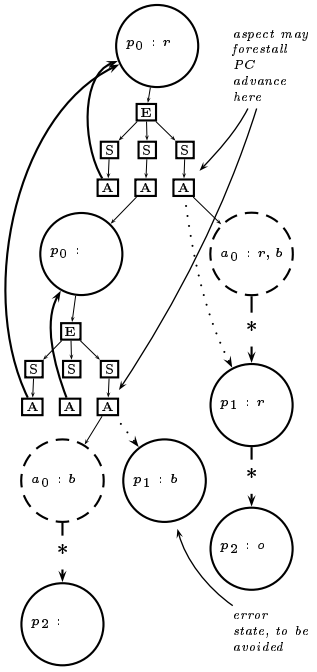


Figure 2: Fragment of the ATS translation

The states are labelled with the program point (either in the program or the aspect: while one is active, the other is assumed to be suspended), followed by a list of the propositional variables holding in that state. These are abstractions of the concrete program state, defined as follows:

Abstract Variable	Description	Definition
r	Precondition holds	$(\mathtt{x} \geq 0) \wedge (\mathtt{y} \geq 0)$
b	Button pressed	`button=true`
o	Postcondition holds	$(\mathtt{pt.x}{=}\mathtt{x}) \wedge (\mathtt{pt.y}{=}\mathtt{y})$

At the root, the program is at p_0 with a valid input. The environment can choose to either enter valid input again, enter invalid input, or press the button. If the button is pressed, the system *attempts* to advance the program counter to p_1; however, the aspect is able to interrupts it, moving to a_0. Since the input is still valid, the aspect executes `continue`, which returns control to the system at p_1, allowing it to execute `movePoint`; since the aspect has no power to interrupt once the call has actually been reached, the system is able to reach p_2 with the postcondition satisfied.

However, in the middle subtree, we see what happens when the environment invalidates the input, and then presses the button; the aspect takes control, but since the input is not valid it executes `skipover`, returning control at p_2 without allowing the program to execute `movePoint`. From the system's perspective, the aspect could have chosen to allow this transition: this possibility is indicated by a dotted transition. The aspect thus has the capability of preventing $p_1 \wedge b$ being true while r is false.

This translation is simplified, and does not take into account the possibility of other aspects taking control both before and after the guarantee aspect. Since the ATSs are compiled from code, the control-flow of the code is represented using a program counter variable; however, if ATSs are used in high-level modelling (aspect-oriented *design*), then this can be dispensed with. Formal support for such design is another goal of this work. It is not necessary for the model to be finite-state, as it is here, but finiteness is a sufficient condition for model-checking to be decidable.

4. REASONING WITH ASPECTS

In this section we discuss formal specification in alternating-time logic, and compositional reasoning about ATS translations of aspect-oriented programs.

4.1 Specification

The aspect we describe is introduced in order to guarantee a simple temporal safety property: it is *never* possible to reach a state which is a call to `movePoint` (p_1 in Figure 1) while the precondition is violated. This is a property which should hold in all executions of the system, regardless of nondeterministic choices.

A classical state machine model allows us to quantify over possible executions using temporal logic [4]. Is there an execution where the point is never moved? Certainly, the user need never press the button. Does the point pass through (2,3) in every execution? No: there are sequences of inputs which prevent this. An ATS model encodes not only this information, but also *which agents* co-operate to create an execution. In this case, both of the executions we have discussed are enforced by the environment alone: it is able to prevent the point from ever moving, while the system must move the point whenever the button is pressed.

We say that the environment has a *strategy* – a way to resolve the choices available to it – which prevents the point from moving. At any given time step in the program, the environment never has control over the entire program state: only the two fields and the button. How the rest of the state evolves is up to the system. However, with the system specified as it is, this limited control suffices for the environment to keep the point still. It is often desirable to show that environment properties are preserved [3].

DEFINITION 4.1 (STRATEGIES AND PATHS). *Given an ATS* (Ω, S, P, L, R), *a* strategy *for agent* $a \in \Omega$ *is a map* $f_a : S^+ \to 2^S$, *such that for all* $w \in S^*$ *and* $s \in S$, $f_a(ws) \in R(s, a)$.

The choice of a strategy by an agent constrains the possible executions. Given a strategy f, $\pi_f \in 2^{S^\omega}$ *is the set of all infinite paths which the remaining agents are able to choose.*

Alternating-time logic is defined with respect to strategies and the paths they determine. For example, given a formula φ:

- $s \models \langle a \rangle \mathbf{X}\varphi$ if a has a strategy f such that for all paths $st \ldots$ in π_f, $t \models \varphi$
- $s \models \langle a \rangle \mathbf{F}\varphi$ if a has a strategy f such that for all paths $st_0t_1 \ldots$ in π_f, there is an i such that $t_i \models \varphi$

The purpose of adding the precondition-guarantee aspect is to impose a new property which did not hold before: that the environment *cannot* cause `movePoint` to be called without the precondition holding. It may still press the button, but the aspect alters the system's response, skipping over the call to `movePoint` if the precondition is violated[1]. At the same time, the aspect should not prevent `movePoint` from being called if the precondition does hold. An aspect's specification, then, as has been observed [12], is twofold: it has new properties that it must guarantee, and old properties that it must preserve. Both are dependent upon being used according to a contract.

We are transforming the system's code to meet the specification. To encode at the state-machine level the fact that we are doing this with an *aspect* rather than a *code patch* – that is, for modularity and maintanability rather than correctness – we represent this modification as the actions of another agent (the aspect) which is able to seize control of the system's program-counter at certain points, execute some code, and then return control.

4.2 Compositional Reasoning

How, then, do we show that the aspect meets both parts of its specification? The simplest approach to verifying a program is to create a formal model of the entire program, and prove it meets its specification. In general this is not feasible for two reasons: first, a state-machine translation grows exponentially in size with the number of variables, and rapidly becomes far too large for available memory; second, the reasoning task becomes forbiddingly complicated and cannot be decomposed or distributed in any way.

So it has been held that reasoning should be modular: and the most straightforward way to make reasoning modular is to follow the modular structure of the program. That is, given modules X and Y, which interact in a given way, we prove that if X behaves according to specification, so will Y; and if Y behaves according to specification, so will X. This is *compositional*, *modular* or *assume-guarantee* [10, 7] reasoning.

Since aspects form an alternative decomposition, it seems desirable to do reasoning that follows the aspect structure of a program. Not only would this facilitate reasoning about aspect-oriented programs, it would also promote more flexibile compositional reasoning in general, providing alternative decompositions which might be more amenable to proof. Our goal,then, is to develop compositional proof rules for aspect-oriented programs. The remainder of this section presents two such proposed rules.

4.2.1 Imposition Rule

Let M be the module, and F (for 'feature') be the aspect; they are attached using a binding c which identifies join-points in M with F's pointcuts (in our example, the calls to `movePoint` make up the pointcut `MP`). Consider first one half of the compositional reasoning task: assuming M is correct, and c is the correct binding, we wish to show that F must have the desired effect. That is, we wish to analyze F in isolation. However, since it is a controller which reacts to its environment – the program-counter changes of a module – it is, by itself, an open system, and needs to be closed with some model of its environment.

[1] If the `Point` class were not a black box, an aspect could instead insert precondition-checking code at the beginning of `movePoint`, changing the class specification.

p'_0: `goto` p'_0 □ `goto` p'_1
p'_1: `beforeJP`
p'_2: `pt.movePoint(x,y)`
p'_3: `afterJP`
p'_4: `goto` p'_0

Figure 3: Abstraction A_M of the program M: □ indicates a nondeterministic choice

This model should be considerably simpler than M itself, since most of the behavior of M is of no interest to the aspect – only the entering and leaving of join-points. Such an abstraction A_M is shown in Figure 3; it is not much simpler than the program, but `button` has been abstracted away as irrelevant to the aspect's correctness. All that matters is that the abstraction calls `movePoint` some number of times with varying values of x and y.

Given this abstraction, we construct its composition $A_C(A_M, F)$; note that the composition C must be abstracted as well – since the pointcuts remain the same, but the join-points in a pointcut are different in M and A_M. The important property of this composition is that it preserves all the capabilities of F: the system is abstracted, but F's behaviors are neither expanded nor constrained. More formally, we say that $A_C(A_M, F)$ is an *S-abstraction* of $C(M, F)F$, which we write:

$$A_C(A_M, F) \leq_S C(M, F)$$

and that the aspect has the same capabilities (is A-equivalent):

$$A_C(A_M, F) \leq_S C(M, AF)$$

We refer to this composition as the *abstract aspect*, to follow the terminology of AspectJ; but note that it is not the aspect which was been abstracted!

Recall that the new property we wish to demonstrate is that the environment can be prevented from calling `movePoint` when the precondition is unsatisfied; that is, that the system and aspect together can prevent the call whatever the environment does. If we can demonstrate this for the abstract aspect, and show that it the abstract aspect has all of the system and aspect capabilities of the full composition, we can conclude that the full composition meets its specification.

Thus we state our first compositional reasoning rule, which we'll call the Imposition Rule, used to prove that an aspect guarantees a capability $\langle S, A\rangle\varphi$ of S and A:

$$\frac{A_C(A_M, F) \models \langle S, A\rangle\varphi \qquad A_C(A_M, F) \leq_{\{S,A\}} C(M, F)}{C(M, F) \models \langle S, A\rangle\varphi}$$

That is, if the abstract aspect meets the specification *and* it is an S, A-abstraction of the full composition with respect to S, then the full composition also meets the specification. By construction, the abstract aspect is A-equivalent to the full

composition, and an S-abstraction, so the remaining obligations are to show that it is an S, A-abstraction (which is not necessarily implied by being both an S and A abstraction [2]), and that $\langle S, A\rangle\varphi$ holds in the abstract aspect.

To sum up, the Imposition Rule enables us to verify an abstract aspect, and show that the verification holds in the full composition by proving the abstraction relation between the abstract aspect and the full composition.

4.2.2 Preservation Rule

Of course, we also wish to show that the aspect preserves some of the existing properties of the base program. We take a similar approach: composing the module with an abstraction of the aspect, checking the desired property on this composition (which we dub the *program-in-context*), and using the abstraction relation to conclude that the property holds in the full composition, without actually needing to construct and analyze it.

The intuition behind the program-in-context is that it is the program in a very general aspect-oriented runtime environment, with the join-points fixed. At any join-point, the aspect may choose to interrupt, take control, and return to any allowable program point, possibly with some changes to program state. The program-in-context is *not* equivalent to the unmodified program: some capabilities of the system are likely to be broken by the imposition of the aspect.

The program-in-context $C(M, A_F)$ for our example is a composition of the base program with the following abstraction:

```
over mp(x,y) {
  continue [] skipover;
}
```

In other words, it is an A-abstraction of the full composition. Note that this embodies two additional assumptions: the aspect does not modify any state readable to the program, and that it only inserts before and over advice – never after. These assumptions are necessary for the program-in-context to preserve the desired property.

Part of the ATS for the program-in-context appears in Figure 4. In the unmodified system, whenever the PC is at line p_1 with the precondition satisfied, the postcondition is satisfied at the next step. This system capability is stated in ATL as:

$$\langle S\rangle p_1 \wedge r \Rightarrow \mathbf{X}o$$

We must check the program-in-context to see that this system capability is also preserved there. Looking at 4, we see that although the aspect can prevent the system getting to p_1, once it is there it can execute `movePoint` and satisfy the precondition.

So we state the Preservation Rule: if system capability $\langle S\rangle\varphi$ holds in the program-in-context $M \parallel_c A_F$, and the program-in-context is an S-abstraction of the full composition, then we can conclude that $\langle S\rangle\varphi$ holds in the full composition:

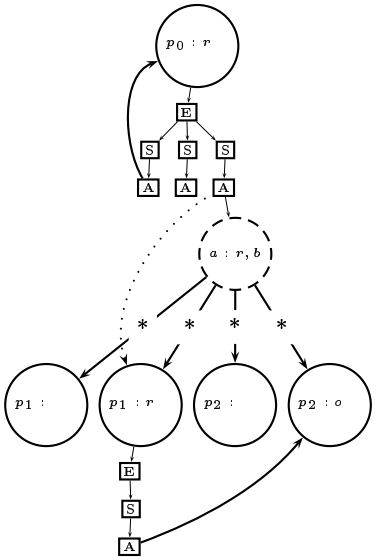


Figure 4: Fragment of ATS for program-in-context

$$\frac{C(M, A_F) \models \langle S\rangle\varphi \qquad C(M, A_F) \leq_S C(M, F)}{C(M, F) \models \langle S\rangle\varphi}$$

5. RELATED WORK

Researchers studying the *feature interaction problem* have encountered similar issues, since features often have similarly cross-cutting effects. Ryan et al. [11] have developed a *feature construct* for state-based modeling languages; their features are expressed at the state-machine level of abstraction rather than the program-code level. They have also [3] used an alternating transition system framework to prove that imposition of a feature maintains desirable properties of a system, even though it is a non-monotonic composition in general. The difference is that in their formalization, each module is represented by an agent; this allows for reasoning about capabilities of agents before and after feature imposition, but does not allow for the distinction of base and feature as separate agents, and thus does not lead to the kind of modular reasoning we wish to do.

Clifton and Leavens [5] address modular reasoning with aspects, and suggest two types of explicit contracts: observers and assistants. An observer is an aspect which does not change the existing specification of any module it is attached to; it only ever changes its own state. That is to say, the capabilities of the module by itself are identical to the capabilities of the composition of module and observer; only the aspect gains new capabilities. Assistants may modify system capability.

Fisler and Krishnamurti et al. [6] also use a state-based model of feature composition, and aim towards compositional reasoning. They have an effective decision procedure for proving that a feature preserves and guarantees properties without needing to construct the full state space; however, they do not consider the situation where a feature can disable a transition of the system it modifies.

We are not aware of any work which formalizes, in a general way, the weaving of aspects in general, dealing with cases like weaving of class hierarchies or data-flow graphs that are not handled by the proposed method.

6. CONCLUSION AND FUTURE WORK

We have discussed a proposed approach to modular reasoning with aspects. This approach is a variety of assume-guarantee reasoning, using an alternating transition system model, with alternating temporal logic [1] as a specification language. We have illustrated the fundamentals of this approach on an example, demonstrating how the proposed formalism enables compositional reasoning.

This work is in its early stages, and there is much to be done. Proving the necessary abstraction relations is the difficult part of the approach, and this must be shown to be scalable and relatively automatable. Further, the decidability of the analysis depends on a finite-state model, and so abstractions of the state-space are necessary to make a general program finite-state; any interactions between these abstractions and those of the proposed compositional rules must also be considered.

7. ACKNOWLEDGMENTS

I thank Marsha Chechik for extensive assistance with the ideas and presentation and Steve Easterbrook for fruitful discussions. Shmuel Katz started me thinking about the problem, and directed me to the literature on superpositions. The reviewers' comments helped to improve and clarify the material.

8. REFERENCES

[1] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. "Alternating-time Temporal Logic". In *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages pp.100–109, 1997.

[2] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. "Alternating Refinement Relations". In *Proceedings of CONCUR '98*, pages 163–178, 1998.

[3] F. Cassez, M. D. Ryan, and P.-Y. Schobbens. "Proving Feature Non-interaction with Alternating-Time Temporal Logic". In S. Gilmore and M.D. Ryan, editors, *Language Constructs for Describing Features*. Springer-Verlag, 2001.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] C. Clifton and G.T. Leavens. "Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning". In *Proceedings of FOAL 2002 (Foundations of Aspect-oriented Languages)*, 2002.

[6] K. Fisler and S. Krishnamurthi. "Modular Verification of Collaboration-Based Software Designs". In *Proceedings of FSE '01*, September 2001.

[7] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. "You Assume, We Guarantee: Methodology and Case Studies". In *Proceedings of CAV '98*, pages 440–451, 1998.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ". *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[9] John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[10] Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. "Assume-Guarantee Model Checking of Software: A Comparative Case Study". In *Proceedings of the 1999 SPIN Workshop on Software Model Checking*, 1999.

[11] M. Plath and M. Ryan. "Feature Integration using a Feature Construct". *Science of Computer Programming*, 41(1):53–84, 2001.

[12] M. Sihman and S.Katz. "A Calculus of Superimpositions for Distributed Systems". In *Procedings of AOSD 2002*, 2002.

Model Checking Applications of Aspects and Superimpositions

Marcelo Sihman and Shmuel Katz
Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
{sihman, katz}@cs.technion.ac.il

ABSTRACT

The model checking of applications of aspects is explained, by showing the stages and proof obligations when a collection of generic aspects (called a superimposition) is combined with a basic program. We assume that both the basic program and the collection of aspects have their own specifications. The Bandera tool for Java programs is used to generate input for model checkers, although any similar tool could be employed. New *verification aspects* and superimpositions are defined to modularize the proofs, and separate the proof-related code from the program and the aspects. This allows generating and activating a series of model checking tasks automatically each time a superimposition is applied to a basic program, achieving *superimposition validation*. A case study that monitors and checks an underlying bounded buffer program is presented.

1. INTRODUCTION

Aspects help to isolate cross-cutting concerns in programs and designs. Many researchers have been working on programming and design techniques, software evolution and other implications of AOP. However, little work has been done about formal verification of aspects. In this paper, we show in detail how to verify the combination of collections of aspects over basic programs, using model checking techniques. The use of special aspects for verification is also presented, providing yet another natural application of aspect-oriented software design.

We introduce this approach as a new feature of SuperJ, an AOP construct that we have proposed in [15]. SuperJ provides language support for defining collections of parameterized aspects independently of any basic program, where such a collection is called a *superimposition*. A superimposition is a module describing an algorithm that may be applied to different underlying basic programs. A brief introduction to SuperJ is presented in Section 2.

In this paper we consider how model checking of software can be used in the formal verification of combinations of superimpositions and basic programs. Model checking has the advantages of automatic verification (in that difficult invariants do not need to be supplied, as is the case in formal verification based on theorem proving), yet provides full verification, as long as any data abstractions preserve the properties being checked. Additionally, it has proven popular with verification of hardware designs mainly because it provides counter-examples when the property of interest does not hold.

We have chosen Bandera [5] as the prototype generator of input to model checkers such as SMV or Java Pathfinder, and thus use Bandera's specification notation BSL for describing temporal properties to be model checked. A brief introduction to Bandera is given in Section 3.

When binding a collection of aspects (a superimposition) to a basic program (a collection of basic classes), we need to bind each relevant class of the basic program to a generic aspect (of the superimposition), where basic classes may be left unbound to any generic aspect if they do not play a role in the superimposed algorithm.

In a superimposition, we specify *assumptions* about the basic programs and parameters to be bound and *desired results* that must be true in the augmented program, where an *augmented program* is the result after binding a superimposition to a basic program. We assume here that the result of such binding and instantiation (often called *weaving*) is a Java program in itself, rather than, for example, Java byte-code. (The implications of this assumption for our implementation are considered later.) A *superimposition is correct* if, when the aspects in it are woven into a basic program that satisfies the superimposition's assumptions, the augmented program satisfies the desired results and does not violate the original specification of the basic program.

As will be shown, in Bandera, code is added to the program to be model checked in order to define functions, predicates, control locations, and assertions used only for the model checking. We take advantage of the superimposition construct to define *verification aspects* that are used to separate these additions from the code of the programs. All of the verification aspects concerning the assumptions are grouped into an *assumptions superimposition*, and similarly, those related to the results are in a *results superimposition*. The superimpositions and basic programs of the application under consideration can thus be kept free of verification augmentations. This is possible in SuperJ because it supports weaving multiple superimpositions over a basic program, so both the application superimpositions and those needed for verification can be combined before applying Bandera to

generate input for a model checker.

There are several possibilities for using the approach seen here to check superimpositions and their combination with basic programs. These vary according to the modularity in the proof itself, and whether we wish to prove the superimposition correct independently from any specific basic program. In the case of model checking, this may be done by writing a suitable abstraction of a basic program that respects the superimposition requirements, along with an inductive proof. However, we claim here that a more practical alternative is to use the verification superimpositions to set up the automatic generation and activation of four model checking tasks each time a superimposition is applied to a basic program. This procedure, explained and justified later in the paper, is known as *superimposition validation.*

2. SUPERJ

SuperJ introduces constructs that extend the expressiveness and modularity of AOP. Among the new facilities in SuperJ are grouping related aspects into a superimposition, providing specifications, extending parameterization of aspects, dealing with interaction and interference among aspects, and combining superimpositions to obtain new superimpositions. The new superimposition construct comes from the merging of ideas from two distinct research subjects: 'classic' superimposition and AOP.

Well-known examples of 'classic' superimpositions are termination and deadlock detection, monitoring or debugging, adding scheduling restrictions, imposing mutual exclusion, or bounding the possible values of variables that were unbounded in the basic program. These examples have in common the need to add or *superimpose* an algorithm over a basic program. Numerous suggestions ([1, 2, 3, 4, 10, 11]) have been made for a syntax that allows augmenting program units, such as processes. A brief survey about several proposals of a language construct for superimpositions may be found in [16].

In SuperJ, a superimposition is defined as a collection of generic parameterized aspects and singleton concrete classes. A generic aspect has no built-in connection with any program unit of any basic program, and in contrast to usual aspects, a generic aspect contains an extensive parameter list that allows binding it to any appropriate basic class. The singleton concrete classes define unique objects that must be instantiated in an augmented system, where these unique objects interact with the generic aspects. We have defined an AspectJ-based implementation for SuperJ, and have written a preprocessor that translates SuperJ to pure AspectJ code. The same preprocessor is responsible for several tasks, such as: binding arguments from the basic program (classes, methods, etc.) to the parameters of the generic aspects, and applying a superimposition to a basic program, generating concrete aspects from generic aspects and then weaving them to the basic classes.

3. BANDERA

The Bandera Tool Set [5], as defined by its authors, is an integrated collection of program analysis, transformation, and visualization components designated to allow experimentation with model-checking properties of Java source code. Bandera takes as input an augmented Java source code and a program specification written in Bandera's temporal Specification Language (BSL), and produces a program model and a specification as input to one of four model-checking applications: SMV [12], Spin [8], dSpin [9] and Java PathFinder [7]. This 'input' generated by Bandera is written in the model and specification languages of one of the four model-checking applications mentioned. Then Bandera uses the model-checking application to prove whether the model satisfies the required specification (the Java program satisfies the BSL specification). If the specification is not satisfied, then a counter-example is returned, as is common in model-checking tools. Moreover, Bandera shows the problematic execution path, which does not satisfy the required specification, directly in the Java code.

Bandera deals with the state explosion problem, as the program state model must be finite, by providing data abstraction and program slicing features when customizing the model. These features help produce a much simpler finite-state model of the Java program.

To understand the changes we propose in the verification process, we first need to give a brief introduction to the specification and verification stages in Bandera, and other software model checkers. We ignore some actual limitations imposed by Bandera due to implementation restrictions or arbitrary design decisions not to implement some features of Java, and relate to a somewhat idealized version.

Given a Java program, we need to augment it to include definitions using BSL. For a simple assertion about the state whenever a given location is reached, or pre and post-conditions of a method, we write the assertion definitions - using BSL - as Javadoc comments directly in the source code. An assertion is identified by a @assert tag in BSL, where the three assertion types supported by BSL are identified by the identifiers: LOCATION, PRE and POST.

The specification of a more general temporal program property is divided into defining the predicates to be used in the property's definition, and then separately writing the property itself, using the defined predicates. Predicates are, like simple assertions, also planted directly in the source code, where there are several types of predicates that Bandera allows us to define. For example, we may define a *location* predicate, which is *true* whenever the location is reached (and *false* otherwise), by introducing a Java label at a given control point (inside a method) of the program, and also writing a Javadoc comment (right before the associated method heading) containing the predicate definition in BSL.

An *instance* predicate defines a given property that is not connected to any control point of the program, e.g., invariant properties that must hold during the whole life cycle of an object. In addition, it is also possible to define predicates associated with two different method call control points: when a given method is invoked and when it returns a value. In this case, the predicate evaluates to *true* both when the given method is invoked and when it returns a value. Every predicate definition is written in a Javadoc comment. A predicate definition is identified by a @observable tag in

BSL, where the four predicate types supported by BSL are identified by the identifiers: LOCATION, EXP, INVOKE and RETURN; location, instance, method invoking, and return predicates, respectively.

In the second step needed for defining a given temporal property, after having defined all the predicates that it needs, we need to specify the required temporal property using the temporal specification patterns supported by Bandera, which are: absence, existence, precedence, response and universality. Let P, Q, R be predicates defined using BSL. P is *absent* in a program if it never evaluates to true. P *exists* if it is evaluated to true at least once in the program. P *precedes* Q when P does not evaluate to true before Q is true (which is automatically satisfied when P is *absent*). P *responds* to Q if after Q is true, then P *exists* (which is automatically satisfied when Q is *absent*). P is *universal* if P always evaluates to true.

In Bandera's temporal specification pattern system, we may require a temporal property to hold *globally*, i.e. during all the program execution, or at certain points during the program execution, such as $after\ Q$, $after\ Q\ until\ R$, $before\ Q$, $between\ Q$ and R, where Q and R are predicates defined using BSL.

The temporal specification of a given program is stored in a separate specification file. After having specified all the assertions and temporal properties required for verifying the correctness of the program, we may use Bandera's graphic tool to define a verification session and supply all the data needed, such as the names of the files containing the source code and the specification. When running a correctness check, we may choose exactly which of the assertions and temporal properties defined we want to verify.

Moreover, it is also possible to use data abstractions to simplify the finite-state model generated by Bandera. For example, in a pipeline program shown in [6], a series of integer values, ranging from 1 to 100, is sent from the first stage to the last, passing by all the pipeline stages. When the pipeline program finishes, the first stage sends a 0 value, and then all the stages finish consecutively. In the specification of this example, the integer values - ranging from 1 to 100 - sent in the pipeline are not important. We only need to know when a stage receives a 0 value. Therefore, we may use Bandera's *Signs* data abstraction, which will generate only three different states for the possible values that are sent in the pipeline: negative, zero and positive; instead of more than a hundred different states. Bandera's graphic tool has an interface for defining data abstractions, which we can afterwards store in a separate file. We may also select Bandera's program slicing feature for simplifying the finite-state model generated. After defining the verification session, we only need to run the verification checker, obtaining formal verification of the property if the model checking completes without discovering an error, and otherwise provides a counter-example in terms of the Java code.

In the Appendix, we use a bounded buffer program to give a brief demonstration of all the Bandera concepts introduced in this section. This program is a slightly changed version of an example seen in [6]. Explanations of the example may be also found in the Appendix.

4. PROVING CORRECTNESS IN SUPERJ

4.1 Introduction

In this and the following sections we explain and demonstrate the different options for verifying that a combination of a superimposition and a basic program is correct, as supported by the new features of SuperJ. In Section 4.2, we explain the verification of a combination of a superimposition and a basic program. In Section 4.3, we introduce the intuitively attractive option of proving the correctness of a superimposition independently of any basic program, and discuss the practicality of this option. In Section 5, we use a simple superimposition example to demonstrate some of the concepts introduced by the new SuperJ features, and discuss the implications for superimposition validation in Section 6.

4.2 Superimposition over a Basic Program

In this subsection, we assume a superimposition and a basic program. We want to apply the superimposition over the basic program, checking that the basic program satisfies the superimposition assumptions and that the resulting augmented program is indeed correct, i.e., satisfies all the desired results of the superimposition, as well as the original specification of the basic program. The simplest possibility is to simply view the result of weaving the superimposition's aspects with the basic program as a Java program that should satisfy the original specification, plus the result assertions of the superimposition. Following the description in Section 3, we then may build in all the needed functions, predicates, labels, and BSL statements to the augmented program, create the separate specification file, and model check all at once that the needed temporal BSL assertions are satisfied (or obtain counter-examples).

This is the simplest option for verifying the correctness of a combination of a superimposition over a basic program, since we directly consider the augmented program, and add in all of the needed predicates and assertions in BSL, as seen in the previous section. However, in this case the assumptions and desired results of the superimposition are already instantiated for the combination, and are mixed together with the original specification of the basic program. When a new combination is done, a completely new annotation has to be added before Bandera can be applied. This makes the model checking impractical when the superimpositions are to be used in many contexts. Thus we now propose a better option.

In order to more clearly organize the proofs, and thus to help in identifying the source of any errors, new verification aspects and superimpositions can be used to modularize the treatment. This allows having regular superimpositions and basic programs, free of verification definitions. The extra definitions needed for Bandera's verification are isolated in dedicated aspects, which are used just for proving the correctness of the augmented program in separate steps.

When completely separating the verification definitions from the superimposition and basic program, we have a series of verification aspects that may be sequentially applied to the basic program, or may be combined using combinations of

superimpositions. Moreover, we may now define a *verification superimposition* as a collection of verification aspects. We may classify the verification superimpositions in three different types, defining:

Spec the specification of the basic program;

Asm the superimposition assumptions;

Res the superimposition desired results.

The *Spec* superimposition will have one or more verification aspects, which will contain (AspectJ) advice declarations needed for introducing the verification definitions of the basic program's specification. It also includes the BSL temporal properties which in Bandera are kept in a separate file.

The *Asm* superimposition, dealing with assumptions, will have a collection of verification aspects: one verification aspect for each generic aspect that assumes some properties about the basic class to be bound to it; and one verification aspect for the global assumptions of the superimposition that must be satisfied by the basic program, where these assumptions are not connected to only a generic aspect and its (bound) basic class. Clearly, the assumptions should be as weak as possible, in order to allow applying the superimposition to a large class of basic programs.

A *Res* superimposition is very similar to an *Asm* superimposition, except that it specifies the superimpositions desired results instead of its assumptions. *Res* will also have a collection of verification aspects, like *Asm*.

The complete verification process is composed of four steps:

1. apply *Spec* over the basic program and check its correctness;
2. apply *Asm* over the basic program, and check that the basic program satisfies the superimposition assumptions;
3. apply the superimposition over the basic program, apply *Spec* over the augmented program, and then check that the superimposition does not cancel any desired result of the basic program;
4. apply the superimposition over the basic program, apply *Res* over the augmented program, and check that the augmented program achieves the desired results.

Note that *Spec* is used twice, and that the separation of the verification definitions into aspects and superimpositions is a cleaner solution than the comments used by Bandera to sometimes use and sometimes ignore the verification definitions. Of course, if some of the model checking has already been done for a basic or augmented program, it need not be redone. For example, if the basic program has been shown to satisfy *Spec* once, this need not be redone when applying a superimposition. The parameterization in the verification aspects allows their reuse for different basic programs, with different weavings and instantiations. The advantages of this reuse are further considered in the Discussion section.

4.3 Proving Superimposition Correctness

In this section we consider how to prove that a superimposition is correct independently of any basic program. If we succeed, then we are assured that when this superimposition is applied over a basic program that satisfies its assumptions, then the augmented program will have the superimposition's desired properties. Such a verification is desirable if the superimposition is intended to be put in a library for reuse in many contexts. Of course, if such a proof has been done, we still need only the model checking proofs that the basic program satisfies the assumptions of the superimposition, and that the result of weaving does not violate the specification of the basic program.

The generic correctness requirements and stages in such a proof are not difficult to state in terms of inductive assertions about the structure of every possible basic program to which the superimposition can be applied. However, any such proof has a part which is inductive, and thus non-algorithmic, requiring the invention of inductive assertions. This is true both when the entire proof is based on inductive theorem proving, and when the proof can be divided into a model checking part and an inductive part proving that if the model checking part is successful, then the desired conclusion is justified.

One way to do such a combination of model checking with an inductive proof to obtain a correctness proof of a superimposition uses what can be called *dummy* basic programs, first proposed in [11]. Note that model checking tools verify a model of a fully defined program by checking that the specified properties hold in all execution paths of the program. A superimposition, however, is itself not a program, since it cannot be run, so there are no execution paths. Therefore, we need to write an abstraction of a basic program that fits the superimposition's assumptions, so that we can apply the superimposition over the abstraction. Then we will have execution paths that may be used to prove the correctness of the superimposition combined with the abstract program. This program abstraction may be seen as a dummy basic program.

The dummy program will have no desired results, since it does not do any useful computation. Thus, there will be no *Spec* verification superimposition in the correctness verification process. On the other hand, the other types of verification aspects and superimpositions will still appear, as explained in the previous section. The abstract program must have classes and states that satisfy the assumptions of the superimposition, and also states that correspond to predicates tested by the superimposition or locations that can be reached. That is, if a predicate is tested whenever a (parametric) method is called, the abstract program should have a state where the predicate is false when a (corresponding concrete) method is called, and another where it is true.

This is analogous to the abstraction seen in usual Bandera verifications, where only the 'significant' differences are maintained, as in the abstraction of message values already mentioned. It is also related to work on model checking a representative model built from a model-generating graph grammar and then concluding that any model that can be generated from the grammar will be correct [14].

Ideally, if the model checking succeeds for the combination of the superimposition over the abstract basic program, then it would succeed for any basic program satisfying the assumptions of the superimposition. However, techniques for proving this ideal conclusion are not yet developed, and in any case they are inductive except when there are trivial structural similarities between the 'real' basic program and the dummy actually model checked. If done successfully, any basic program satisfying the assumptions, and with sufficient components and states to allow binding to the superimposition and its aspects, can be abstracted to this canonic abstract basic program.

In general, the justification that a representative abstraction is indeed sufficient can itself involve infinite or very large state spaces and may require inductive theorem proving. In the Discussion section, we show that by carefully using the techniques in the previous subsection, it may not be necessary to generate such non-algorithmic proof obligations to obtain fully verified combinations of aspects and basic programs in practice.

5. CASE STUDY

5.1 Introduction

In this section, we demonstrate the stages in verifying a combination of a superimposition and a basic program using SuperJ by means of a case study over the Monitoring superimposition, which is shown in Figure 1. Monitoring is a simple superimposition that gathers statistics on basic objects, such as counting the total number of external method calls for all relevant basic objects. The superimposition does not modify the values of the variables. It also checks that objects intended to be constant, actually are - and stops the program when a violation is discovered. It thus does regulate the behavior of the basic program and can affect its properties. In reading the example, note that SuperJ has a keyword BC (an abbreviation for Bound Class) which is like *this* of Java, indicating the class to which this instance of the aspect is bound. Formal parameters are in capital letters, to distinguish them from local variables.

The Monitoring superimposition contains two generic aspects (Constant and Mutable) and one singleton class (Coordinator). Constant and Mutable extend the Common abstract aspect, which contains code common to both generic aspects. The Common aspect defines the Coordinator class and creates its single instance *coord*, which is used by Constant and Mutable; moreover, Common's advice increments the *nCalls* counter after each external call to any method of the bound class, where each aspect instance will have its particular *nCalls* counter. The Common's allExternalCalls pointcut is defined in both generic aspects of the superimposition (Mutable and Constant). The join points determined by this pointcut - in some bound (basic) object - are all the method calls where the basic object is the callee, but not the caller. In a basic object bound to Mutable, after each field assignment performed, Mutable's advice increments the *nAssigns* counter, where each instance of Mutable has its particular *nAssigns* counter. The only instance of Coordinator (*coord*) accumulates the global statistics gathered by Constant and Mutable. Basic objects intended to be constant, whose field values should not be changed, must be bound to the Constant aspect; and then, if a field assign-

```
superimposition Monitoring {
class Coordinator {
  private int totCalls = 0;
  private int totConCalls = 0;
  private int totMutCalls = 0;
  private int totMutAssigns = 0;

  public void conMethodCount(int x) {
    totConCalls += x; totCalls += x;
  }
  public void mutMethodCount(int x) {
    totMutCalls += x; totCalls += x;
  }
  public void mutAssignCount(int x) {
    totMutAssigns += x;
  }
}

abstract aspect Common {
  protected final static Coordinator coord =
    new Coordinator();
  protected int nCalls = 0;

  abstract protected pointcut allExternalCalls();
  after(): allExternalCalls() {
    nCalls++;
  }
}

aspect Constant(EM) extends Common {
  protected pointcut allExternalCalls(): !cflowbelow
    (within(Element)) && execution(* BC.* (..));
  before(): set(* BC.*) &&
            !cflow(initialization(BC.new(..))) {
    System.out.println("Constant err: illegal assignment");
    System.out.exit(-1);
  }
  after(): execution(* BC.EM(..)) {
    coord.conMethodCount(nCalls);
  }
}

aspect Mutable(EM) extends Common {
  protected int nAssigns = 0;

  protected pointcut allExternalCalls(): !cflowbelow
    (within(Element)) && execution(* BC.* (..));
  after(): set(* BC.*) {
    nAssigns++;
  }
  after(): execution(* BC.EM(..)) {
    coord.mutMethodCount(nCalls);
    coord.mutAssignCount(nAssigns);
  }
}
```

Figure 1: A monitoring superimposition.

ment is tried, the aspect prints an error message and finishes the execution of the augmented program.

Each basic object augmented by Mutable will call *coord*'s $mutMethodCount$ and $mutAssignCount$ methods, while objects bound to Constant will call *coord*'s $conMethodCount$ method. The $mutMethodCount$ and $conMethodCount$ methods both update the $totCalls$ common method call counter, and, respectively, update their $totMutCalls$ and $totConCalls$ individual counters. The $mutAssignCount$ method updates the $totMutAssigns$ assignment counter. Of course, Monitoring could make more sophisticated use of the gathered statistics. Generalizations of the same idea should be useful for bookkeeping and debugging. In particular, superimposition is especially appropriate when the generic aspects have more interaction, as when the statistics collected by each generic aspect are combined.

The assumptions and desired results of the superimposition are introduced stepwise in Section 5.2, where we verify the correctness of Monitoring over the bounded buffer program (seen in the Appendix), which is used as an example of a basic program.

5.2 Superimposition over a Basic Program

In this subsection we want to apply the Monitoring superimposition over the bounded buffer basic program, and verify the correctness of the augmented program, which we get as a result of their combination. We apply Mutable over BoundedBuffer, binding BoundedBuffer's $finish$ method to Mutable's EM parameter (an abbreviation for End Method). In addition, we apply Constant over Element, binding Element's $finish$ method to Constant's EM parameter. We show the whole verification process stepwise, as introduced in Section 4.

In the first step, we want to check that the basic program itself is correct, i.e., satisfies its specification. In the Appendix, we show the BoundedBuffer class with all the Bandera specification definitions interleaved with its code, where all these definitions are needed for verifying that the basic program satisfies BoundedBuffer's specification when using Bandera. In our approach this is already the result of applying the *Spec* superimposition of the bounded buffer to the original version of the program. This is given as input to Bandera, defining a new verification session with all the information needed by Bandera for running the verification, as shown in Section 3. We then run Bandera's verification to check if all the properties specified are satisfied. In this example, we succeed to show that the basic program is correct, since it indeed satisfies its specification, completing the first stage of the model checking.

In the second step, we want to check that the basic program satisfies all the assumptions specified by the superimposition. For this purpose, we use an *Asm* verification superimposition. *Asm* has a verification aspect for each generic aspect of Monitoring that assumes some property about the basic class to be bound to it. In addition, *Asm* has a verification aspect for the global properties assumed by Monitoring about the basic program, such as invariant properties, which are not connected to only a specific generic aspect.

```
superimposition MonitoringAsm {
  aspect CommonAsm {
   /**
    * @observable
    *     LOCATION[beforeCall] beforeCallLoc;
    *     LOCATION[afterCall] afterCallLoc;
    */
   void around(BC C): target(C) &&
           execution(* BC.*(..)) {
     beforeCallLoc:
     proceed(C);
     afterCallLoc:
   }
  }
  properties {
    alwaysFinishProp: forall [bc:BC].
           {BC.EM.beforeCall(bc)} exists globally;
    singleNoCallAfterFinishProp: forall [bc:BC].
           {BC.*.beforeCall(bc)}
           is absent after {BC.EM.afterCall(bc)};
  }
}
```

Figure 2: Monitoring's *Asm* superimposition

A property that both Mutable and Constant assume about basic classes is that the basic method that is bound to the EM parameter is called exactly once, where the EM parameter must be bound to the last method that is called in the basic object. In the sequel, the basic method bound to EM is called *bound_EM*. Another property that both Mutable and Constant assume is that *bound_EM* is the last method called in every instance of the basic class.

As explained in Section 3, in a usual Bandera verification session we write the specification of the temporal properties to be checked in a separate specification file. However, in SuperJ, we write this specification in a new *properties* section of the verification superimposition. We have written a preprocessor that supports this design decision, which separates the definitions in the properties section from the rest of the superimposition code and then prepares a new verification session for running the verification.

The specification of the properties assumed by the generic aspects need to use two location predicates that must be defined in the basic classes. These two predicates are defined in the verification aspect by the same advice, as shown in Monitoring's *Asm* superimposition, seen in Figure 2.

The single *Asm* verification aspect must be applied over all the basic classes to be bound to Constant and Mutable. In the bounded buffer example, they are applied, in turn, over Element and BoundedBuffer. The two predicates defined in the verification aspects are associated with two locations in each method of every basic class bound to Constant or Mutable (e.g. Element and BoundedBuffer). Each of these predicates is true during execution when the augmented program reaches the control points where they were defined, i.e., in an execution path. The control points associated with these predicates (beforeCall and afterCall) are right before the first and after the last commands executed

in the basic methods of Element and BoundedBuffer.

After having defined the two predicates needed for the verification, we can write the two properties that, if satisfied, will ensure that the basic program satisfies the two assumptions, which are required by both Mutable and Constant. These two properties are written in temporal logic using BSL, and appear in the properties section of the *Asm* superimposition.

The first property, which is called alwaysFinishProp, checks that *bound_EM* is eventually called. However, that is not enough, since we want this method to be called exactly once, and no other method to be called after that. Therefore, the second property (singleNoCallAfterFinishProp) checks that no basic method will be called after *bound_EM* is called.

We put a '*' character in the place where we should write the name of the basic method where beforeCall was defined. The '*' character fits every method of the basic class. Unfortunately, BSL does not support this special '*' character. In a usual Bandera specification, we need to write separate temporal properties for each method of the basic class. However, our preprocessor overcomes this limitation, generating all the properties needed for every method of the basic class.

In the example seen, both Mutable and Constant shared exactly the same requirements, so in this particular case we can use the same *Asm* aspect for both generic aspects. However, if the assumptions required by two distinct generic aspects differ, then we obviously need to write them in two separate aspects. Moreover, Monitoring does not assume any global property about the basic program, so there is no *Asm* aspect for checking if the global assumptions are satisfied.

At this stage, we are able to apply the verification superimposition over the basic program. We then create a new verification session for checking the superimposition assumptions, and then run the verification in Bandera.

After having demonstrated the second step of the new verification feature, we now go on to the third step, where we check that the superimposition does not cancel any of the desired results of the basic program. Initially, we need to apply the superimposition over the basic program, e.g., Mutable over BoundedBuffer and Constant over Element. Finally, we apply the *Spec* verification superimposition - containing the verification definitions needed for checking the basic program's specification - over the augmented program. Here we do not show the *Spec* superimposition, since we show its verification definitions interleaved with the code of the bounded buffer in the Appendix, together with its specification file. We then supply all the data that Bandera needs for the desired check and run the verification. If the augmented program passes the verification, then we are assured that the superimposition does not cancel any desired result of the basic program.

In the fourth and last step of the verification process, we want to check that the augmented program has all the desired results specified by the superimposition. For this purpose, we apply the superimposition over the basic program (Monitoring over the bounded buffer program), and then

```
superimposition MonitoringRes {
 /**
  * @observable
  * EXP Eq: (totCalls == (totConCalls + totMutCalls));
  */
 aspect GlobalRes {
  /**
   * @observable
   *     LOCATION[beforeConMC] beforeConMCLoc;
   *     LOCATION[afterConMC] afterConMCLoc;
   */
  void around(Coordinator C): target(C) &&
     execution(void Coordinator.conMethodCount(int)) {
    beforeConMCLoc:
    proceed(C);
    afterConMCLoc:
  }
  /**
   * @observable
   *     LOCATION[beforeMutMC] beforeMutMCLoc;
   *     LOCATION[afterMutMC] afterMutMCLoc;
   */
  void around(Coordinator C): target(C) &&
     execution(void Coordinator.mutMethodCount(int)) {
     beforeMutMCLoc:
     proceed(C);
     afterMutMCLoc:
  }
 }

 aspect ConstantRes {
 /**
  * @observable
  * LOCATION[beforeConFieldSet] beforeConFieldSetLoc;
  * LOCATION[afterConFieldSet] afterConFieldSetLoc;
  */
 before(): set(* Element.*) &&
         !cflow(initialization(Element.new(..))) {
  beforeConFieldSetLoc:
 }
 after(): set(* Element.*) &&
        !cflow(initialization(Element.new(..))) {
  afterConFieldSetLoc:
 }
 properties {
  totCallsEqBeforeProp: forall [c:Coordinator].
   {Eq(c)} is universal
   before{Coordinator.conMethodCount.beforeConMC(c) ||
          Coordinator.mutMethodCount.beforeMutMC(c)};
  totCallsEqAfterProp: forall [c:Coordinator].
   {Eq(c)} is universal
   after{Coordinator.conMethodCount.afterConMC(c) ||
         Coordinator.mutMethodCount.afterMutMC(c)};
   until{Coordinator.conMethodCount.beforeConMC(c) ||
         Coordinator.mutMethodCount.beforeMutMC(c)};
  conObjTermIfSetProp: forall [bc:BC].
   {BC.*.afterConFieldSetLoc(bc)}
   is absent after {BC.*.beforeConFieldSetLoc(bc)};
 }
}
```

Figure 3: Monitoring's *Res* superimposition

we apply the *Res* superimposition over the augmented program, where the *Res* superimposition checks that all the desired results of Monitoring are present in the augmented program. The complete *Res* verification superimposition is shown in Figure 3.

A desired result that the superimposition requires the augmented program to satisfy is that the value of Coordinator's field *totCalls* must be always equal to the value of its *totConCalls* field plus the value of its *totMutCalls* field, except when the augmented program is executing one of the two methods of Coordinator that change the values of these fields (*conMethodCount* and *mutMethodCount*). We need to define four predicates in the Coordinator class before specifying the required property in BSL. In addition, an instance predicate must be also defined, stating the desired result itself. Thus, the *Res* aspect associated with Monitoring must contain the definition of the instance predicate, and two advice declarations defining the other four predicates needed.

Moreover, Constant has one desired result and Mutant has none. The desired result of Constant is that the augmented program terminates if a field assignment is tried in the basic object bound to Constant. Therefore, we must write a *Res* aspect associated with Constant. Mutable does not need a separate *Res* aspect beyond the global required result of the Monitoring superimposition.

The four predicates - defined by the global *Res* aspect - are associated with the augmented program's control points before and after the *conMethodCount* and *mutMethodCount* method, respectively. The *Eq* instance predicate defines the property that must be satisfied in the augmented program. The two predicates - defined by the *Res* aspect associated with Constant - will be true before and after a field assignment is tried in the basic object bound to Constant (an instance of Element).

We write the specification of the superimposition desired results, using BSL, in the properties section. In the two first properties seen, we specify that the *Eq* property must hold from the beginning of the execution of the augmented program until either *conMethodCount* or *mutMethodCount* is called, and after finishing to execute either one of them until calling one of them again. The third temporal property specifies that if the augmented object bound to Constant (an instance of Element) reaches the control point right before a field assignment, then it will not reach the control point right after the field assignment.

Above, we have seen a demonstration of the complete process of verifying the correctness of a superimposition over a basic program. The augmented program that we get from applying Monitoring over the bounded buffer program passes all the stages of the verification process. However, some slight changes in the bounded buffer program could cause it to not satisfy the assumptions required. For example, if we substitute an infinite loop in place of the *for* loops of InOut1 or InOut2 that take and add an element from the buffer one hundred times, the model checking produces a counter-example and shows incorrectness. This is because the *finish* methods of the buffer and its elements would never be called, violating one of the assumptions of the Monitoring superimposition.

If the Monitoring superimposition could change the indices of the buffer of the underlying bounded buffer program, a counter-example would be produced when *Spec* were model checked for the augmented program (in stage 3), because the assertions involving the indices would be violated.

6. DISCUSSION: SUPERIMPOSITION VALIDATION

The separation of verification annotations into the different verification superimpositions described above allows a clean application of instances of model checking for combinations of superimpositions and basic programs. Note that when a verification superimposition is woven either with a new basic program or with the augmented program obtained after weaving the application superimposition and the basic program, the weaving process binds classes, methods, fields, and pointcuts of the generic verification superimposition to those of the application. No change is needed in *Asm* or *Res* themselves. Of course, the specification of the new basic program, *Spec* needs to be produced, and expressed as a verification superimposition.

Once the bindings have been determined, the entire process is in principle automatic, ignoring practical restrictions of the tools involved. When a superimposition is woven with a basic program, SuperJ's preprocessor generates AspectJ code, and AspectJ's preprocessor is used in the mode which generates source Java code. Then for each of the four steps described, the appropriate verification superimposition is woven with the basic or augmented program, as appropriate, and the processing of SuperJ and AspectJ are again activated, to obtain 'Bandera-ready' Java. Bandera then is applied to generate input to a model checker such as SMV, and the algorithmic model-checking either succeeds in verifying or generates a counter-example.

Therefore, although it might seem expensive to model check every combination of a superimposition with a basic program, this is in fact a viable alternative to the inductive (non-algorithmic and therefore very difficult) proof that a superimposition is always correct. The time-consuming, and difficult manual creation of the BSL annotations only needs to be done once for each superimposition, even though the model checker is used for each combination.

Such an alternative is analogous to the idea of *translation validation*, first seen in [13], where assertions are generated and automatically checked whenever a compiler is applied to a source program. The correctness of the assertions implies that *for this activation* the translation of the compiler is correct. This is instead of a full verification of the correctness of the compiler, which is too difficult for non-toy compilers. As here, the key to its practicality is that the generation and verification of the needed assertions is completely automatic for each compilation, and only takes seconds to perform. Similar ideas are seen in some versions of proof-carrying code, that show there are no memory leaks for a particular instance of an applet.

In this paper we have shown how *superimposition validation*

can be similarly applied whenever an application superimposition is woven, if the needed verification superimpositions have been prepared. The other alternative - of a full correctness proof for a superimposition - is, of course still a desirable research goal. However, due to the inductive proof involved, doubt remains that such results can be applied in practice. In any case, the direction seen here does provide the first pathway to practical machine proofs for combinations of aspects and superimpositions with basic programs.

7. REFERENCES

[1] R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[2] L. Bougé and N. Francez. A compositional approach to superimposition. In *ACM Symposium on Principles of Programming Languages*, pages 240–249, Jan 1988.

[3] K. Chandy and J. Misra. *Parallel Program Design - a Foundation*. Addison-Wesley, 1988.

[4] N. Francez and I. Forman. *Interacting Processes*. Addison-Wesley, 1996.

[5] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *CONCUR 2001*, LNCS 2154, pages 39–58, Aug 2001.

[6] J. Hatcliff and O. Tkachuk. The Bandera tools for model-checking Java source code: A user's manual. Technical report, Kansas State University, Department of Computing and Information Sciences, March 2001. `http://www.cis.ksu.edu/%7Esantos/bandera/tut/tut-html.tar.gz`.

[7] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), Apr 2000.

[8] G. J. Holzmann and M. H. Smith. The model checker SPIN. *IEEE trans. SE*, 23(5):279–295, 1997.

[9] R. Iosif and R. Sisto. dspin: A dynamic extension of spin. In *Proc. of the 6th SPIN Workshop*, LNCS 1680, pages 261–276, Sep 1999.

[10] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings ICSE'90*, pages 63–71. IEEE Press, 1990. `http://disco.cs.tut.fi`.

[11] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. on Programming Languages and Systems*, 15(2):337–356, Apr 1993.

[12] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[13] A. Pnueli, O.Shtrichman, and M.Siegel. The code validation tool(cvt) - automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2:192–201, 1999.

[14] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Proc. of the international workshop on Automatic verification methods for finite state systems*, pages 151–165. Springer-Verlag, 1990.

[15] M. Sihman and S. Katz. Superimposition and aspect-oriented programming. to appear in BCS Computer Journal. Available at `http://www.cs.technion.ac.il/~katz/cj.ps`.

[16] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proceedings of AOSD 2002*, pages 28–40. ACM Press, Apr 2002.

APPENDIX

A. BOUNDED BUFFER EXAMPLE

A.1 Introduction

The bounded buffer example is a multi-threaded Java program introduced in [6] as an example for demonstrating a verification session in Bandera. The BoundedBuffer class has three methods: $add(Element)$, $take()$, $isEmpty()$. When the buffer is not full, the *add* method adds an Element object to the buffer, which is defined as a fixed array of Element objects. The *take* method takes an Element object (*element*) from the buffer, if the last is not empty. The *isEmpty* method returns true when the buffer is empty, and false otherwise. The constructor of BoundedBuffer receives (as parameters) the size of the buffer array and the number of threads running (using the bounded buffer), and then initializes all the object fields.

The other classes that appear in this example are: CompleteBoundedBuffer, InOut1, InOut2 and Element. The first is the main driver class that runs the program, creating two BoundedBuffer instances and single instances of InOut1 and InOut2. The InOut1 instance is a thread that contains a finite loop where it takes an *element* from the first buffer and adds it to the second, while the InOut2 instance has an identical finite loop that takes an *element* from the second buffer and adds it to the first buffer. CompleteBoundedBuffer creates two *elements* and adds them respectively to the first and second buffers, where an *element* contains an *Object* instance as its only field, and has two methods that allow changing and getting the *Object* instance that it contains. Both BoundedBuffer and Element classes contain a $finish$ method that performs computation destined to be executed when the program finishes.

Five properties are checked in the bounded buffer example. The BoundedBuffer's constructor parameter (for the size of its array) must be a positive number, which is specified by the PositiveBound assertion. The *add* method always adds the *element* in correct position, which is specified by the addPost assertion. The buffer indices ($head\_$ and $tail\_$ BoundedBuffer fields) always stay in range, which is specified by the temporal property IndexRange, which uses the IndexRange instance predicate. A full buffer eventually becomes non-full, which is specified by the FullToNonFull temporal property, which uses the Full instance predicate. An empty buffer must have an *element* added to it before an *element* is taken from it, which is specified by

the NoTakeWhileEmpty temporal property, which uses the Empty instance predicate and the takeReturn and addInvoke location-sensitive predicates.

A.2 Source Code

```
public class CompleteBoundedBuffer {
 public static void main (String [] args) {
    BoundedBuffer b1 = new BoundedBuffer(3,2);
    BoundedBuffer b2 = new BoundedBuffer(3,2);
    b1.add(new Element(new String("1")));
    b2.add(new Element(new String("2")));
    (new InOut1(b1,b2)).start();
    (new InOut2(b2,b1)).start();
 }
}

class Element {
 Object obj;
 Element(Object o) {···}
 public void set(Object o) {···}
 public Object get() {···}
 public void finish() {···}
}

/**
 * @observable
 * EXP Full: (head_ == tail_);
 * EXP Empty: head_ == ((tail_+1) % bound_);
 * EXP IndexRange: (head_ >= 0 && tail_ >= 0 &&
 *                  head_ < bound_ && tail_ < bound_);
 */
class BoundedBuffer {
 Element [] buffer_;
 int bound_;
 int head_, tail_;
 int nThreadsRun, nThreadsEnd = 0;

/**
 * @assert
 * PRE PositiveBound: (b > 0);
 */
 public BoundedBuffer(int b, int n) {···}

/**
 * @assert
 * POST addPost: (head_==0) ? buffer_[bound_-1]==o :
 *                            buffer_[head_-1]==o;
 * @observable
 * INVOKE addInvoke;
 */
 public synchronized void add(Element o) {···}

/**
 * @observable
 * RETURN takeReturn;
 */
 public synchronized Element take() {
    ...
    successTake:
    ...
 }

 public synchronized boolean isEmpty() {···}

 public synchronized void threadFinished() {
    if (++nThreadsEnd == nThreadsRun) {
       finish();
    }
 }

 public synchronized void finish() {···}
}

class InOut1 extends Thread {
 BoundedBuffer in_,out_;
 public InOut1(BoundedBuffer in, BoundedBuffer out) {···}
 public void run() {
    ...
    for(int i=0; i<100; i++) {···}
    in_.threadFinished();
    out_.threadFinished();
 }
}

class InOut2 extends Thread {
 BoundedBuffer in_,out_;
 public InOut2(BoundedBuffer in, BoundedBuffer out) {···}
 public void run() {
    ...
    for(int i=0; i<100; i++) {···}
    in_.threadFinished();
    out_.threadFinished();
 }
}
```

A.3 Specification

```
PositiveBoundAndPost: enable assertions
      {PositiveBound, addPost};

IndexRange: forall[b:BoundedBuffer].
      {IndexRange(b)} is universal globally;

FullToNonFull: forall[b:BoundedBuffer].
      {!Full(b)} responds to {Full(b)} globally;

NoTakeWhileEmpty: forall[b:BoundedBuffer].
      {BoundedBuffer.take.takeReturn(b)} is absent
      after {BoundedBuffer.Empty(b)}
      until {BoundedBuffer.add.addInvoke(b)};
```

Understanding AOP through the Study of Interpreters

Robert E. Filman
Research Institute for Advanced Computer Science
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov

ABSTRACT

I return to the question of what distinguishes AOP languages by considering how the interpreters of AOP languages differ from conventional interpreters. Key elements for static transformation are seen to be redefinition of the set and lookup operators in the interpretation of the language. This analysis also yields a definition of crosscutting in terms of interlacing of interpreter actions.

1. INTRODUCTION

I return to the question of what distinguishes AOP languages from others [12, 14].

A good way of understanding a programming language is by studying its interpreter [17]. This motif has been recently emphasized in recent work by Masuhara and Kiczales [21], and is a theme of the work on the Aspect Sandbox [22, 30]. This position paper suggests studying the foundations of Aspect-Oriented Languages by considering what changes have to be made to conventional language interpreters to get aspect behaviors.

Interpreters express meaning. Compilers can be understood as optimizations that enable more efficient renderings of the work of interpreters, without changing the underlying meaning of programs. A compiler that builds a "new program" from several fragments can be understood as a substitute for the dynamic, run-time building of that new program from fragments. Thus, while compilation techniques for AOP (e.g., partial evaluation as weaving [22]) are quite worthwhile activities, they do address the question of the nature of AOP.

2. A GENERIC INTERPRETER

Consider the pseudo-code for interpreter for A Generic Programming Language (AGPL) in Figure 1. This is a "meaning" function over an "expression" (an object in expression space), and an "environment," a structure that maps names to values, perhaps with a characterization of what kind of mapping is of interest (e.g., variables vs. functions). The pseudo-code includes only enough detail to convey the ideas I'm trying to express.

Of course, a real implementation would need implementations of the helper functions. In general, the helper functions on environments—lookup, set, and extend—can be manipulated to create a large variety of different language features. The most straightforward implementation makes an environment a set of symbol–value pairs (a map from symbols to values) joined to a pointer to a parent environment. Lookup finds the pair of the appropriate symbol, (perhaps chaining through the parent environments in its search), returning its value; set finds the pair and changes its value field; and extend builds a new map with some initial values whose parent is the environment being extended. In this model, lookup and set are "reference assignment" pairs: they act like elements setting and retrieving the value of a location. Programming languages vary by their use of chaining in environments. Most languages have some notions of global environment (a parent of or shared by all elements) and of constant elements (ones that don't change, such as a global function being assigned to a particular value.) Some languages may use the name being looked up or set as a structured object that guides the search in the environment space. More formal approaches would substitute a monad for the state expressed in the environment, but that level of formality would only obscure the discussion here.

I have provided set and lookup types (e.g., `VARIABLE` and `FUNCTION`) so that the implementations of set and lookup can be manipulated to separate things such as the function and variable space (as in Common Lisp [26]) or to conflate them (as in Scheme ([5]). By providing a richer notion of "set," one can create languages that export and restrict visibility; by providing a richer notion of "lookup" one can get inheritance. Most appropriate for doing independently created aspects (as opposed to aspects merely defined in the same "file") is the idea that certain varieties of environment.set change or extend some root (or at least non-leaf) environment.

Focusing on set and lookup corresponds to the importance of naming in practical programming languages. Much of the art of programming language design is the rules for associating names with meanings and groupings, and the visibility of these names; much of the act of programming is invoking named entities, dynamically associating names with values, and retrieving the values of names.

/* Compute the meaning of an expression, exp, given a environment, env

<0> I cheat by using the stack of the machine interpreting meaning as the stack for meaning. A richer (and perhaps more appropriate) system can be build by maintaining our own stack, allowing searches within that stack for elements like catch/throw and dynamic calling scope.

<1> The meaning of a literal expression is the constant of the expression. Numerals, strings, and quoted expressions are literals.

<2> If exp is a variable, look up its meaning in the environment with respect to variable lookup.

<3> If exp is a primitive operator (one that executes on the underlying machine, like "plus" or "print,") evaluate the meanings of its arguments in the current environment, assemble them into a "value list," and invoke the primitive operator on that list.

<4> If exp is some form of language-explicit interpreter control (like an "if" or "switch" statement), compute the meaning of the condition of the expression, and then return the meaning of the appropriate choice element (like the "else part" or the "default case.")

<5> If exp is an assignment statement, change the environment appropriately. The assignmentType covers the varieties of assignments one might want to make—for example, assigning to a variable, defining a local function, defining the fields of a record, or defining a new global function.

<6> Call a function. Find the body associated with that function. Build a new environment, based on the original environment and perhaps some environmental information of the definition itself, which binds the formals of the called function to the values of the actual parameters, and compute the meaning of the body in this new environment. I could have generalized this a bit beyond call by value, but it's not worth the trouble for the ideas I'm trying to convey.

*/

```
meaning (exp, env) =                                                                    /* <0> */
    typecase (exp) :
        literal     (exp) -> exp.literalValue                                           /* <1> */
        variable    (exp) -> env.lookup (exp.variableName, 'VARIABLE)                   /* <2> */
        primop      (exp) -> apply (exp.primop, meaningList (exp.args, env))            /* <3> */
        conditional (exp) -> meaning (exp.conditionChoice (meaning (exp.condition, env)), env)   /* <4> */
        assignment  (exp) -> env.set (exp.variableName, meaning(exp.value, env), exp.assignmentType) /* <5> */
        funcall     (exp) -> let definition = env.lookup (exp.functor, 'FUNCTION)       /* <6> */
                             in meaning (definition.body,
                                         env.extend (definition.formals,
                                                     definition.environment,
                                                     meaningList (exp.args, env)))
```

Figure 1: AGPL interpreter

3. CROSSCUTTING AND BLAME

One can assign credit (or blame) to every external action (a primop) or manipulation in the interpreter. Each action in the interpreter is associated with a particular expression, the most immediate cause for that action. One can divide expressions into "modules." In general, actions follow the structure of expressions, and actions tend to proceed within a module. I call this overall notion of the corresponding continuity in the expression space and the action sequence *locality*.

We have *crosscutting* when sequences of actions intermix from different modules. In conventional languages crosscutting arises most often as explicit invocation: an expression in one module names an entry of another module, and the system transfers control to that other module. Some conventional languages allow other crosscutting mechanisms. For example, in languages with function pointers or dynamic binding, the value of a dynamic environmental element can be used as an expression for further evaluation. Inheritance mechanisms also combine the code of several modules (equivalent to modifying the lookup function to search parent environments). In some languages, type declarations can have the effect of remotely modifying behavior. (Such mechanisms lie between the explicit invocation of a Fortran subroutine call and AOP.) Exception generation and handling can cause jumps in the execution sequence. One can also define the system in a "feature specific" manner, so that user-supplied code always runs in some specific circumstance. These latter mechanisms cause crosscutting.

The novelty of AOP is that the crosscutting mechanisms are implicit (oblivious) and general-purpose. That is, examination of the source code doesn't indicate that the crosscutting takes place. Instead, some external mechanism performs the surgery on the execution process. Modern AOP demands that the crosscutting mechanism be "general purpose," allowing modifying any code with respect to the structure of that code, not just a particular semantics. (This contrasts with some of the earlier special-purpose "aspect" languages [20].) Thus, a system that allows the user to define, say, "security code" to be invoked in particular contexts is a framework, not an AOP language.

4. MODIFYING THE INTERPRETER

The purpose of this exercise is to ask what does one have to do to make AGPL aspect-oriented? Here we are concerned with general aspect behavior, not a hook for solving a particular problem. That is, we want to be able to invoke arbitrary user code at joint points, not merely a selection from some predefined or parameterized behaviors.

We first note that modifying the interpreter for the specific requirements of a particular aspect language can always yield any (implementable) aspect language. Most generally this is true because any implemented aspect language has an interpreter. More specifically, every aspect language defines certain elements or events as joint points, places where it is possible to associate aspect behavior with the underlying code. We can change the interpreter to pause at every such join point and consult the (perhaps dynamic) dictionary of current aspects to see which apply. (And, as many

have observed, "Anything you can do I can do meta"—in a meta-interpreter architecture, we can delay to the meta level the decision about whether each execution point is a join point [4, 27].) Given a rich enough language for describing the desired aspect conditions, determining the places that need modification (effectively, the shadow points in the program or the execution points of such shadows in the interpreter) may be an interesting problem [15, 22].

The problem with such an analysis is that changing the body of the interpreter is the way to implement any conceivable language. We'd prefer to restrict the changes to more neatly describe the aspect space. More specifically, the problem is not so much describing mechanisms to implement aspect languages but, ideally, mechanisms that implement only aspect languages, or, more realistically, mechanisms whose parameterization approximates the space of aspect languages.

4.1 Advising a function

More than one research group has provided its interpretation of how best to implement AOP. Perhaps the most primitive mechanism, common to most approaches is "advice" (wrapping) [28]. With advice, the definition of a function is embedded inside other behavior, which can execute before, after, or around the original function. Systems that allow wrapping include Composition Filters [3], OIF [13], AspectJ [18], and JAC [25]. A structurally consistent way to get advice is to change the definition of functions to include advice. To advise a single function F with advice A, creating $A(F)$, we could find the pair that joins F to its definition, and replace its value by $A(F)$.

More commonly, we want to advise not one function, but an entire set of them, particularly the ones that pass some predicate test. That is, we want to quantify over the function space. An AOP system can be built with either an *open-world* or *closed-world* assumption. Closed world systems know at the start of execution all the code that might run in the system. Thus, a closed-world system could implement quantified advice by finding all the function definitions and redefining the ones that need the advice. An open-world system can dynamically acquire new code. In an open-world system, we also need to modify environment.set so that function definition and redefinition work with the advice mechanism—defining or redefining an advice-worthy function, must make the setting include the advice.

Note that there is also a natural symmetry between set and lookup. Anything one imagines doing at "set" time can be done at "lookup" time, so long as sufficient information is retained to perform the action.

4.2 Advising a field

Some AOP approaches (e.g., Hyper/J [24]) treat object fields as combinations of other elements. For example, one has the ability to externally state that field f in object r is to be the same as field f' in object r' when r and r' are regarded as parts of the definition of the same object, or that f in r and f in r' are not the same, even when r is merged with r'. Treating a variable as a combination of other elements in some sense, is symmetric to the functional advice problem. With functional advice, we are working in function space and know only a few combinators (e.g., before, after, and around), though others are easy to imagine (for example, consider mixins in Flavors [23]). With variables, we're working in variable space, and can think of a variety of combinators—for example, the "same as" and "different" examples, above, "union" for set-valued fields, "append" for sequence valued ones, and so forth.

4.3 Program transformation

Several authors have argued for doing AOP by program transformation [6, 11, 15, 16, 19]. From the point of view of an interpreter, program transformation can be realized by performing the transformation steps as part of the function definition process. (This is, of course, a somewhat heavy-handed interpretation of transformation.)

4.4 Frameworks

Frameworks (e.g., [8]) combine functional wrapping with wrappers specific to framework decision points. This can be seen as a structured step in function assignment. However, frameworks more naturally resemble modifying the interpreter to the special case doing additional behavior on function calling.

4.5 Field and method insertion

Some AOP approaches (e.g., [18]) allow the introduction of additional fields and methods. Once again, these are examples of changing the semantics of environment setting.

4.6 Dynamic flow

There have been several proposals for aspects that pay attention to the dynamic behavior of program execution. For example, aspect invocation in AspectJ can be predicated on what's in the calling history (cflow) [18]. At the first FOAL workshop, we argued for generally treating AOP as generically reacting to execution events [15], a theme also expressed by others [7, 10, 9, 16, 29]. The effects of such proposals are more problematic for interpreter transformation. Cflow can be accommodated if we create our own stack for the interpreter, rather than using the implicit stack of the system executing the interpreter and search that stack at appropriate join points. Alternatively we could change the definitions of functions to leave appropriate markers lying around to be recognized at the right instants. These require some structural changes to the interpreter. Similarly, event reaction can be seen to be requiring pervasive interpreter change.

5. CLOSING REMARKS

In this position paper, I've explored the idea that the changes required in "ordinary" interpreters to realize AOP languages reveals elements about the essence of AOP languages. Many (particularly the static varieties) of AOP mechanisms can be seen as redefinition of the storage or retrieval actions in the interpreter, often at record and method definition time. Join point definitions that span multiple locations require the definition, storage or retrieval mechanisms to "quantify" over the space of candidate points. I've also defined crosscutting in terms of the mixture of modules causing actions to execute, and identified AOP with that crosscutting that lacks explicit or implicit mention in the module code.

6. REFERENCES

[1] *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, June 2001.

[2] *FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002)*, Mar. 2002.

[3] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[4] N. M. N. Bouraqadi-Saâdanii and T. Ledoux. How to weave? In *Workshop on Advanced Separation of Concerns (ECOOP 2001)* [1].

[5] W. Clinger and J. Rees. Revised4 report on the algorithmic programming language scheme. *LiSP Pointers*, 4(3), 1991.

[6] G. A. Cohen. Recombing concerns: Experience with transformation. In *Workshop on Multi-Dimensional Separation of Concerns (OOPSLA 1999)*, Nov. 1999.

[7] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM Symp. on Principles of Programming Languages*, pages 54–66, Jan. 2000.

[8] C. A. Constantinides, T. Elrad, and M. Fayad. Extending the object model to provide explicit support for crosscutting concerns. *Software Practice and Experience*, 32(7):703–734, May 2002.

[9] K. De Volder, J. Brichau, K. Mens, and T. D'Hondt. Logic meta-programming, a framework for domain-specific aspect programming languages. http://www.cs.ubc.ca/kdvolder/binaries/cacm-aop-paper.pdf.

[10] K. De Volder and T. D'Hondt. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd International Conference on Reflection*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[11] K. De Volder, T. Tourwé, and J. Brichau. Logic meta programming as a tool for separation of concerns. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[12] R. E. Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns (ECOOP 2001)* [1].

[13] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Comm. ACM*, 45(1):116–122, Jan. 2002.

[14] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[15] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In AOSD-FOAL02 [2], pages 45–49.

[16] P. Fradet and M. Südholt. AOP: Towards a generic framework using program transformation and analysis. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[17] D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages (2nd ed.)*. Massachusetts Institute of Technology, 2001.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Comm. ACM*, 44(10):59–65, Oct. 2001.

[19] G. Kniesel, P. Costanza, and M. Austermann. JMangler—a framework for load-time transformation of Java class files. In *First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, Nov. 2001.

[20] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[21] H. Masuhara and G. Kiczales. A modeling framework for aspect-oriented mechanisms; draft. http://www.cs.ubc.ca/

[22] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In AOSD-FOAL02 [2], pages 17–26.

[23] D. A. Moon. Object-oriented programming with flavors. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8. ACM Press, Nov. 1986.

[24] H. Ossher and P. Tarr. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM*, 44(10):43–50, Oct. 2001.

[25] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*, pages 1–24. Springer-Verlag, Sept. 2001.

[26] G. Steele Jr. *Common Lisp: The Language, 2nd Edition*. Digital Press, Bedford, Massachusetts, 1990.

[27] G. T. Sullivan. Aspect-oriented programming using reflection and meta-object protocols. *Comm. ACM*, 44(10):95–97, Oct. 2001.

[28] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4):25–34, Apr. 1981.

[29] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[30] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In AOSD-FOAL02 [2], pages 1–8.

Adding Superimposition To a Language Semantics

— Extended Abstract —

Ralf Lämmel[1,2]
[1] CWI, Kruislaan 413, NL-1098 SJ Amsterdam
[2] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Ralf.Laemmel@cwi.nl

Abstract

Given the denotational semantics of a programming language, we describe a general method to extend the language in a way that it supports a form of *superimposition* — just in the sense of aspect-oriented programming. In the extended language, the programmer can superimpose additional or alternative functionality (aka advice) onto points along the execution of a program. Adding superimposition to a language semantics comes down to three steps: (i) the semantic functions are elaborated to carry advice; (ii) the semantic equations are turned into 'reflective' style so that they can be altered at will; (iii) a construct for binding advice is integrated. We illustrate the approach by representing semantics definitions as interpreters in Haskell.

1 Introduction

One might say that the essence of an aspect-oriented programming (AOP) language like AspectJ is that it is an amalgamated language in the following sense. Besides ordinary object-oriented expressiveness, one can also write code that superimposes advice onto points along the execution of object-oriented functionality. An important class of join points are method calls. One can give a precise definition of this sort of AOP on the basis of a formal semantics [8]. (There are further models of AOP, e.g., the Hyper/J-like model, which we will not address in this paper.)

Question: Can the perception of 'superimposition' be captured in a language-parametric manner, that is, without talking about method calls, or other constructs, without commitment to a specific language? There has been work on studying some forms of superimposition (say, AOP) at a fundamental level [3, 6, 1, 4] on the basis of specific computational models. However, we seek an approach that applies immediately to actual programming languages.

In this extended abstract, we describe a general method to add superimposition to the denotational semantics of a language. The overall approach is described in Sec. 2. An illustrative example is developed in full detail in Sec. 3. The design space for our form of superimposition is briefly scanned in Sec. 4. Related work is discussed in Sec. 5, and the paper is concluded in Sec. 6.

2 The overall approach

Suppose we consider a semantic function of the following type:

$$[\![\cdot]\!] : S_i \to D_i$$

Here, S_i is a syntactical domain, and D_i is the associated domain of semantic meanings, say *denotations*. Semantic functions are specified using case discrimination on S_i while the meaning of a syntactical form is expressed in terms of the meanings of its subterms (aka compositionality). Let us consider a specific semantic equation. Without loss of generality, we pick up an equation for a construct with one subconstruct:

$$[\![C\ x]\!] = f\ [\![x]\!]$$

Here, $C\ x$ is the syntactical pattern at hand, and f is an operation that turns the denotation of x into the denotation of $C\ x$. (We use curried function application.) Adding superimposition is now performed in three steps:

1. The semantic functions are elaborated to carry advice.
2. Denotations are made 'reflective' so that they can be altered.
3. A construct for binding advice is integrated.

The type of the above semantic function is adapted as follows:

$$[\![\cdot]\!] \quad : \quad S_i \to R\ D_i$$

Here, R is a domain constructor to add a *registry* with superimposed advice to domains of ordinary denotations. There are a few options for the actual type of the registry. In essence, the registry is a transformer for reflective denotations. To obtain a modular semantics, we assume that R is a monad. Depending on the binding policy and other language-design decisions, R could be the environment monad, the state monad, or a combination of both.

The semantic equation from above is adapted as follows:

$$[\![C\ x]\!] \quad = \quad (C\ x)\ \mathbf{:\text{-}}\ ([\![x]\!] \gg\!= \lambda x'.\ return\ (f\ x'))$$

The occurrences of "$\gg\!=$" and *return* point to monadic style. (We use the common convention that the monad operator, which lifts values to computations, is denoted by *return*, whereas the monad operator, which applies a value consumer to a computation, is denoted by "$\gg\!=$".) The semantic equations can be turned into monadic style by a systematic transformation. It is avoidable to introduce monadic style if we fix the binding policy for advice, e.g., by explicitly passing around an environment for the registry.

The occurrence of the infix operator " **:-** " in the above equation points to reflective style. Both the operator " **:-** " and this use of the term 'reflective style' are inventions of the present paper. The operator " **:-** " shapes the reflective denotation as follows. If d is the original denotation for a term t, and d_m is d in monadic style, then the reflective denotation is of the form t **:-** d_m. This is read as 'by default, at t do d_m'. There are a few options for the precise definition of " **:-** ". In essence, the operator does nothing but application of denotation transformers as provided by the registry. That

is, given the registry r, the term t, and the denotation d, the operation t **:-** d applies r to t and d. So basically, each semantic equation can be revised via the registry r.

It remains to perform the third step from above, that is, the integration of a construct for binding advice. This comes down to adding one equation for $[\![\cdot]\!]$. There are a few options depending on the favoured binding policy and other language-design decisions. One option is to hold superimposed advice in an environment with a binding scope that is local to a given program fragment. That is:

$$[\![h \textbf{ adapts } t]\!] \quad = \lambda h'.\ [\![t]\!]\ h$$

We call h and h' *hooks*. The old hook h' is replaced by the new hook h. We say that h *adapts* the program fragment t. The denotation transformers accomplished by h will transform the denotations determined for t and its subconstructs. (We try not to use the AOP term 'advice' for h because h accomplishes both the advice code and the definition of join points or pointcuts.) Hooks and registries are of the same type: a family of denotation transformers indexed by the syntactical domains. For example, the denotation transformer for S_i is of type $S_i \to R\ D_i \ \to \ R\ D_i$. The first argument of type S_i emphasises that reflective denotation transformers can look at the program text to make a decision.

Our method works, in principle, for all possible language semantics (such as imperative languages, different object-oriented languages, functional languages, etc.). However, language-specific properties and obligations occur in this process. Those have to be studied to arrive at a useful notion of language-parametric superimposition.

In the present paper, we will only deal with dynamic semantics. It is desirable to couple the adaptation of dynamic and static semantics. In the ideal situation, type safety of programs that involve superimposition should hold by construction. We might actually want to amalgamate static and dynamic semantics to be able to formulate certain kinds of pointcuts for superimposition that deal with types.

From here on, we will represent semantics definitions in Haskell. This immediately allows us to run these definitions as interpreters.

3 An example

We will now illustrate how to make the semantics of a very simple expression language fit for superimposition, and how to make use of superimposition. The language comprehends expression forms for integer constants, variables, and binary arithmetic expressions. Superimposition will be used in a way to catch division by zero.

In Fig. 1, the semantics of the expression language is defined in the denotational style. We define a type of denotations for each syntactical category, namely *Dexpr* for *Expr*, and *Dbinop* for *Binop*. The type *Dexpr* expresses that the meaning of an expression is a mapping from environments to values. The environment maps variable identifiers (i.e., strings) to values. The type *Dbinop* expresses that the meaning of a binary operator is a function that maps two values to a single value. We define one meaning function for each syntactical category using case discrimination according to the syntactical forms. The definition is compositional, that is, the meaning of a construct is defined in terms of the meanings of its subconstructs only, but not the subconstructs themselves.

We will now add superimposition to the semantics according to the three steps listed in Sec. 2. The first step is to elaborate the semantic functions to carry advice. One way to realise this step is to perform 'monad introduction', that is, to migrate to monadic style. This step is independent of the fact that we deal with semantics and superimposition. In principle, any family of recursive function definitions can be turned into monadic style. In [7], we define such a transformation. So the denotation types in the types of the semantic functions have to be wrapped by a monad. In the semantic equations, all compound denotations are sequentialised, and they are recomposed by "$\gg\!=$". Without loss of generality, we assume a call-by-value order. The result of this step is shown in Fig. 2. Notice the elaborated types of the semantic functions, which involve a monad m. Also notice the monadic *do*-sequence for the compound meaning of a binary expression, and several occurrences of *return*. Instantiating the monad m by the identity monad, and β-reducing away sequentiality, we would obtain the original interpreter.

```
Expression syntax
  data Expr   =  Const Int | Var String | Bin BinOp Expr Expr
  data BinOp  =  Div | ···

Denotations
  type Dexpr  =  Env → Val
  type Dbinop =  Val → Val → Val

Variable environments
  type Env    =  String → Val

Values incl. an error value
  type Val    =  Maybe Int

Expression evaluation
  expr :: Expr → Dexpr
  expr (Const int) ρ    =  Just int
  expr (Var id) ρ       =  ρ id
  expr (Bin o e1 e2) ρ  =  binop o (expr e1 ρ) (expr e2 ρ)

Interpretation of binary operators
  binop :: BinOp → Dbinop
  binop Div (Just x) (Just y)  =  Just (x `div` y)
  binop ...                    =  ...
  binop _ _ _                  =  Nothing
```

Figure 1. Denotational semantics of a simple expression language. Because it is a Haskell program it can be viewed as an interpreter as well.

```
Evaluation function
  expr                 ::  Monad m ⇒ Expr → m Dexpr
  expr (Const int)     =   return (λρ → Just int)
  expr (Var id)        =   return (λρ → ρ id)
  expr (Bin o e1 e2)   =   do de1 ← expr e1
                              de2 ← expr e2
                              dop ← binop o
                              return (λρ → dop (de1 ρ) (de2 ρ))

Interpretation of binary operators
  binop                ::  Monad m ⇒ BinOp → m Dbinop
  binop Div            =   return (λv1 v2 →
                                 case (v1, v2) of
                                   (Just x, Just y) → Just (x `div` y)
                                   _ → Nothing)
  binop ...            =   ...
```

Figure 2. The interpreter in monadic style

```
Expression evaluation; adapted
  expr, ōverline{expr}  ::  (Superimposable m Expr Dexpr,
                             Superimposable m BinOp Dbinop)
                        ⇒  Expr → m Dexpr
  expr (Const int)      =  return (λρ → Just int)
  expr (Var id)         =  return (λρ → ρ id)
  expr (Bin o e1 e2)    =  do de1 ← ōverline{expr} e1
                              de2 ← ōverline{expr} e2
                              dop ← ōverline{binop} o
                              return (λρ → dop (de1 ρ) (de2 ρ))
  ōverline{expr} e      =  e :- expr e

Interpretation of binary operators; adapted
  binop, ōverline{binop}  ::  Superimposable m BinOp Dbinop
                          ⇒  BinOp → m Dbinop
  binop Div               =  return (λv1 v2 →
                                case (v1, v2) of
                                  (Just x, Just y) → Just (x `div` y)
                                  _ → Nothing)
  binop ...               =  ...
  ōverline{binop} o       =  o :- binop o
```

Figure 3. The interpreter with reflective denotations

The second step in our procedure for adding superimposition to a semantics is to turn the semantic equations into reflective style by invoking the " **:-** " operator prior to case discrimination. For each semantic function, we define an overlined version that adds the application of " **:-** ", e.g., $\overline{expr}$ complements *expr*. (Alternatively, we could adapt *all* existing equations to invoke " **:-** " as described in Sec. 2.) In the semantic equations, all references to the original semantic functions are replaced by references to the overlined ones. The result of this step is shown in Fig. 3. Notice the new definitions of $\overline{expr}$ and $\overline{binop}$. Also notice the references to $\overline{expr}$ and $\overline{binop}$ in the semantic equations for *expr*.

Recall that the operator " **:-** " models transformation of reflective denotations. Since a language normally comprehends several syntactical domains and corresponding denotation types, the operator " **:-** " needs to be overloaded for all couples of syntactical domains and associated denotation types. So a registry is actually a *tuple* of denotation transformers — one for each denotation type. For a given denotation type, the operator is meant to look up the corresponding denotation transformer from the registry tuple and to apply it to the term and the denotation at hand. This can be conveniently represented in Haskell using a class for overloading. So we place the operator " **:-** " in a class *Superimposable*, which subclasses the standard class *Monad* as follows:

```
class Monad m ⇒ Superimposable m s d
  where
    (:-) :: s → m d → m d
```

There are three parameters: *m* is the type constructor of the monad for the registry, *s* is a syntactical domain, *d* is the type of denotations for *s*. We will see in a second that the instances of the *Superimposable* class follow a simple scheme.

The third step in our procedure for adding superimposition to a semantics is to integrate a construct for binding advice. This includes the obligation to opt for a specific instance of a *Superimposable* monad. We will now provide a general realisation of the third step including its illustration for the simple expression language. As always with our method, the language designer might bypass the language-parametric approach if a more language-specific form of superimposition is favoured.

```
The registry domain constructor
  type R = Reader Hook

Types of hooks for superimposition
  data Hook = Hook (Hexpr, Hbinop)
  type Hexpr = Expr → R Dexpr → R Dexpr
  type Hbinop = BinOp → R Dbinop → R Dbinop

The identity Hook
  idHook = Hook (λe ce → ce, λo co → co)

Run a reflective denotation
  run :: R d → d
  run d = runReader d idHook

Instances of Superimposable class
  instance Superimposable (Reader Hook) Expr Dexpr where
    t :- d = ask >>= λ(Hook (h, _)) → h t d
  instance Superimposable (Reader Hook) BinOp Dbinop where
    t :- d = ask >>= λ(Hook (_, h)) → h t d
```

Figure 4. Uniform definition of registry

```
Syntax of the superimposition construct
  data Expr = ... | Adapts Hook Expr

Semantics of the superimposition construct
  ōverline{expr} (Adapts h e) = local (const h) (ōverline{expr} e)
```

Figure 5. Uniform integration of a superimposition construct

In Fig. 4, we define a specific monad *R* that models a registry for superimposition in our example. In fact, we choose the environment monad (aka *Reader* in Haskell).[1] The type *Hook* is a product with two components, one for each denotation type. Each such hook component is a denotation transformer. The received denotation is the normal denotation, whereas the computed denotation is the revised, ultimate denotation. The types make clear that a denotation transformer also receives a syntactical entity, which can contribute to the decision whether to replace or to preserve the normal denotation. In the figure, we also define an identity hook (i.e., *idHook*), which models that the normal denotation is preserved regardless of the 'join point' (i.e., the syntactical form at hand). Running a reflective denotation is like 'running' the *Reader* monad with *idHook* as initial registry; see *run*. The last few lines in Fig. 4 instantiate the *Superimposable* class for our example semantics. That is, " **:-** " is defined by looking up the denotation transformers from the environment and by applying the relevant transformer to the ingredients of the given reflective denotation.

In Fig. 5, we complete the extension of the sample semantics by adding a specific case to the semantic function for expressions. This new equation provides the most simple and uniform kind of a superimposition construct. The meaning of *Adapts h e* is that the hook *h* adapts the denotation for the expression *e* and all denotations for

[1] We recall the operations of the *Reader* monad:

```
ask    ::  m r                      -- read environment
local  ::  (r → r) → m x → m x      -- locally adapt environment
```

```
noDivByZero = Hook (λe d → d, noDivByZero')
  where
    noDivByZero' :: BinOp → R Dbinop → R Dbinop
    noDivByZero' Div d =
      do d' ← d
         return (λv1 v2 → case (v1, v2) of
                            (_, Just 0) → Nothing
                            _ → d' v1 v2
                )
```

Figure 6. A hook for superimposition to catch division-by-zero

subconstructs of *e*. The use of the *local* operator makes it as clear as crystal that the hook *h* is only used for the interpretation of *e*. The use of *const* makes clear that previous bindings will be replaced by the new hook. We will investigate alternatives in the next section.

In Fig. 6, we define a hook for catching division-by-zero for any interpretation of *Div*. To this end, the second argument of the denotation is checked to be "0", and if this is the case, then the error value *Nothing* is returned. Otherwise, the original denotation is retained. So finally, we can demonstrate superimposition in action. To this end, let us consider the following program together with an environment for the used variables:

```
myexp = Bin Div (Const 42) (Var "myvar")
myenv = λid → if id ≡ "myvar" then Just 0 else Nothing
```

Using the original denotational semantics as an interpreter for this program, we will obviously encounter a division-by-zero run-time error. Using the aspect-oriented interpreter, we can catch division by zero. The following program execution returns *Nothing*:

```
run (expr (Adapts noDivByZero myexp)) myenv
```

4 Design space exploration

We will now walk through a few locations in the design space for a language semantics with superimposition. This will further substantiate the generality of our method, and it will clarify how it can be customised for a specific language at hand.

Binding policies We will first discuss different binding policies for advice. The policy that we have seen above employs an environment to carry advice. Here the affected program fragment is explicitly part of the binding construct. Also, we favoured the replacement of previous bindings by the new binding. Both design decisions can be altered. We will first discuss cumulative advice binding as opposed to replacement semantics before. We will then discuss the use of a state for the registry as opposed to an environment before.

In Fig. 7, we provide a new definition of the *Adapts* construct; see Fig. 5 for the original definition. We *chain* the previous binding and the new binding (cf. "∘"). The new binding gets into control but if it wanted to resort to the standard denotation, it actually accesses the denotation as processed by the previous binding. It is now not too difficult to think of further binding policies. For example, we could favour denotation transformers with yet another denotation argument for the standard denotation prior to any adaptation by previous bindings. This way, newly installed hooks could abandon previously installed hooks.

```
Syntax of the superimposition construct
  data Expr = ... | Adapts Hook Expr

Semantics of the superimposition construct
  expr (Adapts (Hook (he, ho)) e) = local chain (expr e)
    where
      chain (Hook (he', ho')) = Hook (λe → he e ∘ he' e,
                                      λo → ho o ∘ ho' o)
```

Figure 7. Superimposition with cumulative advice binding

```
Registry-aware computations
  type R = State Hook

Values incl. Void for pure side effects
  data Val = ... | Void

Syntax of the superimposition construct
  data Expr = ... | HookUp Hook

Semantics of the superimposition construct
  expr (HookUp h) = put h >>= λ() → return (return Void)
```

Figure 8. Superimposition with a state for advice binding

In Fig. 8, we use the *State* monad as opposed to the *Reader* monad for the registry; see Fig. 4 for the original definition.[2] Here we assume that the language semantics at hand provides a notion of a purely side-effective computation. Hence, there is a designated result value *Void*. The construct for superimposition now also takes a different form because we do not list the affected program fragment, but we simply register advice along the execution of the program. So the construct for binding advice is of the form *HookUp h* with the intended semantics that the hook *h* as installed as registry at the time when the *HookUp* expression is executed. As one can see, the expression evaluates to *Void*. A problem with this approach is that the base semantics and the *do*-sequences for the introduced registry monad might accidentally disagree on the order of computation. We will come back to this problem in a minute.

Effect composition So we have seen that both an environment and a state make sense for carrying advice. Capturing this variation point in a monad parameter is a good idea because the superimposition level might even deal with further effects than just carried advice. For example, we might want to maintain dynamic join point information [13], or we might want to reflect on the success and failure of denotation transformation. Regardless of the choice monadic-style semantics vs. hard-wired effects, a discussion of the relationship between the superimposition level and the base semantics is in place.

By default, we assume that the semantics is made fit for carrying advice without looking at the denotation types. For example, even if the original semantics is already in monadic style, we can perform monad introduction. This will result in nested monadic style. In Fig. 9, this is illustrated for a variation on our expression language. The variation provides an *Assign* statement the semantics of which relies on the *State* monad for the variables in a program. The reflective denotation for an *Assign* expression is a nested monadic computation. At the top level, the computations for enabling super-

[2] We recall the operations of the *State* monad:

```
get  ::  m s          -- read state
set  ::  s → m ()     -- write state
```

```
Syntax extension for assignments
  data Expr  =  ... | Assign String Expr

Revision of expression denotations
  type Dexpr  =  State Env Val

Semantics of assignments
  expr (Assign id e') = do de' ← exprˉ e'
                           return (do ρ ← get
                                      v ← de'
                                      put (λid' → if id ≡ id'
                                                  then v
                                                  else ρ id')
                                      return v)
```

Figure 9. The *Superimposable* monad on top of a base monad

imposition are arranged in a *do*-sequence. The inner *do*-sequence directly models the semantics of assignment. That is, the state is looked up with *get*, the right-hand side of the assignment is evaluated to *v*, the state is updated in the point for the variable *id*, the updated state is '*put* back', and the right-hand side value *v* is *return*ed as the value of the assignment.

These nested *do*-sequences pinpoint a problem. Suppose, we use a state for the registry; recall Fig. 8. A subexpression e_1 might hook up another subexpression e_2 while e_2 would be normally executed before e_1. That is, the nested *do*-sequences could disagree on the order of computation. Note that the inner sequence represents the base semantics whereas the outer sequence was established by systematic monad introduction. To enforce a common order, we should transform the monad in the base semantics to integrate the registry or any other superimposition effect as well. We could even elaborate an existing effect in the base semantics, e.g., an environment or a state, so that it accomplishes the registry as well. If the base semantics is not in monadic style, then it is not really prepared for such an amalgamation of effect spaces. In the view of these problems, our earlier choice of an environment monad for the registry is more favourable. The ordering problem is here a non-issue because advice binding is local with respect to a given term.

Intercepting invocations Our approach allows us to intercept *any* point of the program execution in the sense of syntactical fragments. It is at the heart of AOP to intercept invocations of methods or other procedural abstractions. So we want to briefly examine how this looks like in our setting. In Fig. 10, we further extend our expression language to accomplish a form of named function application. The semantic equation for function application is already prepared to carry advice. (We again use nested monadic style.) We use a helper function *apply* to apply a function-type value to an argument. For brevity, we do not define any expression form for function abstraction (i.e., λ-abstraction), but we assume that the environment can hold functions, e.g., a function "*div*". At the bottom of Fig. 10, we define a hook *noDivByZero*, which intercepts applications of the "*div*" function to catch division by zero. Notice the plain use of pattern matching for filtering out the relevant (nested) function application. The constructed denotation returns *Error* if the second argument of "*div*" is "0", and otherwise it applies the original binding of "*div*". This hook looks a bit verbose because it reconstructs the normal denotation to a large extent. This could be captured by a reusable operator for 'function-application interception'.

```
Syntax extension for function applications
  data Expr  =  ... | Apply Expr Expr

Functions evaluate to functions
  data Val  =  ... | Fun (Val → Val)

Adapted semantics of function application
  expr (Apply e1 e2) = do de1 ← exprˉ e1
                          de2 ← exprˉ e2
                          return (do v1 ← de1
                                     v2 ← de2
                                     return (apply v1 v2)
                                 )

Helper for function application
  apply :: Val → Val → Val
  apply (Fun f) val = f val
  apply _ _ = Error

Another division-by-zero catcher
  noDivByZero :: Expr → R Dexpr → R Dexpr
  noDivByZero (Apply (Apply (Var "div") e1) e2) d =
    do de1 ← exprˉ e1
       de2 ← exprˉ e2
       return (do ρ ← get
                  v1 ← de1
                  v2 ← de2
                  case v2 of
                    Int 0 → return Error
                    _ → return (apply (apply (ρ "div") v1) v2)
               )
  noDivByZero _ d = d
```

Figure 10. Intercepting a function application

5 Related work

For distributed systems (of communicating processes), there exists a notion of superimposition [3, 6, 10] which is (like aspects in AOP) orthogonal to the usual breakdown of modules. This sort of superimposition contributes to the theoretical basis of AOP. Another abstract, formal model of AOP is provided in [4]. It is based on execution monitors for the events that correspond to the points of interest along the program execution. Another formal semantics of AOP is based on CSP with CSP synchronisation sets as join points [1]. All this work differs from ours in that we start from an ordinary denotational semantics, and make it fit for superimposition in a systematic manner. That is, we do not resort to any designated formal model, but we just stay in the denotational setting.

Our approach to reflect on the syntactical patterns along program interpretation is inspired by the event grammars in [2]. Auguston suggests to formalise the execution of a program in a language in terms of an event grammar. Such a behavioural model can then be used to superimpose functionality on the event traces of a program, e.g., to check assertions, or to perform debugging. This approach has been used in the development of several debugging tools. Our notion of 'reflective denotations' is a semantic transposition and a strong generalisation of a tweaked monadic-style of functional programming proposed by Meuter in [9]. In this style, the programmer informs a non-standard monadic bind combinator about the names of functions that are applied to intermediate results. These names can be viewed as (explicit) join points. By contrast, we prepare a

semantics in a way that a 'superimposable' monad can revise denotations for syntactical patterns. Our method is based on a systematic transformation as opposed to an encoding style.

There is an enormous amount of related work on reflection [11, 12, 5]; its relevance for AOP is generally acknowledged. We have not seen a generic method to systematically elaborate a denotational semantics for AOP-like reflection in the available literature. The reflection literature is normally concerned with some kind of staged interpretation as opposed to the provision of a superimposition construct. However, it seems that our approach could take great advantage of the reflective theory for the purpose of formal reasoning on aspect-oriented programs. Also, ideas on 'full computational reflection' are of use to further generalise our approach.

6 Concluding remarks

The described method defines how to extend an ordinary language *semantics* so that one obtains an aspect-oriented language semantics. We call this achievement '*superimposition for free*'. Technically, it is based on a 'reflective denotation style'. Accidentally, the approach also suggests a normative style of *aspect-oriented functional programming*, but this has to be discussed elsewhere. The aspect-oriented programming terms are instantiated for 'superimposition for free' languages as follows:

Static join point	=	Syntactical pattern
Dynamic join point	=	Computation on syntactical pattern
Point cut	=	Pattern matching predicate
Advice	=	Denotation transformer
Program execution	=	Monadic do-sequence
Aspect	=	Hook for superimposition
Dynamic weaving	=	Registry update
Static weaving	=	Partial evaluation

We contend that this is a rather simple, uniform, and general way to define aspect-oriented language semantics. We are also willing to say that our approach can be seen as another definition of reflection.

The to-do list for an exhaustive treatment of the subject is long:

- Transpose the method to static semantics.
- Cover the standard forms of dynamic join points.
- Make even the denotation transformers reflective.
- Recover compositionality in some way.
- Cover SOS in addition to denotational semantics.
- ...

Acknowledgement

I am very grateful to the three anonymous FOAL 2003 workshop referees for their encouraging and constructive remarks. I am also grateful to the participants of the Belgian-Dutch poster workshop on AOSD, on 21 January 2003 in Twente, with whom I had stimulating discussions on the subject of the paper.

7 References

[1] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209, Berlin, Heidelberg, and New York, Sept. 2001. Springer-Verlag.

[2] M. Auguston. Program behavior model based on event grammar and its application for debugging automation. In M. Ducassé, editor, *AADEBUG, 2nd International Workshop on Automated and Algorithmic Debugging*, pages 277–291, Saint Malo, France, 22–24 May 1995. IRISA-CNRS.

[3] L. Bougé and N. Francez. A compositional approach to superimposition. In ACM, editor, *Proc. of the 1988 conference on Principles of programming languages (POPL'88), January 13–15, 1988, San Diego, CA*, pages 240–249, New York, NY, USA, 1988. ACM Press.

[4] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*, pages 170–186. Springer-Verlag, Sept. 2001.

[5] S. Jefferson and D. P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2/3):181–202, May/June 1996.

[6] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, Apr. 1993.

[7] R. Lämmel. Reuse by Program Transformation. In G. Michaelson and P. Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.

[8] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[9] W. D. Meuter. Monads as a theoretical foundation for AOP. In S. Mitchell and J. Bosch, editors, *Workshop Reader, ECOOP'97*, volume 1357 of *LNCS*. Springer-Verlag, 1998.

[10] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 28–40. ACM Press, 2002.

[11] B. C. Smith. Reflection and semantics in lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, ACM, Jan. 1984.

[12] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In R. P. Gabriel, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 298–307, Cambridge, MA, Aug. 1986. ACM Press.

[13] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Report 02-06, Dept. of Comp. Sc., Iowa State Univ., pages 1–8, Apr. 2002.