Recognizing Join Points from their Context through Graph Grammars

Walter Cazzola
DICo, University of Milano, Italy
cazzola@dico.unimi.it

Stefano Salvotelli
DICo, University of Milano, Italy
stefano.salvotelli@studenti.unimi.it

ABSTRACT

Aspect-oriented software development has been proposed with the intent of better modularizing object-oriented programs by confining crosscutting concerns in aspects. Unfortunately, the aspects do not completely keep their promises. Most of the current approaches revealed to be tightly coupled with the base-program's code compromising the modularity. Moreover, the feasible modularization has a coarse-grain since the aspects can only be woven at the public interface level but not on a generic statement. We have designed the Blueprint framework to overcome these limits. The join points are located through the description of the context where they could be found. This work is about the framework realization and the role that graph grammars play in locating the join points in the base-program from the context description.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors—*Parsing*.

General Terms: Languages.

Keywords: AOP, Join Point Selection, Graph Grammars.

1. INTRODUCTION

Aspect-oriented techniques are widely used to better modularize OO programs by introducing *crosscutting* concerns in a safe and non-invasive way. Each aspect-oriented approach is characterized by a *join point model* consisting of the *join points* (well defined points in the computational flow), a mechanism for selecting the join points (*pointcuts*) and a mechanism for raising effects at the join points (*advice*). Crosscutting concerns might be poorly modularized as aspects without an appropriate join point model that covers all the interested elements. In most of the aspect-oriented approaches, the pointcut definition language allows the programmer to select the join points on the basis of the program's lexical structure, such as explicit program element names. This dependency renders the pointcut definitions fragile [5] and hinders aspects reusability and evolvability [1, 3] since they are tailored on the base-program. Moreover, the join points are defined at the *operation level*. It implies that the possible set of join points includes every operation that the system performs. Whereas, in many contexts we wish to define aspects that are expected to work at the *statement level*, i.e., by considering as a join point every point between two generic statements. Several attempts [3, 4] to overcome these issues have been investigated. It is widely recognized [3, 4, 6] that the solution lies on a more semantic approach that could exploit the base-program's design information. On these considerations, we have designed the Blueprint framework [2].

AOM'09, March 2, 2009, Charlottesville, Virginia, USA.

2. A GLIMPSE AT THE BLUEPRINTS

The key idea behind the Blueprint approach consists of describing where the join points could be through a "template" of the base-program's behavioral model without depending on its syntax. This permits to select the join points by describing their supposed position in the base-program's code through *patterns* on the base-program's behavior, called *join point blueprint* (blueprint for short). These blueprints do not describe the base-program's behavior rather they describe the desired properties and behaviors we are looking for. Due to its independence of the base-program's code, the blueprint cannot be a complete description of the base-program but just an abstraction of some significant parts. Each blueprint will be matched against the base-program and its context information will be unified with the base-program's concrete data.

The Blueprint framework recalls the AspectJ terminology with some slightly deviations. In our view, the *join point*s are *hooks where the code may be added* rather than *well-defined points in the execution of a program where effects can be raised*. In AspectJ, the considered join points are at the operation interface but a *join point could occur everywhere in the code not only at the operation interface* — the Blueprint exploits this concept. This view grants a statement-level granularity to the Blueprint join point model. The Blueprint approach allows to *loosely* describe the base-program's behavior. The aspect programmer can use different levels of detail to describe a single blueprint by using any possible combinations of *loose* and *tight elements*. So, it is possible to describe a well identified behavior tightly coupled to code by specifying the names of the involved elements as well as a less known behavior by using meta-information to abstract from the real code.

The blueprints are the key elements of the whole approach. They graphically depict where a join point should be in the base-program's behavior. They look like UML activity diagrams and behave similarly. Both represent part of the computational flow of the base-program. What differs is their use: the activity diagrams *model* the base-program's behavior whereas the blueprints depict where a join point should be in the base-program's behavior and structure.

In the rest of the section we describe the blueprints structure and potentiality by an example; for a complete overview of the blueprint's syntax refer to [2]. The example has been presented

in [1] and is based on the `HealthWatcher` application [9]; it consists of a simple blueprint (Fig. 1) that describes the algorithm to check if a value is already in the dictionary and to add it when missing. The selected join points can be used to force some synchronization policy on the dictionary access.

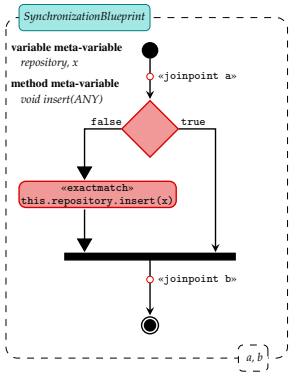


Figure 1: Case Study

Each blueprint is a diagram that contextualizes the join point location by describing some crucial *events* that should occur close to the join point and characterizing the context of the join point. In our case, it must describe a test followed by the insertion of the element in the dictionary when the test fails. Since the blueprints are decoupled from the base-program's code and structure, the context is expressed through meta-information that will assume different values depending on the code matched by the blueprint. In our example both the test and the insertion operation are meta-information; e.g., the `insert()` method is a *method meta-variable* that will be replaced by the real name used in the code during the weaving.

The piece of base-program's computational flow is described similarly to an activity diagram by actions and arrows connecting them; the context change is described by a swimlane. A blueprint can provide a loose or tight description of the computational flow; the latter is realized by *action*s (red action states labeled with the stereotype «`exactmatch`») and *tight transition*s (solid arrowhead transitions) and what they describe is exactly what we are looking for but the meta-information. The loose description, *template action*s (yellow action states) and *loose transition*s (open arrowhead transitions), is more a suggestion of what we are looking for and do not provide a strict sequence of how the events happen. In our example, we are interested in a specific computational path, i.e., the test/insert sequence that we tightly describe; on the other side, what happens when the test is passed is not relevant and loosely described. The join point location is denoted by the «`joinpoint name`» stereotype. In the example we have two join points located before and after the interested piece of code respectively.

3. JOIN POINTS OUT OF A BLUEPRINT

Given a blueprint, we have to determine if and where the described join points are in the base-program to weave the advice. This is equivalent to compare the blueprint to the control flow graph (CFG) of the base-program since both the blueprint and the CFG are graphs themselves. Unfortunately, their different level of abstraction renders clumsy to apply graph isomorphism algorithms as done in our first attempt [2]. Rather we consider more straightforward to exploit graph rewriting and graph grammars [8] to determine whether from the CFG can be derived the blueprint and if so where the join points are. Therefore, to find the join point locations can be summarized in deriving from the base-program's CFG the looked blueprint and deducting the join point position on the starting CFG. To this aim, we defined a well-formed graph grammar that rewrites CFGs into blueprints. A rewriting path from the CFG to the blueprint grants a match for the blueprint.

Of course, to go from the CFG to the blueprint is too expensive to be pursued due to the huge number of rewriting paths that will be generated. The idea is to proceed from the blueprint to a portion of the CFG by applying backward the production of the graph grammar. The yielded CFG locates where the join points are if any. During the rewriting process we also resolve all the meta-variables to the names in the CFG (*unification process*) as done by the Rekers-Schürr algorithm [7].

Please note that the programmer neither need to know the graph grammar nor which productions should be applied to match the blueprint. (S)He must exclusively draw the blueprint and this process will be automatically and transparently realized by the weaver.

3.1 A Graph Grammar for the Blueprints

The graph grammar introduced in this section permits to derive all the blueprints that can be expressed on the base-program's control flow graph. If a blueprint provided by the programmer is in the generated language we have a match.

The basic components of our graph grammar are: a finite set of terminals, a finite set of non-terminals, a set of starting axioms from which deriving the language and a finite set of productions.

Symbol	Description
	basic block
	condition
	control flow
	start and end

Table 1: non-terminals.

Symbol	Description
1-2	*template action*
	action
	loose transition
	tight transition
	fork (and/or)
	join
	condition
	swimlane
	loop
	start and end
	join point

Table 2: terminals.

Symbols. Since we are deriving the blueprints from the base-program's CFG, the symbols composing a CFG are non-terminals and the symbols composing a blueprint are terminals in our grammar.

Please note that these sets are fixed and they neither depend on the base-program nor on the specific blueprints.

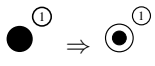
Axioms. Given that we have a control flow graph for each method, we consider the set of all the connected subgraphs of these CFGs as the set of the axioms for the graph grammar. Each CFG is initially decorated with the marker ⓪ that will be transformed in ①, ②, ③, ④, ⑤ during the rewriting process. Note that we are not building the whole axiom rather we expand the CFG for the method on demand when looking for the blueprint (see the method call productions).

3.1.1 The Productions of the Graph Grammar

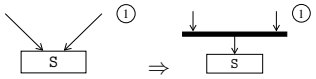
The productions of the graph grammars are grouped according to the desired application order and labeled by the marker. The markers automate the application of the productions during the rewriting process granting that productions in a certain group can be applied only if the current marker corresponds to their group. The markers do not grant that all the productions that could be applied have been applied when the marker changes; this check must be realized out of the rewriting process. The derivation process starts by applying the production ⓪ ⇒ ① that changes the marker to the first step.

Basic Productions.

End point. The CFG's end point is rewritten into the blueprint's end point.



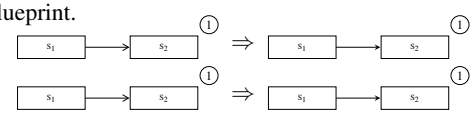
Basic Join. A join in the CFG (a couple of arrows entering into the same node S) must be expanded into a blueprint's join.



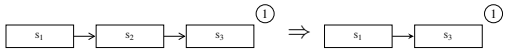
We reported the case where the node S is a generic basic block but the production applies to conditions and end points as well.

Sequence Productions.

One Step. The arrow between two basic blocks in the CFG can be rewritten both into a loose transition and into a tight transition in the blueprint.



Two Steps. Any combination of arrows (loose transition, tight transition and control flow) connecting three basic blocks can be rewritten into a sequence of two basic blocks (the first and the last of the original sequence) connected by a loose transition.



The above production shows one of the many productions related to the two-step sequences, the others are intuitive variations of this.

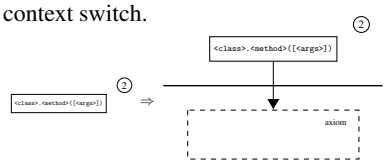
In all the reported productions:

- S_1, S_2 and S_3 are depicted as generic basic blocks but they could be conditions, join and start/end points as well;
- if the original nonterminal arrow was decorated with `true` or `false`, also the new arrow will be decorated after it;
- the loose transition can be decorated with a scope («`block`» and «`method`»); we do not report the productions.

Basic and *sequence productions* belong to phase 1. A production (① ⇒ ②) to step the marker forward is necessary.

Method Call Productions.

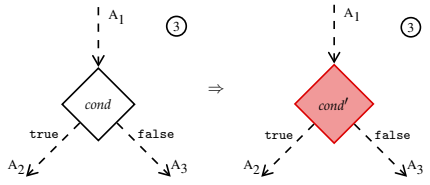
Swimlane. A basic block containing a method call can be expanded by connecting the CFG of the invoked method to this basic block. The introduced transition will be crossed by a swimlane stressing the context switch.



The *method call productions* belong to phase 2 of the rewriting process. A couple of productions to step the marker on and backward are necessary (② ⇒ ③ and ② ⇒ ①).

If and Loop Productions.

If. A condition block in the CFG can be rewritten in the equivalent terminal; the condition is rewritten in a more abstract condition with meta-variables, if necessary.

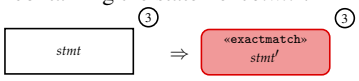


- A_1, A_2 and A_3 can be either loose or tight transitions, and do not change by applying the production;
- if A_1, A_2 or A_3 are missing they are not created by applying the production;
- the original condition *cond* is rewritten into *cond*′ by abstracting some elements and contextually declaring the needed meta-variables.

Since the loop rule does not add anything to the discussion, for sake of brevity it is omitted.

Action and Template Action Productions.

Action. A basic block containing a statement *stmt* can be rewritten in an action containing the statement *stmt*′.



stmt′ abstracts *stmt* by replacing all variables with meta-variables; the new meta-variables must be defined in the blueprint context.

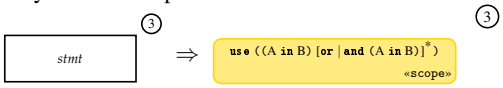
At this time, each action contains a single instruction or an abstraction of it but this production allows to merge a sequence of basic blocks in a single action in the blueprint when the basic block sequence is connected by tight transitions. The corresponding production is not reported for sake of brevity.

Template Action. A basic block containing a statement *stmt* can be rewritten in a template action containing a statement of type:

use ((A **in** B) [**or** | **and** (A **in** B)]*)

where A is a meta-variable and
B ∈ {`booleancondition`, `left`, `right`, `index`, `return`, `statement`}.

The newly generated template action must be decorated with a scope. According to the language definition, «`block`» and «`method`» are the only allowed scopes.



When we create a meta-variable, contextually we define it in the context in the same side of the swimlane where the involved node belongs.

If and loop and *action and template action productions* belong to phase 3; a production (③ ⇒ ④) to step the marker on is necessary.

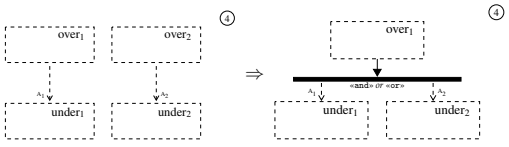
How to Abstract an Instruction or a Condition

Both *if and loop* and *action and template action productions* imply the abstraction of the content of the basic block or condition to match the vagueness of the corresponding blueprint element. The abstraction operation takes place after the following rules:

- the method arguments are replaced by `..` to neglect the number, the name and type of the arguments;
- a name is replaced by $*$ to represent every possible element without introducing new constraints;
- a name is replaced by a meta-variable name to couple the blueprint to the code; in this case the meta-variable name must be defined in the context as meta-informations.

Fork and Join Productions.

Fork. A couple of blueprints with a common prefix can be rewritten in a blueprint with a single occurrence of the common prefix and a fork operator decorated with «`and`» or «`or`»[1] forking on the distinct parts of the original blueprints.

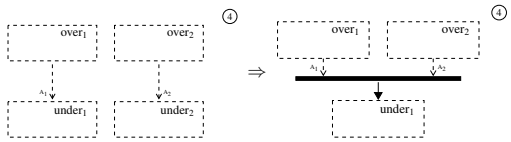


Note that,

- $over_1$ and $over_2$ are the common prefix and must coincide;
- A_1 and A_2 can be tight or loose arrows.

[1]Please note, that there are two distinct productions representing the two cases, but, to be brief, we describe them as one.

Join. A couple of blueprints with a common suffix can be rewritten in a blueprint with the distinct parts of the two original blueprints joining, through the join operator, into the common suffix.



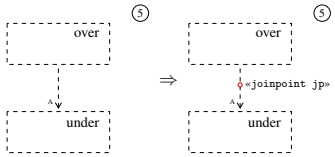
Note that,

- $under_1$ and $under_2$ are the common suffix and must coincide;
- A_1 and A_2 can be tight or loose arrows.

Fork and join productions belong to phase 4; a production (④ ⇒ ⑤) to step the marker forward is necessary.

Join Point Productions.

A blueprint can be rewritten by decorating one of its transitions by the local join point symbol. The decoration will be accompanied by a join point name and a scope information («source» or «target» if the join point must be matched at the beginning or at the end of the transition respectively).



To add the join points represents the last step in the rewriting process; after this, the markers must be removed.

3.1.2 *Some Notes about the Rewriting Process*

The rewriting process can be summarized in six phases (numbered from 0 to 5):

0 to choose an axiom, then
1 to apply the *basic productions* and the *sequence productions* until all the nonterminal arrows have been rewritten; then
2 to expand the necessary basic blocks by applying the *method call productions*; step 1 will be repeated on the expanded CFG;
3 to apply *if and loop productions* and *action and template action productions* with the side effect of getting out the meta-variable from the CFG; after this step the graph is a blueprint not a CFG/blueprint hybrid; then
4 to compose the generated blueprints by applying the *fork and join productions* when possible;
5 at last, to decorate the blueprint with the join points by applying the *join point productions*.

This separation is expressed by the markers used in the definition of the productions and automatically imposed by the rewriting process (look at Sect. 3.2).

3.2 The Rekers-Schürr Algorithm for Blueprint

To carry out the match between the blueprint and the CFG we adopt the Rekers-Schürr algorithm [7] for graphical parsing. This algorithm permits to find out if a graph belongs to the language generated from a given graph grammar and to determine which rewriting paths can generate it. In our case, it will permit to determine if a blueprint is a subgraph of the CFG by exploiting the graph grammar described in the previous section. The algorithm is divided in two phases named *bottom-up* and *top-down*.

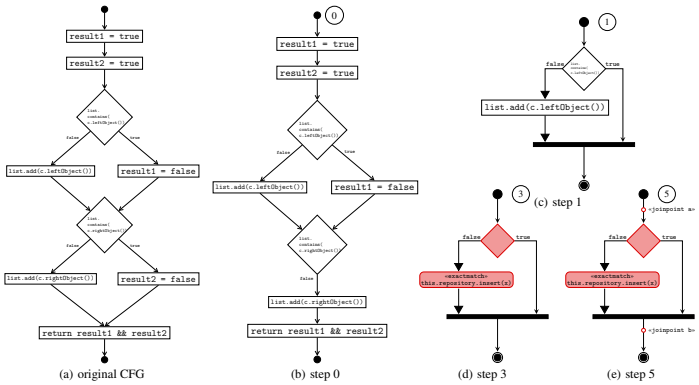


Figure 2: a) CFG and b-e) the axiom rewriting process.

The Bottom-up Phase. We look at the CFG for matches of the right-hand side of the grammar productions, respecting the markers. When a right-hand side is recognized, a production instance *pi* is created, and the elements that are in the left-hand side of the production but missing from the graph (to avoid endless cycles in the rewriting process) are added to it. The additions might in turn lead to the recognition of other right-hand sides. The result of the bottom-up phase is the collection *PI* of all production instances discovered. The productions instances created have dependency relations among each other, such as *above(pi, pi′)*, which means that *pi* should occur before *pi′* in a derivation, or *exclude(pi, pi′)*, which states that *pi* and *pi′* may not occur in the same derivation. These relations are computed during the bottom-up phase.

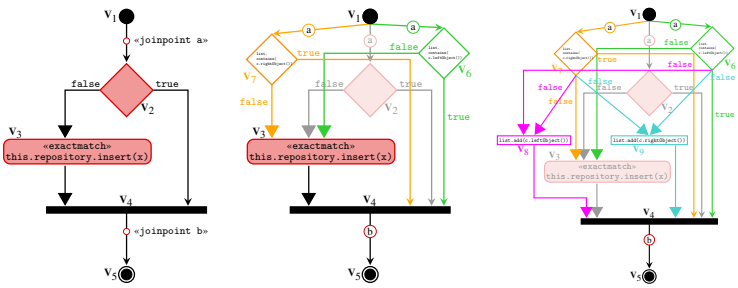
The Top-Down Phase. The dependency relations found in the first phase are used to direct this second phase. It starts with an empty graph and applies production instances of *PI* in such a way that the *above* and *exclude* relations are respected. By knowing all possible production instances and their dependency relations in advance, the top-down phase is able to postpone exploration of alternative derivation branches as long as possible. When necessary, these alternative derivations are developed in a pseudo-parallel fashion, with a preference for depth-first development.

3.3 Unification Process

The unification process maps the recognized blueprint back on the code through the CFG localizing the join points. To this respect, the rewriting path that has generated the blueprint is followed backward and, during the process, the meta-variables are unified to the names in the code to verify the correctness of the match.

In this process we must be careful on respecting the scope of the join points; in general we have some join points on some loose arrows that, during the backtracking process, will be expanded, so we must know if the join point must be placed over or under this expansion according to its scope.

5 due to the final aim of the unification process the join point productions are not inverted;
4 to decompose the previously merged blueprints by inverting the fork and join productions;
3 to apply the bottom-up phase of the Rekers-Schürr algorithm (Sect. 3.2) to find all the possible unifications. Note that, we must respect all the constraints defined by the meta-information during this phase. We also create a table, according to the algorithm, that describes all the possible inversions of the productions and the dependencies between them.
1 to apply the top-down phase of the Rekers-Schürr algorithm to the previously generated table and to determine the feasi-



(a) step 3a (b) step 3b (c) step 3c

Figure 3: Application of the Rekers-Schürr Algorithm

ble production paths by looking if there is a possible inversion for the *sequence productions*;

2 all the previous phases are applied to a single part of the blueprint delimited by the swimlanes: established a match, we restart the process for the portion of the blueprint below the swimlane.

At the end we have a subgraph of the original CFG with some join points on it and a unification for all the blueprint's meta-variables.

4. CASE STUDY

In this section, we show how the join points described by a blueprint are located inside a program. The example taken in consideration is the one reported in Sect. 2. We carry out our test on the following listing that presents at least a couple of matchings.

```
import java.util.LinkedList;

public class Set {
  private LinkedList list;
  public boolean add(Couple c) {
    boolean result1 = true, result2 = true;
«a»
    if (list.contains(c.leftObject())) result1 = false;
«b»
    else list.add(c.leftObject());

    if (list.contains(c.rightObject())) result2 = false;
«a»
    else list.add(c.rightObject());

    return result1 && result2;
«b»
  }
}
```

Now we will apply the productions as defined by the grammar step by step. Figure 2(a) shows the CFG for the `add()` method.

0 We consider the subgraph in Fig. 2(b) as the axiom to use in the algorithm.

1 Fig. 2(c) shows the graph after rewriting the axiom by applying the *basic productions* and the *sequence productions*.

2 In this example we do not need to expand any method.

3 Fig. 2(d) shows the graph after we have applied the *if and loop productions* and the *action and template action productions*; they have introduced the `repository` and `x` variable meta-variables and the *insert* method meta-variable.

4 The *fork and join productions* cannot be applied.

5 Finally, by applying the *join point productions* we decorate the graph with the «a» and «b» join points (Fig. 2(e)).

Now we will backtrack the productions application starting from Fig 2(e) by using the algorithm described in 3.3:

5 the join point are brought back to the original graph through the inversion of the other phases.

4 We do not need to invert any fork and join production.

3 Now we use the bottom-up phase of the Rekers-Schürr algorithm to match the found blueprint with the CFG and locate the join points in the code.

(a) We name all the nodes of the graph (Fig 3(a)).

(b) We add to the graph all the possible unifications of conditions and actions (Fig. 3(b) and Fig. 3(c)).

During this phase we fill the table of the possible production instances. We mark V_2 as a possible result of an *if production* that can be unified by `list.contains(c.leftObject())` generating V_7 and pi_2, or by `list.contains(c.rightObject())` generating V_6 and pi_1. V_3 is the result of an `action production` that can be unified by `list.add(c.leftObject())` generating V_8 and pi_3, or by `list.add(c.rightObject())` generating V_9 and pi_4.

pi	prod	left	common	right
pi_1	if	V_6		V_2
pi_2	if	V_7		V_2
pi_3	action	V_8		V_3
pi_4	action	V_9		V_3

1 To apply the top-down phase of the Rekers-Schürr algorithm, we have to identify the dependencies between the unifications to remove the non compatible productions. In our example, we find out that the only two allowed unifications are identified by the productions $\{pi_1, pi_3\}$ and $\{pi_2, pi_4\}$.

2 We do not have swimlanes so we have finished.

At the end of the matching process we have all the possible matchings, found by inverting all the production paths identified as valid. In particular, in the example we can find two possible matchings of our blueprint and consequentially of our join points.

5. CONCLUSIONS & FUTURE WORK

This work presents the Blueprint framework and a graph grammar based mechanism for locating the join points whose position is described by a blueprint. The work is just a first attempt that proves the feasibility of the approach. At the moment, we use the Rekers-Schürr algorithm without exploiting the specific peculiarity of the blueprint language but in the future we are going to optimize the approach as well as the graph grammar used.

6. REFERENCES

[1] W. Cazzola and S. Pini. AOP vs Software Evolution: a Score in Favor of the Blueprint. In *RAM-SE'07*, pp. 81-91, 2007.

[2] W. Cazzola and S. Pini. On the Footprints of Join Points: The Blueprint Approach. *J. of Obj. Tech.*, 6(7):167-192, 2007.

[3] A. Kellens, K. Gybels, J. Brichau, and K. Mens. A Model-driven Pointcut Language for More Robust Pointcuts. In *SPLAT'06*, Bonn, Germany, 2006.

[4] J. Klein, L. Hélouët, and J.-M. Jézéquel. Semantic-based Weaving of Scenarios. In *AOSD'06*, pp. 27-38, 2006.

[5] C. Koppen and M. Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proc. of EIWAS'04*, Germany, 2004.

[6] H. Masuhara, Y. Endoh, and A. Yonezawa. A Fine-Grained Join Point Model for More Reusable Aspects. In *Proc. of APLAS'06*, pp. 131-147, Sydney, Australia, Nov. 2006.

[7] J. Rekers and A. Schürr. A Parsing Algorithm for Context-Sensitive Graph Grammars. TR95-05, Leiden Univ., 1995.

[8] G. Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. 1997.

[9] S. Soares, E. Laureano, and P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proc. of OOPSLA'02*, pp. 174-190, Seattle, USA, Nov. 2002.