Aspect-oriented Programming as Model Driven Evolution

Mattia Monga
Università degli Studi di Milano
Dip. di Informatica e Comunicazione
Via Comelico 39, I-20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

Aspect-oriented programming (AOP) aims at managing crosscutting concerns at the programming language level. AOP is basically an evolution technique that may be used to augment a system with a new concern considered orthogonal to the others. The augmentation is applied automatically to a code base and is described with respect to a model of it. With AspectJ-like approaches this model has to be described as a set of join points, a solution that is in most cases too low level. Programmers should instead have the power of abstracting from the code base the model they prefer. Then, the augmentations described with respect to this specific model can be woven into the original system. Since the model introduced is specific to the concern the designer is trying to tackle, it may explicitly express design decisions, fostering safe evolutions.

1. ASPECT ORIENTATION À LA ASPECTJ

The notion of aspect-oriented programming was introduced by Kiczales et al. in [11]. Their approach was successfully implemented in AspectJ [1] by Xerox PARC. AspectJ aims at managing tangled concerns at the level of Java code. AspectJ allows for definition of first-class entities called `aspects`. This construct is reminiscent of the Java `class`: it is a code unit with a name and its own data members and methods. In addition, aspects may *introduce* an attribute or a method in existing classes and *advise* that some code is to be executed `before` or `after` a specific event (*join points*) occurs during the execution of the whole program. Aspect definitions are *woven* into the traditional object-oriented (Java) code at compile-time. Nevertheless, events that can trigger the execution of aspect-oriented code are run-time events: method calls, exception handling, and other specific points in the control flow of a program. In fact, integration, or weaving in the aspect-oriented jargon, can be in principle performed in different ways and at different times. In fact, it can be done at link-time [2], at run time [14], or at deployment-time [2].

AOSD – LATE 2005 Chicago, Michigan USA

The nature of the join points strongly affects the properties of the integration: its flexibility, the ability to understand the integrated system in terms of its components, reusability of components, and the nature and complexity of weaver and other supporting tools. In particular it affects the evolution of the system, since potential join points are an (implicit) interface to the aspect code. Any change in this interface might break the aspect code. Moreover, programmers should be very careful in writing aspects that make use of the implementation details of classes as little as possible if they want to be able to reuse their aspects. Also, it is still not clear how to cope with the difficult problem of multiple aspect interaction (see [8, 10, 5] for some work in progress and discussion).

In this paper I claim that aspect-oriented programming itself is basically an evolution technique that may be used to augment a system. However, the augmentation is applied automatically to a code base; thus, the augmentation should be described with respect to a model of this code base. With AspectJ-like approaches this model has to be described as a set of join points, a solution that is in most cases too low level. Instead, I propose to give programmers the power of abstracting from the code base the model they prefer. Then, the augmentations described with respect to this model can be woven into the original system. The paper is organized as follows: Section 2 defines aspect-oriented programming as an evolution technique; Section 3 presents a running example used in the following to explain the advantages of the approach; Section 4 sketches the proposed model-driven approach, by leveraging on graph transformations; Section 5 shows the feasibility of the approach on the running example, and, finally, Section 6 draws some conclusions.

2. AOP AS AN EVOLUTION TECHNIQUE

AspectJ is surely the most successful tool enabling aspect-oriented programming: however, several other approaches to separation of concern are normally considered to be aspect-oriented. Some (for example, [16, 13]) depart considerably from the concepts introduced in Section 1. Due to space restrictions it is not possible to discuss them here, but in order to avoid the common confusion between the end (separation of concern) and the means software engineers may use to pursuit it (aspect-oriented development), this paper would like to contribute in clarifying what is at the core of aspect orientation.

A relevant characteristic of aspect orientation is its enabling of *quantification* (doing something when a specific

property happens to be true [9][1]). Thus, the essential part of any aspect-oriented approach is the *two-step process*, by which quantification is exploited. In fact, aspects are inherently *a posteriori* with respect to some system they augment[2]. The system is described, designed, or implemented in any coherent way. However, some issues are more or less *orthogonal* to this description, design, or implementation. Better, designers would like to deal with some concerns *as if* they were orthogonal. Thus, in order to describe, design, and implement this augmentations, a model (an abstraction) of the system is considered. The original conception of the system and any augmentation are put together by some algorithmic device that, by knowing the ontology of the model, is able to weave augmentations to produce a running system. It is worth noting that, since the model is an abstraction of the system itself it is more general: as a consequence of this generality, an augmentation concerning a general element may also affect some entity that is not part of the original system. For example, if one states that "every function call should be traced by a `println`", the model of the system is the function call graph, and a universal quantification implies the affection of `println`s too; in order to avoid infinite recursion, one should state the augmentation as "every function call in the original system should be traced by a `println`".

In other words, aspect-oriented approaches are evolution techniques in which one defines a computation Δ that transforms a software system S in a new system S'. Δ can be defined by the tuple

$$< M, A, \mu, \omega >$$

where:

- M is a model of the system obtained by applying the abstraction function

  $$\mu : S \rightarrow M$$

  ,
- A is the augmentation of S defined on M
- ω denotes the weaving

  $$\omega : S, A \rightarrow S'$$

  .

According to this framework, an aspect-oriented approach can be described by stating the features of $< M, A, \mu, \omega >$. Thus, obliviousness about Δ becomes a property of the construction of S, irrelevant for the approach (while still significant when the properties of S and, in general, S' are concerned). Distinct aspect-oriented approaches may differ for the ontology and granularity of M (what can be quantified), for the mapping mechanisms μ (what is considered in mapping: which entities and which perspective, whether a static viewpoint is involved, or the system is considered in its run-time, link-time, deployment-time structure. This is a critical decision that heavily affects information hiding assumptions), for the ontology A (what can be part of the augmentation), and for the weaving mechanism ω.

[1]In order to distinguish aop from event-driven programming, Filman introduces the concept of *obliviousness*, for programmers are more or less unaware of what advises can be triggered by their code. Obliviousness is quite controversial among aspect orientation experts. A loosely regulated obliviousness as the one allowed by AspectJ constructs may cause turbulent *ripple effects* that in general force programmers to take into account most of the statements of a system in order to understand the properties of the system they are building. As a consequence, several scholars are proposing techniques to mitigate, discipline, or even avoid obliviousness in aspect oriented approaches [6, 3]

[2]I mean that the system about which aspects predicate is *conceptually* a priori, and not necessarily implemented before them

3. AN ASPECTJ EXAMPLE

In the following I will describe a simple yet typical use of AspectJ. The code is taken from [12]. In that paper the authors try to show that the AspectJ solution indeed improves the modularity of the design with respect to a classical object-oriented solution. The example implements a minimal graphical framework, as depicted in Figure 1. The important point is that whenever a Shape changes the code should signal the Display to update itself.

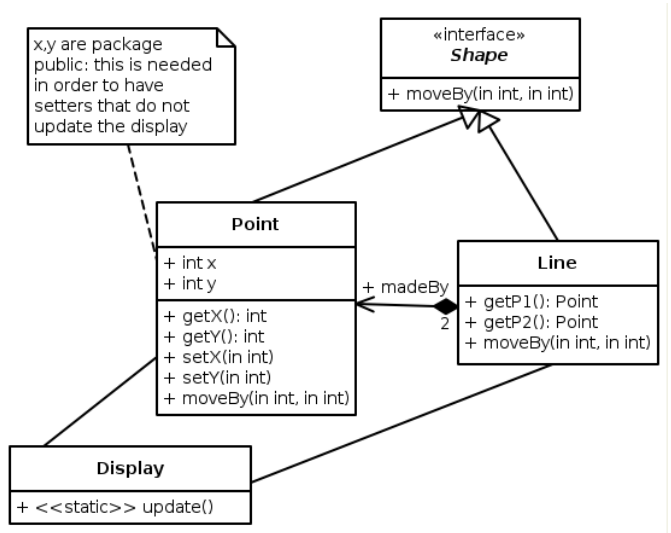


Figure 1: UML class diagram of the Shape example

A classical object-oriented solution would scatter the signalling code around the shape framework, thus tangling the shape-logic concern with the shape-show concern. The AspectJ solution (see Listing 1) isolates the updating part in an aspect: the advise is executed after that any operation that changes a Shape is performed. Relevant operations are listed in the pointcut definition.

An important design constraint is that the Display should be updated only once for each *top-level* change to the state of a Shape. Thus, the attributes `x` and `y` are made package public in order to enable transitory changes when the implementation requires them. For example, the `moveBy` operation of Lines is implemented by changing directly the coordinates of its points, otherwise (by using `setX` and `setY`) spurious `Display.update` would be triggered (see Listing 2).

Both the object-oriented and the aspect-oriented solutions cannot make clear this constraint in the code. Thus a modification in the `Line.moveBy` operation like the one shown in Listing 3 breaks the constraint with unpredictable effects.

In [12] the authors claim that global reasoning is required to discover the complete structure of update signaling, since

Listing 1: An aspect to update the Display

```
aspect UpdateSignaling{
 pointcut change():
   execution(void Point.setX(int))
   || execution(void Point.setY(int))
   || execution(
        void Shape+.moveBy(int, int));

 after() returning: change() {
  Display.update();
 }
}
```

Listing 2: Transitory changes during move

```
public void moveBy(int dx, int dy){
  getP1().x += dx;   getP1().y += dy;
  getP2().x += dx;   getP2().y += dy;
  // Display.update() needed
  // in the object−oriented solution
}
```

Listing 3: Transitory changes during move by setters

```
public void moveBy(int dx, int dy){
  getP1().setX(dx);    getP1().setY(dy);
  getP2().setX(dx);    getP2().setY(dy);
  // Display.update() needed
  // in the object−oriented solution
}
```

Listing 4: Revised advise to update the Display

```
after() returning: change()
   && !cflowbelow(change()){
 Display.update();
}
```

it is inherently a cross-cutting concern. However, the aspect-oriented solution, by isolating the concern make easier to change it to take into account a new problem. Thus, an elegant solution would be to update the Display only when no higher level changes are on going (see Listing 4).

Thus, the mapping μ operates at the level of the Java language, even overriding encapsulation. It is aware of most of the details of Java and it could, for example, distinguish between function calls and function dispatching, and take into account static and dynamic types. In other words, it works at the same level of abstraction used by the programmer of S, enabling a kind of *semantic patch* of the system since augmentations A are composed by every legal Java entity and every join point can be augmented before, after, or instead its detection. This is the source of several intricacies, since AspectJ programmers have to be aware of both Java and AspectJ subtilties to master their Δ.

4. REASONING AT A HIGHER LEVEL WITH EXPLICIT MODELS

In most cases, the maintenance of a system designed with an aspect-oriented approach à la AspectJ is quite difficult. By working at the level of the Java code, one is often forced to think about the woven code to understand what is needed to face a new challenge. Some authors (e.g.,[12]) claim that this is the very essence of cross-cutting: if one want to cope with a cross-cutting concern, s/he has to take into account the whole system (and, obviously enough, some sort of closed word assumption is needed). I agree with that: cross-cutting implies that one should consider the *system as a whole;* however, *not the whole system,* in all its details. Instead, an abstraction of the system should be considered. However, since the effectiveness of the abstraction depends on the concern one is working on, the aspect-oriented engine should provide some general mechanism that can be adapted to very different problems. In AspectJ the abstraction mechanism μ consists in the pointcut definition language that is both too low level and not very flexible, since it can express only abstractions about the lexical structure, the syntactic structure, or the run-time object model of the system, at the same level of the plain Java code itself.

In order to solve this problem, designers should be provided with a more general abstraction tool. The result of the abstraction is a model of the system (M) that is manipulated by building an augmentation (A). The final system (S') is obtained by enforcing (at compile-, link-, run-time) the super-imposition of A. Since very often models can be expressed by simple graphs, composed by labelled nodes and arcs, *graph transformations* seems a perfect candidate to be the mean for defining A. In fact, aspect weaving can be mapped naturally to graph rewriting [4]. Thus, the proposed approach can be summarized as follows:

1. a model M of the system is abstracted by applying an

opportune μ to S. If the system S is implemented, μ can be partially automated. M consists in a simple graph $< N, E >$, where N is a set of labelled nodes, and E a set of labelled arcs;

2. a graph transformation γ is defined, modelling the augmentation to applied to S: the left hand side of γ should be mapped on M in order to produce a new graph M';

3. an augmentation A is obtained by applying a concrete-making operation α to M';

4. eventually A is woven into S by the weaving engine ω.

The described approach decomposes the evolution of S in S' in the aggregated transformation $\mu \circ \gamma \circ \alpha \circ \omega$; this enables the analysis of the charactectistic of each step in order to infer which properties can be assumed to be preserved during evolution. However, to be really useful this approach required that α would be, as far as possible, an automatic transformation. On this regard some interesting techniques are described by [7].

5. THE EXAMPLE REVISED

The running example introduced in Section 3 is now tackled by exploiting the graph-grammar approach. Graph transformation rules will be described with single-pushout graph grammars, following the approach used by the tool Agg [15], that was used for implementing the example.

A software engineer wants to augment the system sketched in Figure 1 with the "Update Display" concern. She has to model the system in the context of this concern. A possible solution is depicted in Figure 2(a): the `Line` and `Point` classes are abstracted in two state diagrams. Objects of those classes, with respect to the display, can be only in two states: the `CHANGED` state represents the case when the display needs refresh, the `UPTODATE` state, instead, occurs when no display updates are needed. Some of the methods of the classes change the state of the objects, as represented by the transitions.

By leveraging on this model, the introduction in the system of the new concern "Update Display" can be described by a graph transformation rule as the one depicted in Figure 3. The rule states that if there is a Transition between a State `UPTODATE` and a State `CHANGED` due to an operation o, then an opposite transition "after returning o `Display.Update`" should be introduced in the diagram.

Moreover, the software engineer can now express *explicitly* also the constraint that the Display should be updated only once for each top-level change to the state of a Shape. A possible solution is shown in Figure 4: if the Operation x depends on Operation y and y is advised to `Display.Update`, then add a control flow condition stating that the advise should be activated only when y is not executed inside x. This rule is currently not useful, since no dependencies among operations are present. However, when the system will be evolved according to Listing 3, the model of the system will contain also the part (b) of Figure 2 and this rule will become relevant.

The (repeated) application of the above rules to the graph of Figure 2 produce a new model, depicted in Figure 5. This model can be translated in a concrete augmentation (see Listing 5) to be woven in the system.

Listing 5: Concrete augmentation to update the Display

```
after() returning: Point.moveBy(){
 Display.update();
}
after() returning: Point.setX()
  && !cflowbelow(Line.moveBy){
 Display.update();
}
after() returning: Point.setY()
  && !cflowbelow(Line.moveBy){
 Display.update();
}
after() returning: Line.moveBy(){
 Display.update();
}
```

6. CONCLUSIONS

The meaning of cross-cutting can be defined only if one assumes an existing structure. Thus, aspects are inherently *a posteriori* with respect to a base system. An aspect-oriented approach à la AspectJ forces designers to express aspects at the same level of abstraction of the base system. This means that an AspectJ aspect is no more than a semantic patch to the base system; thus in most case is very difficult to reason *at the aspect level,* since only a system level exists. The approach described in this paper aims at introducing a new level of abstraction needed when a cross-cutting concern is solved: the base is system is modelled with respect to the cross-cutting concern, manipulated with a graph grammar, and finally implemented in a concrete augmentation that can be woven in the system. Since the model introduced is specific to the concern the designer is trying to tackle, it may explicitly express design decisions, fostering safe evolutions.

Acknowledgements

The author would like to thank Katharina Mehner for her insightful comments on a first draft of this paper.

7. REFERENCES

[1] AspectJ. http://www.aspectj.org.

[2] AspectWerkz. http://www.aspectwerkz.org, 2004.

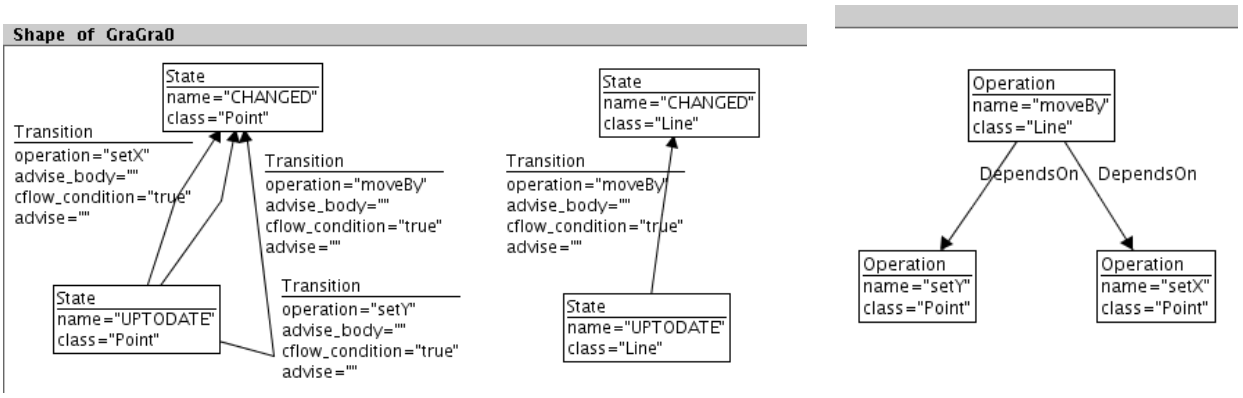
[3] J. Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. Technical Report CMU-ISRI-04-108, Carnegie Mellon University, 2004.

[4] U. Aßmann and A. Ludwig. Aspect weaving by graph rewriting. Technical report, June 1999.

[5] D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 25–29, Lancaster (UK), Mar. 2004. Iowa State University.

[6] C. Clifton and G. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report TR 03-15, Iowa State University, 2003.

[7] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating java code to graph transformation



(a) (b)

Figure 2: Model

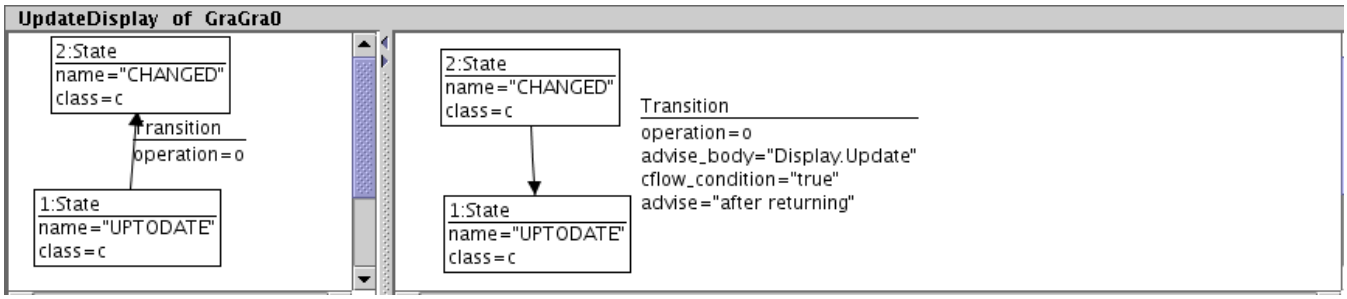


Figure 3: Rule UpdateDisplay

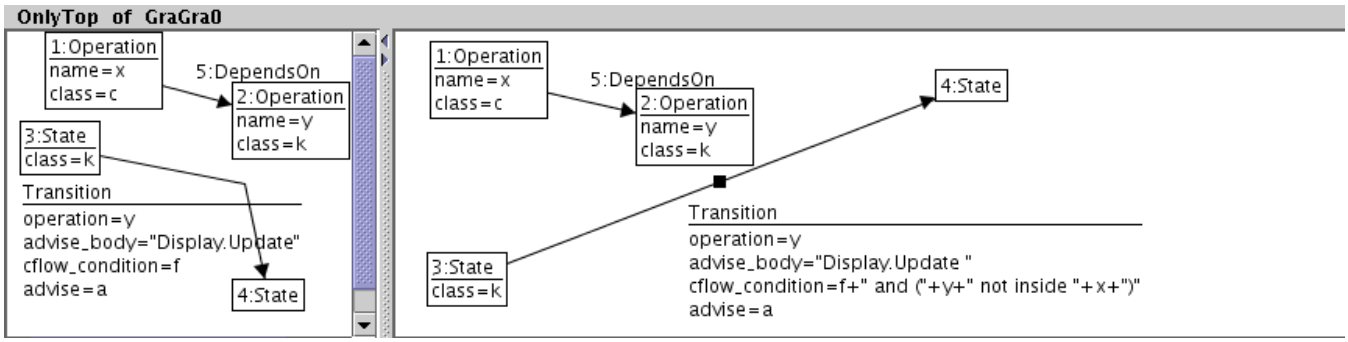


Figure 4: Rule OnlyTop

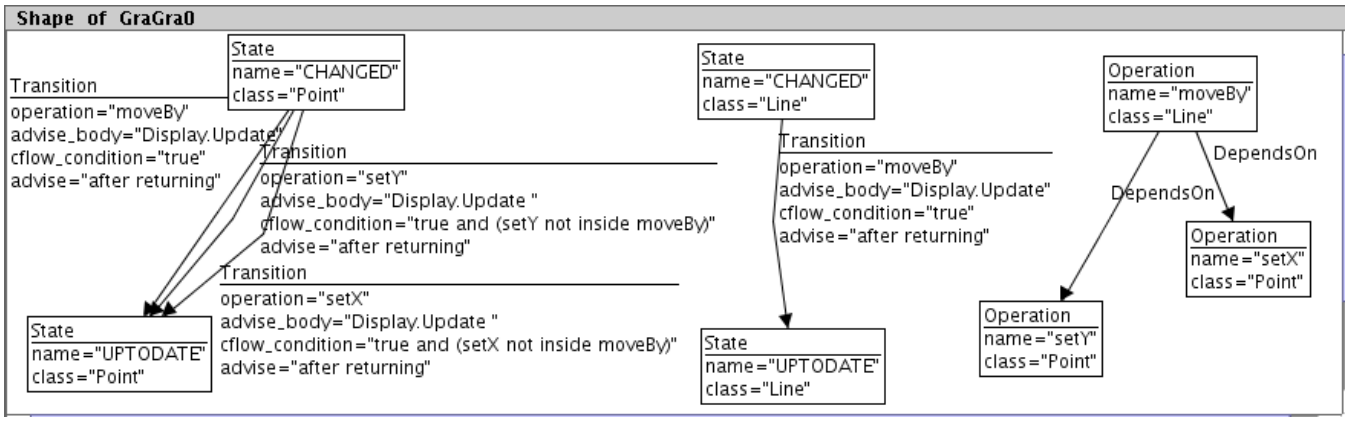


Figure 5: Model revised

systems. In H. E. et al., editor, *Proceedings of ICGT 2004*, volume 3256 of *LNCS*, pages 383–398. Springer-Verlag, 2004.

[8] R. Douence, P. Fradet, and M. Südholt. Composition, reuse, and interaction analisys of stateful aspects. In *Proceedings of the 3rd international conference of aspect-oriented software development*, Lancaster, UK, Mar. 2004. ACM.

[9] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns*, 2000.

[10] S. Katz. Diagnosis of harmful aspects using regression verification. In G. T. Leavens, R. Lämmel, and C. Clifton, editors, *Foundations of Aspect-Oriented Languages*, Mar. 2004.

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

[12] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, 2005. ACM. to appear.

[13] Maya. Boh. In *Proceedings of AOSD 2004*, Lancaster, UK, Mar. 2004.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *3rd International conference on Meta-level architectures and separation of concerns*, number 2192 in Lecture Notes in Computer Science, pages 1–25. Springer-Verlag, 2001.

[15] M. Rudolf and G. Taentzer. *Introduction to the language concepts of Agg*. TU Berlin, Nov. 1999.

[16] P. Tarr and H. Ossher. *Hyper/J^{TM} User and Installation Manual*. IBM Research, 2000.