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

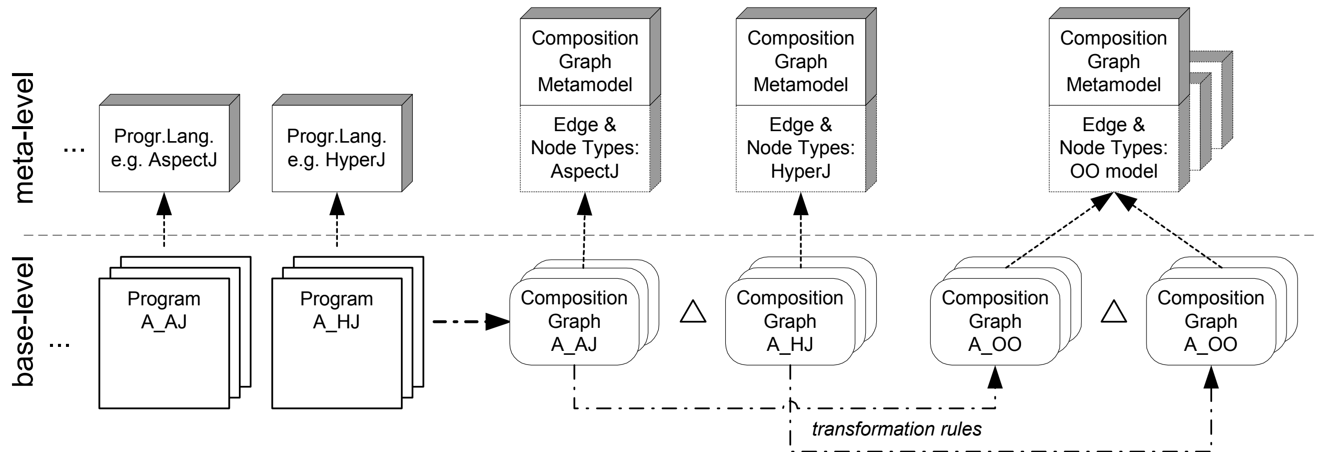
4. Our Approach

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.

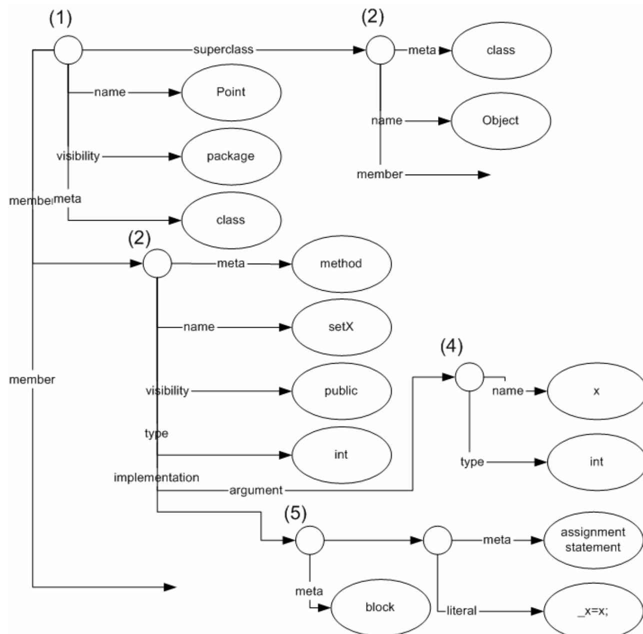


Figure 5. Part of class Point represented as a Composition Graph

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.

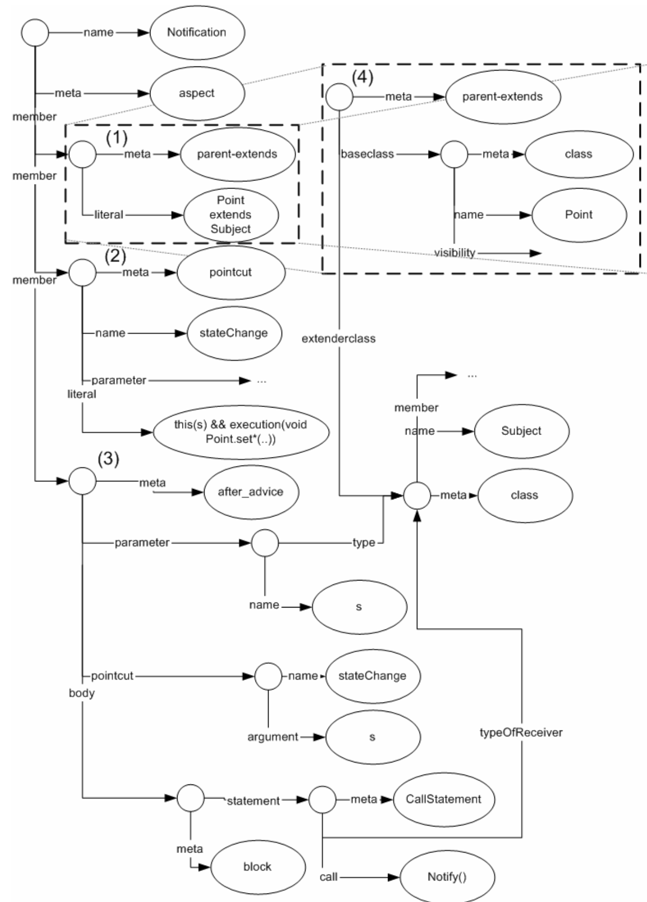


Figure 6. Part of the aspect *Notification* shown as a CG

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.

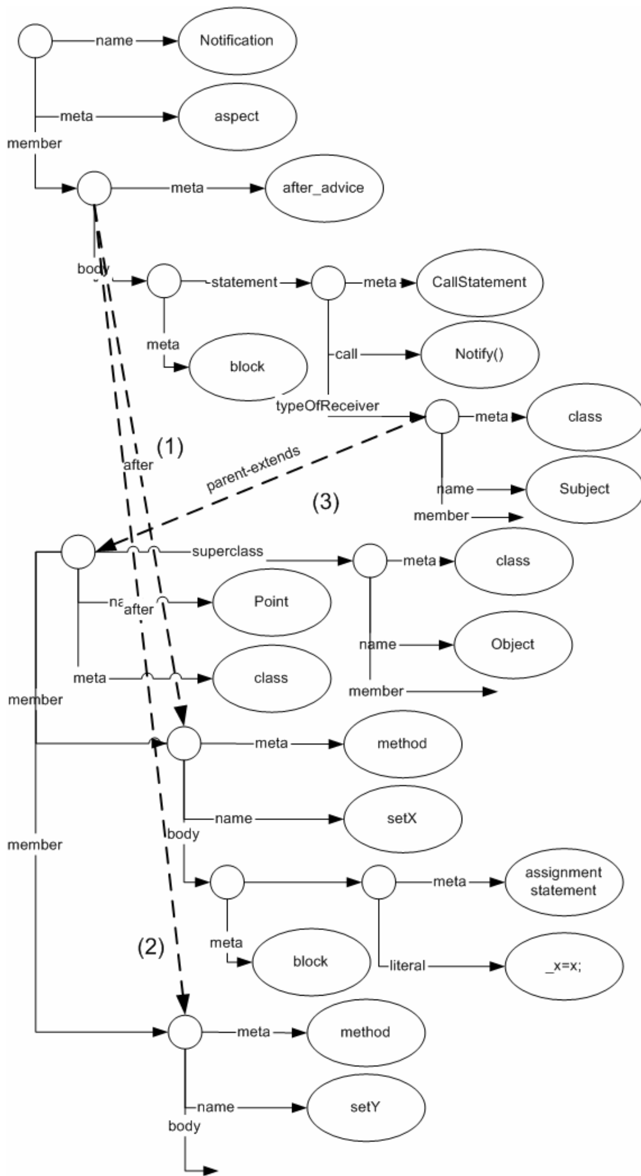


Figure 7. CG of the AspectJ program after the third step

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.

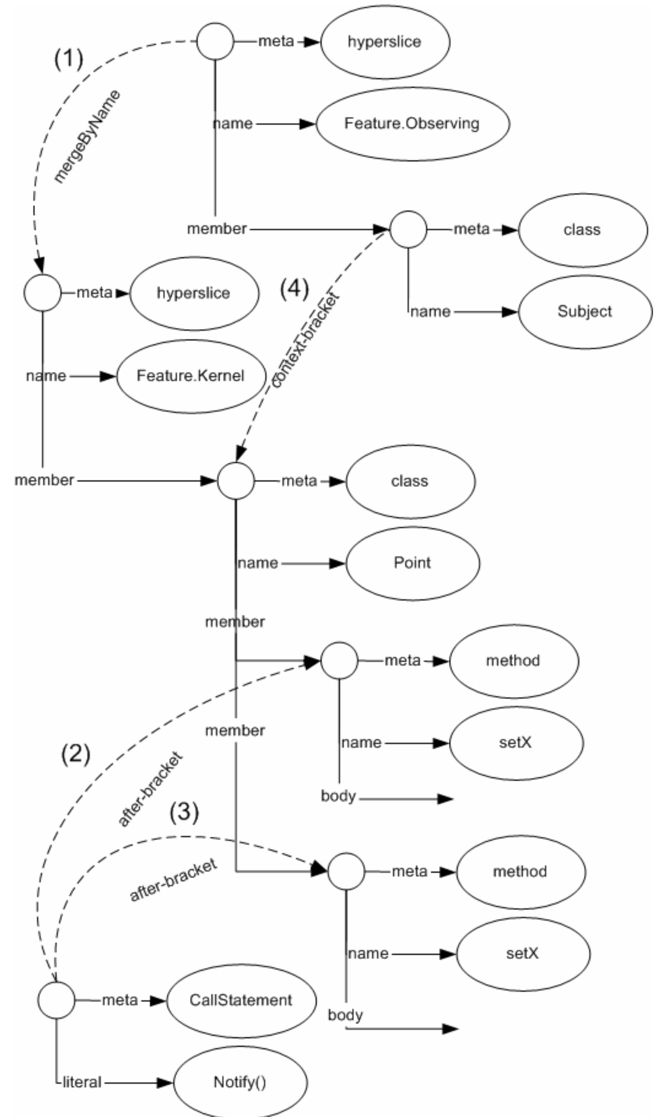


Figure 8. CG of the HyperJ program after the third step

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.

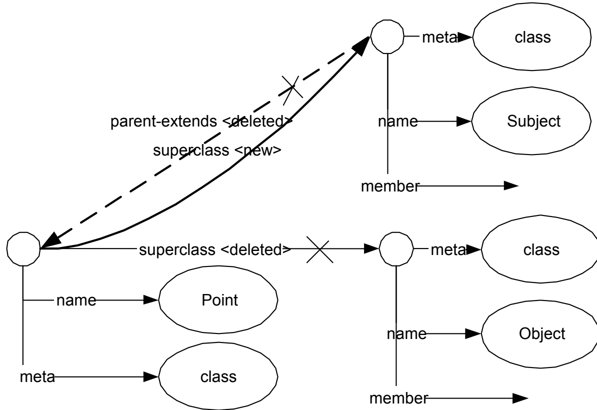


Figure 9. Transformation of the introduction statement

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.

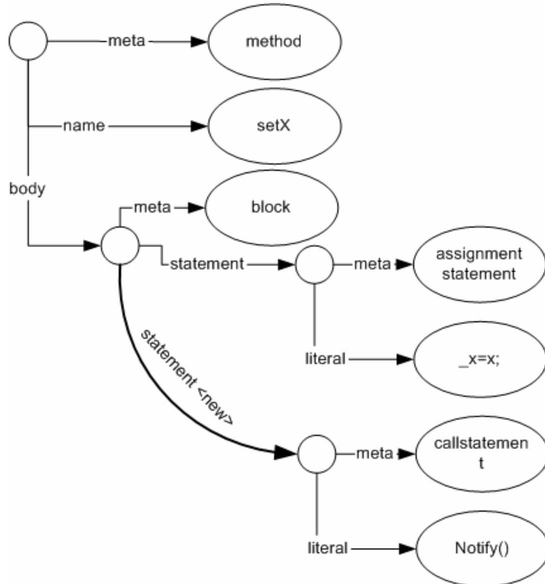


Figure 10. Method *setX* after the transformation

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 | X.meta = class 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 | Y.name=setx AND ∃X∃Z(X.member→Y AND X.superclass→Z 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.