How to Integrate Graph Transformation with an Object-Oriented Language*

Gabriele Taentzer
TU Berlin
gabi@cs.tu-berlin.de

Abstract

Object-oriented as well as visual programming concepts are very much in vogue for software construction. Two ways how graph transformation as a visual concept and an object-oriented language can be integrated are presented and the advantages of this integration for both paradigms are discussed. Firstly, objects may be used as node and arc attributes, and secondly, graph grammars may be encapsulated in classes. The integration of graph transformation with the object-oriented language Java is sketched to get a concrete impression on how such an integration can look like. The AGG system is planned to be a concrete realization of the integration concepts presented.

1 Introduction

Modern programming languages follow the object-oriented paradigm, since it offers the simple, but powerful structuring principle of object classes. This concept demands the encapsulation of static and dynamic aspects of data structures. It supports the development of components and the reuse of software by building up large class libraries. One of the most prominent object-oriented programming language is Java[1] which comes up with a lot of modern programming concepts, e.g. platform independence, concurrency concepts and powerful class libraries structured in a number of packages to implement graphical user interfaces, distributed applications, to handle input and output, etc.

Until now, all commonly used programming languages are mostly textual. Although there are nowadays tools to develop e.g. a graphical user interface visually, the program logics are still textually coded. It is still a visual object-oriented programming language missing that is widely used, although visual languages aim to be easier to use and to communicate to the users. There are a number of visual object-oriented programming languages. A nice overview on this research area is given in [BGL95]. The visual languages presented there, incorporate the object-oriented concepts into visual language elements. Although most of them allow textual attributes, class libraries written in a common textual object-oriented language are not easily accessable.

Visual programming languages follow very different underlying programming paradigms, like concurrent languages, constraint-based languages, data flow languages and rule-based languages. In the following, we want to take a closer look at visual rule-based languages. This programming paradigm offers rules to describe the systems dynamics and it suits very well to describe any kind of reactive systems. The application of a rule follows the *recognize-select-execute cycle* where first one or more occurrences of a rule's left-hand side to a runtime graph is determined. Then, one rule and one of its occurrences are selected and the rule application is executed resulting in an exchange of the occurrence by a copy of the right-hand side of the rule, correctly embedded into the application context. To have the opportunity of dealing with complex data structures we focus on a special kind of rule-based languages, the graph transformation based languages where the underlying structure is a graph.

Integrating a textual object-oriented language with a visual graph transformation based language leads to a hybrid programming language

*Research partially supported by the German Research Council (DFG), the TMR network GETGRATS, and the ESPRIT Basic Research Working Group APPLIGRAPH

[1] "Java" is a trademark of Sun Microsystems, Inc.

with the advantage that large well-programmed class libraries can be used. Furthermore, application-dependent non-standard data and object structures and their manipulating operations can be visualized by graphs and graph rules. These structures are of main interest. If they are visualized, they might be easier to communicate to other developers and users of the application.

The data structures to be visualized are described by graphs where structure elements are presented by nodes and arcs describe relations between these elements. Structure elements and relations may be further attributed by objects. This attribution is the connection to the textual part of such hybrid program. All kinds of standard objects like data objects, containers, widgets, etc. as well as custom-defined objects may be used for attribution. Examples for hybrid structures may be visual file trees which rely on internal file objects handled by a file manager, virtual road maps for shortest path analysis, etc. The manipulation of such a hybrid structure is performed by attributed graph rules where methods may be invoked on attribute objects.

Pure rule-based languages do not provide any control of rule applications, i.e. rules are randomly selected and applied whenever applicable. This behavior is often not desired, and as a consequence, implicit control structures are often mixed up with data structures. Another possibility is to integrate the imperative programming constructs to control rule application. If, moreover, a visually programmed application part should be embedded into a bigger environment, common object-oriented language concepts can be used again. The visual application part is encapsulated into a new class, a so-called graph transformation class, and this class can be used in or inherited by other classes as usual.

Integrating graph transformation and an object-oriented language in this way, as much of the application as desired may be visually developed, i.e. the amount of visually programmed code is scalable. On the other hand, a textual object-oriented language may be used for all standard problems and application parts with simple structures not needed to be visualized. The overall structuring principle for textual and visual parts is object-oriented, since graphical structures and their dynamics described by a start graph and a set of rules may be summarized in a graph transformation class.

In this paper, the concepts of such a hybrid programming language are illustrated at the concrete integration of the algebraic approach to graph transformation [CMR+97, EHK+96] and the Java programming language which is the idea behind the AGG-system [TER98].

2 Objects As Node and Arc Attributes

Describing the topmost user-defined structure of an application by a graph the structure elements are usually complex items that have to be further attributed by additional information. Consider e.g. shipping software where the topmost structure would represent the ingredients of a shipping company as there are the trucks, employees, cargo, etc. Attributes of these items may be names, weight of a truck, list of cargo in a truck, address of an employee, etc. These attributes may be simple data as well as complex object structures. Considering a file system, the graph may serve as a visualization of the internal tree structure. The actual files on the disk can be accessed by attributes of the tree nodes.

Attributes have three components: the attribute type, its name and its value. Taking objects as attributes, the corresponding object class defines the attribute type. The object itself is the attribute value. Attribute values in a host graph have to be constants.

Graphs are transformed by the application of graph rules. In the attribute part such a rule contains computations on attribute values, i.e. objects, at runtime. Computation on objects means method invocation on objects.

The left-hand side of a graph rule may contain constants or variables as attribute values. Variables are usually implicitly declared by just typing them in as attribute values. In this way they also get a type, i.e. the type of the current attribute instance. The scope of a variable is its rule, i.e. each variable is rule-globally defined. In a rule's right-hand side expressions may contain the variables declared on the left-hand side.

Considering AGG, attribute values may be given by Java expressions, i.e. primitive types such as `byte`, `char`, `int`, `boolean`, arithmetic expressions (e.g. `x + 1`), Boolean expressions (e.g. `width <= 7`), bit manipulations, conditional expressions (e.g. `v>8?true:false`), operations on arrays, creating a new object instance by calling a constructor of a class (e.g. `new Vector()`), access to class and instance variables and invocation of class and instance methods (e.g. `Math.sin(0.5)` or `Vector v.addElement(new Integer(i))`).

In Figure 1 the graphical user interface of AGG with graph grammar `ShowFiles` is presented. This graph grammar contains two rules `init` and `addFile`. Rule `init` is shown in Figure 2. The left and right-hand sides of rule `addFile` are depicted in the upper two graph panels. The node in the left-hand side is preserved and occurs again in the right-hand side. This is indicated by the same numbers at the nodes' upper left corners. The start graph `DirTree` is equal to the left-hand side of rule `init` and not shown. This graph grammar is able to scan a file tree on your operating system and to visualize it as a graph. Besides the usual JDK[2] classes it uses the custom class `Entry` presented below.

The lower graph panel contains a host graph visualizing the contents of some directory `/tmp` after applying the rule as often as possible. Rule `addFile` is not applicable anymore, since attribute `expanded` is set true in all nodes.

```
import java.io.*;
public class Entry extends File {
 public Entry(String p, String n) {
  super(p,n);
 }
 public String ChildName(int x){
  String[] contents = list();
  if ( i < contents.length)
   return contents[i];
  else return "";
 }
 public Entry Child(int i){
  String path = getAbsolutePath();
  String name = ChildName(i);
  if(name != "")
   return new Entry(path,name);
  else return null;
 }
}
```

Three methods of this class are used within `ShowFiles`. These are `ChildName` to retrieve the name of the directory entry i and `Child` to refer to the child entry itself. Attribute `exp`(anded) controls the applicability of the rule. `exp` becomes true if the content list of the current directory is fully shown.

[2] Java Development Kit, available at `http://www.javasoft.com/products/index.html`

Attributes which are not changed by a rule do not have to occur in the rule. If an attribute has to have a certain value, it has to occur in the rule's left-hand side. Its occurrence in the right-hand side is not needed unless the attribute value shall be changed by the rule. (Compare e.g. attribute `entry` in figure 1 which is not changed.) If a graph object is newly created, all declared attributes have to be set in the rule.

Moreover, graph rules can have objects as in and out parameters. Applying a graph rule to a host graph, the parameters have to be set from outside whereas the variables are instantiated by matching the left-hand side to the graph. Although a variable may occur several times in the left-hand side, it has to be instantiated with exactly one value. In this way the variable instantiation function as some kind of matching constraints. When a suitable match has been selected the rule application can take place. All attribute expressions in the right-hand side are evaluated with the variable instantiation selected. A copy of the right-hand side together with the computed attribute values rewrites the occurrence of the left-hand side in the host graph.

The application of a rule may be further restricted by attribute conditions. These are Boolean expressions that may contain all the variables declared for this rule. A rule with attribute conditions can only be applied if the match found satisfies also the attribute conditions. In our `ShowFiles` example, value `false` of attribute `exp` can be replaced by attribute condition `d.Child(x+1).compareTo("")!=0`.

3 Graph Transformation Classes and Rule Application Control

A graph grammar consists of a start graph and a set of graph rules. All graph transformation sequences starting at the start graph and applying rules of the given rule set in the way described above are possible runs of the graph grammar and describe its semantics.

Controlling the rule application means restricting the set of runs to a subset. The imperative control constructs offered by a programming language like Java, C, C++, etc. can serve for this purpose. Concerning imperative con-

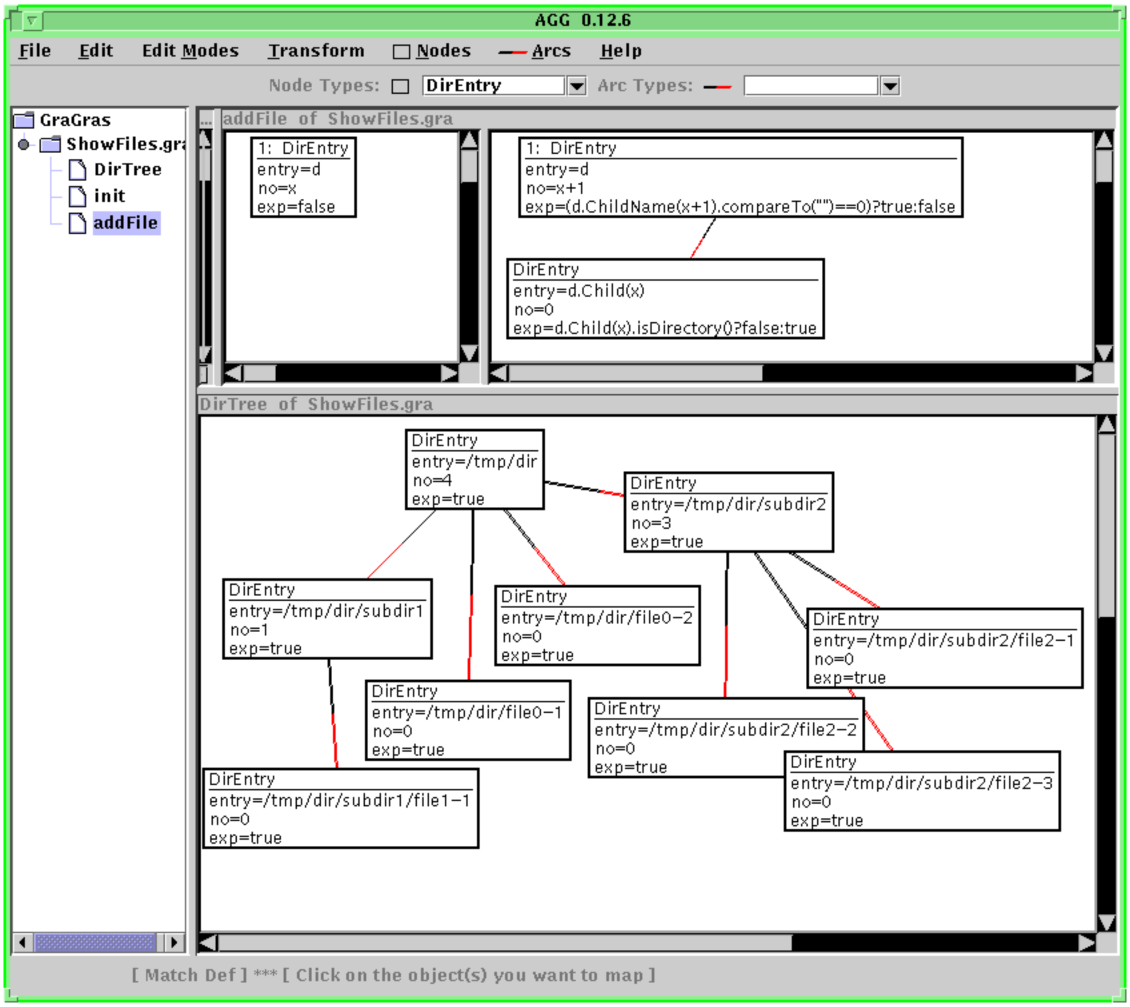

Figure 1: The AGG graphical user interface showing rule `addFile` and a host graph visualizing the contents of some directory `/tmp`

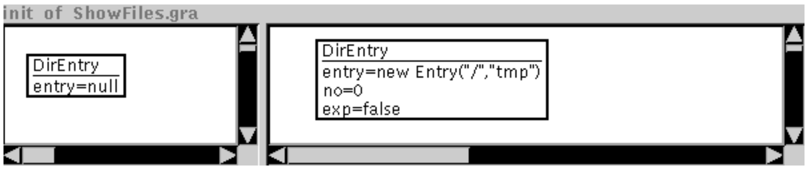

Figure 2: Rule `init` of graph grammar `ShowFiles`

troling it does not make any difference, whether the language chosen is object-oriented or not. The advantage of using an object-oriented language here is the possibility to integrate graph transformation very tightly with the object-oriented language concepts for attribution *and* control.

The idea is to automatically generate a graph transformation class from a graph grammar. This class should be able to perform graph transformation on this graph grammar without any rule application control. The start graph has to be stored as attribute within this graph transformation class. Moreover, this class should offer methods to apply rules. Methods to test the applicability of a rule to the runtime graph are further nice services. (Example class `ShowFiles` is given below.)

After generating such a class from a graph grammar, further classes using this graph transformation class can be established to implement the control flow on rule applications. New methods may be provided to implement complex operations on graphs not describable by one rule. For example, shortest path computation is not realizable by only one rule application if the road map is represented by a graph. Implementing e.g. the well-known branch-and-bound algorithm a number of rule applications have to be performed to compute a (restricted) decision tree step by step. Implementing these new methods the basic methods of the graph transformation class generated may be used. (Compare [TER98] for further details to this sample application.)

Considering AGG, it seems reasonable to implement the control of rule application in Java, i.e. the same language as for attribute computation. Binding graph transformation in this way very tightly to an object-oriented language offers a high flexibility in using both programming paradigms in one application, since new classes inheriting from graph transformation classes can be used again as attribute types in another graph grammar.

Given an attributed graph grammar within AGG it is possible to automatically generate a corresponding graph transformation class in Java. For example `ShowFiles` the graph transformation class should include at least the class skeleton depicted below.

Class `ShowFiles` may contain an attribute where the start graph is stored. Methods `initIsApplicable` and `addFileIsApplicable` formulates tests on applicability of rules `addFile` and `init` whereas methods `addFile` and `init` apply the corresponding rule once. The return value of these methods is `true` if one step could be performed. Otherwise `false` is returned. The class skeleton presented has to be extended if we want to deal with rule parameters. These may be simply added as method parammeters.

Alternatively, the generated class may be more generic, if we do not provide a method for each rule, but a method `executeAny` with a rule set as parameter instead to realize the non-deterministc choice of rule applications which is typical for graph transformation. But in this case, we cannot ensure type reliability for further integrated rule parameters.

```
public class ShowFiles {
   private Graph DirTree;
   public ShowFiles(){ }
   public boolean init(){ }
   public boolean initIsApplicable(){ }
   public boolean addFileIsApplicable(){ }
   public boolean addFile(){ }
}
```

To visualize the whole subtree of a directory given we construct another class which uses class `ShowFiles`. This new class, called `ShowAllFiles`, which is depicted below defines the control flow stating that first rule `init` is applied once and afterwards rule `addFile` is applied as often as possible.

```
public class ShowAllFiles {
  private boolean applicable = true;
  public void visualize(){
    ShowFiles sf = new ShowFiles();
    applicable = sf.init();
    while (applicable)
      applicable=sf.addFile();
  }
}
```

Dealing with more than two rules, rule parameters and partial matches, more sophisticated control structures can be defined. For example, a rule parameter stating the directory's name to be visualize would be more flexible than the solution shown above. The next version of AGG will capture this feature.

4 The Object-Oriented View on the Integration

Throughout this contribution the integration has been considered from the graph transformation point of view. With graph transformation in the center we discussed its integration with an object-oriented language to capture additional features, i.e. attributes and control flow. The aim is to evolve graph transformation based concepts to a (hybrid) programming language.

From the object-oriented point of view the overall programming principle are object classes. As shown in the previous section, graph transformation can be encapsulated into classes. This fact coordinates the integration of graph transformation into an object-oriented language.

The question is, what does the object-oriented world gain from this integration? Integrating graph transformation, two new programming concepts become available within the object-oriented language: visual and rule-based programming becomes possible without destroying the overall object-oriented structure. On one hand the manipulation of complex data structures can be easier realized by drawing the operations, on the other hand reactive systems operating highly independently and concurrently are better described in a rule-based manner to avoid global control flow as far as possible. Several examples for the first kind of application systems have been given throughout this contribution. Into the second category of systems fall e.g. games like PacMan [EHK+96], distributed algorithms [Hec98], etc.

5 Related Work

Concerning attribution and rule application control, the way how graph transformation is embedded into other programming paradigms is similar to that in PROGRES [Sch91]. The main differences are: PROGRES has some basic built-in data types like `integer`, `string` and `boolean`. All other types like `file` have to be defined in a so-called host language which may be Modula-2 or C and to be imported to PROGRES. In AGG no built-in types exist and Java is used for this purpose. Custom-built imperative control constructs are used to formulate programmed graph rewriting, i.e. the control constructs are not taken from an established programming language.

Programmed graph rewriting in PROGRES is based on transactions which are designed to have the same characteristics as rules. Using Java to control rule application as it is planned for AGG such a transaction concept is not automatically provided, but this integration is as flexible as transactions. This or other structuring concepts may be realized. Nowadays PROGRES is used at various different sites for specification and rapid prototyping. This number of developed PROGRES applications demonstrates that there is a potential to attract software developers to use graph transformation based languages within their projects.

For the algebraic approaches, there are several approaches to attributed graph transformation [Sch92, LKW93, CL95]. All these approaches combine graph transformation with algebraic specifications. Whereas [Sch92] presents an attribution concept for the double-pushout approach, [LKW93] and [CL95] formulate attributed graph transformation in the single pushout approach where the data algebras in [LKW93] have to be total and those in [CL95] are allowed to be partial. The integration of objects as node and arc attributes follows these ideas to a large extent. The main difference is that here, Java classes and expressions are used instead of specifications and terms.

It is nearby to discuss the formulation of object-oriented concepts for graph transformation. Main contributions in this line of research are node classes in PROGRES [Sch91, SWZ95] and refinement concepts for graph grammars [CH95].

PROGRES contains the concept of node classes as types of node types. Node classes may be multiply inherited. Similar to node types, they inherit attribute definitions which may be redefined. This can compactify the graph schemes used to define the node and arc types. Node classes can also be used to define generic graph rules. When applying a rule its node classes are instantiated by concrete node types. Here, node classes help to avoid sets of similar rules which differ only in the node types. For AGG we have not considered node classes. As for graph grammars also node and arc classes may be generated, containing attributes, but no methods. Node and arc classes may be further used in graph transformation classes. It is up to future work to examine in detail how these different kinds of classes can be combined.

In [CH95] a refinement concept for graph grammars is presented. For this purpose, refinement morphisms between graph grammars are invented which map the ingredients of a refining graph grammar to those of the refined one. The mappings have to be surjective to guarantee that the refining grammar has all the functionalities the refined grammar has.

Another research work which is closely related started recently at the university of Paderborn. The idea is to generate Java code from graph transformation based on so-called story driven modeling [JZ98]. This kind of modeling is intended to be an extension of the current object-oriented modeling techniques. Class diagrams describe the static aspects of a system as usual. Story flow diagrams contain rules on object structures and application control to describe the system's dynamics. For each class certain story flow diagrams are translated into Java code. The main difference to our approach is that a Java class is generated from a certain node and some set of graph grammar rules being behind corresponding story flow diagrams. In our approach, Java classes are generated from graph grammars, i.e. a start graph and a set of rules.

6 Conclusions

The integration of graph transformation with an object-oriented language can be performed in two ways: firstly, using objects as attributes of graph nodes and arcs, and secondly, encapsulating graph grammars in classes. The languages used for attribution and for application control are the same and both are object-oriented. Integrating both paradigms such tightly the integration can be iterated in a way that a class using or inheriting from a graph transformation class can be used again as attribute type in another graph grammar. This offers a high flexibility in using both paradigms in one application and surely increases the acceptance of graph transformation as programming paradigm.

AGG is a concrete implementation of the concepts presented. It already supports the integration of Java objects as attributes of graph nodes and arcs. The automatic generation of a graph transformation class from a graph grammar is one of the extensions planned. AGG itself is written in Java, and it is available at `http://tfs.cs.tu-berlin.de/agg`.

Several ideas remain open for future work: First of all, the type concept in AGG is still implicitly defined. A type graph containing node, edge and attribute types describes the static system aspects. It is very useful to state integrity conditions on the graphs to be considered. Further discussions have to clarify how far the following object-oriented concepts are to be integrated into AGG. These are node classes and their inheritance where edges define associations as well as integration with object-oriented modeling languages such as UML [Rat97].

The proper aim of AGG is to make graph transformation fit for distributed applications. By the way, that is the most important reason why AGG integrates graph transformation with Java and not any other object-oriented language. Especially distributed algorithms are well described by graph transformation since the highly independent and concurrent actions within such systems are better formulated in a rule-based manner to avoid global control flow as far as possible. The whole development of AGG is headed towards a visual design tool for distributed systems which is able to at least prototypical realize really distributed systems with graph transformation.

Acknowledgments

Most of the integration ideas presented have been discussed within the AGG development group at TU Berlin, consisting mainly of Claudia Ermel, Boris Melamed, Michael Rudolf and Olga Runge. Hartmut Ehrig and Andy Schürr gave valuabel hints to round off this contribution.

References

[BGL95] M. Burnett, A. Goldberg, and T. Lewis (eds.). *Visual Object-Oriented Programming: Concepts and Environments.* Manning, Greenwich, 1995.

[CH95] A. Corradini and R. Heckel. A Compositional Approach to Structuring and Refinement of Typed Graph Grammars. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995.

http://www.elsevier.nl/locate/entcs/volume2.html.

[CL95] I. Claßen and M. Löwe. Scheme evolution in object oriented models: A graph transformation approach. In *Proc. Workshop on Formal Methods at the ISCE'95, Seattle (U.S.A.)*, 1995.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations.* World Scientific, 1997.

[EHK+96] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg (ed.), *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1996.

[Hec98] R. Heckel. Compositional Verification of distributed Systems Specified by Graph Transformation. Submitted, 1998.

[JZ98] J. Jahnke and A. Zündorf. Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modelling. Accepted for publication in IWSSD-9,Kyoto, IEEE to appear, 1998.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M. Sleep, M. Plasmeijer, and M. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[Rat97] Rational software cooperation. Unified Modeling Language. available via *http://www.rational.com*, 1997.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen.* Deutscher Universitätsverlag GmbH, Wiesbaden, 1991.

[Sch92] G. Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme.* Arbeitsberichte des Institus für mathematische Maschinen und Datenverarbeitung (Informatik), University of Erlangen, 1992.

[SWZ95] A. Schürr, A. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer and P. Botella (eds.), *5th European Software Engineering Conference (ESEC'95), Sitges*, pages 219–234. Springer LNCS 989, September 1995.

[TER98] G. Taentzer, C. Ermel, and M. Rudolf. AGG-Approach: Language and Tool Environment. to appear as article in graph grammar handbook 2 "Specification and Programming", 1998.