AGG: A Graph Transformation Environment for Modeling and Validation of Software*

Gabriele Taentzer

Technische Universität Berlin, Germany
gabi@cs.tu-berlin.de

Abstract. AGG is a general development environment for algebraic graph transformation systems which follows the interpretative approach. Its special power comes from a very flexible attribution concept. AGG graphs are allowed to be attributed by any kind of Java objects. Graph transformations can be equipped with arbitrary computations on these Java objects described by a Java expression. The AGG environment consists of a graphical user interface comprising several visual editors, an interpreter, and a set of validation tools. The interpreter allows the stepwise transformation of graphs as well as rule applications as long as possible. AGG supports several kinds of validations which comprise graph parsing, consistency checking of graphs and conflict detection in concurrent transformations by critical pair analysis of graph rules. Applications of AGG include graph and rule-based modeling of software, validation of system properties by assigning a graph transformation based semantics to some system model, graph transformation based evolution of software, and the definition of visual languages based on graph grammars.

1 Introduction

Graphs play an important role in many areas of computer science and they are especially helpful in analysis and design of software applications. Prominent representatives for graphical notations are entity relationship diagrams, control flows, message sequence charts, Petri nets, automata, state charts and any kind of diagram used in object oriented modeling languages as UML. Graphs are also used for software visualizations, to represent the abstract syntax of visual notations, to reason about routing in computer networks, etc. Altogether graphs represent such a general structure that they occur nearly anywhere in computer science.

Graph transformation defines the rule-based manipulation of graphs. Since graphs can be used for the description of very different aspects of software, also graph transformation can fulfill very different tasks. E.g. graphs can conveniently be used to describe complex data and object structures. In this case, graph transformation defines the dynamic evolution of these structures.

* Research partly supported by the German Research Council (DFG), and the EU Research Training Network SegraVis.

J.L. Pfaltz, M. Nagl, and B. Böhlen (Eds.): AGTIVE 2003, LNCS 3062, pp. 446–453, 2004.
© Springer-Verlag Berlin Heidelberg 2004

Graphs have also the possibility to carry attributes. Graph transformation is then equipped with further computations on attributes. Since graph transformation can be applied on very different levels of abstraction, it can be non-attributed, attributed by simple computations or by complex processes, depending on the abstraction level. AGG graphs may be attributed by Java objects which can be instances of Java classes from libraries like JDK as well as user-defined classes.

The graphical user interface provides a visual layout of AGG graphs similar to UML object diagrams. Several editors are provided to support the visual editing of graphs, rules and graph grammars. Additionally, there is a visual interpreter, the graph transformation machine, running in several modes. If another than the standard layout is preferred, it is possible to just use the underlying graph transformation machine and to implement a new layout component or, moreover, a new graphical interface for the intended application.

AGG has a formal foundation based on the algebraic approach to graph transformation [1, 2]. Since the theoretical concepts are implemented as directly as possible – not leaving out necessary efficiency considerations – AGG offers clear concepts and a sound behavior concerning the graph transformation part. Clearly, Java semantics is not covered by this formal foundation.

Due to its formal foundation, AGG offers validation support like graph parsing, consistency checking of graphs and graph transformation systems as well as conflict detection of graph transformation rules. Graph parsing can be advantageously used to e.g. analyze the syntax of visual notations. Modeling a dynamic system structure as graph there is generally the desire to formulate invariants on the class of evolving graphs. AGG offers the possibility to formulate consistency conditions which can be tested on single graphs, but which can also be shown for a whole graph transformation system. If a consistency condition holds for a graph transformation system, all derived graphs satisfy this condition. The conflict detection of graph rules is useful to check dependencies between different actions. It is based on a critical pair analysis for graph rules, a technique which has been developed for term rewriting used in functional programming. Since graph rules can be applied in any order, conflict detection helps to understand the possible interactions of different rule applications.

2 Graph Transformation Concepts

Graph transformation based applications are described by AGG graph grammars. They consist of a type graph defining the class of graphs used in the following, a start graph initializing the system, and a set of rules describing the actions which can be performed. The start graph as well as the rule graphs may be attributed by Java objects and expressions. The objects can be instances of Java classes from libraries like JDK as well as user-defined classes. These classes belong to the application as well. Moreover, rules may be equipped by negative application conditions and attribute conditions. The type graph defines all possible node and edge types, their possible interconnections and all attribute types.

Please note that the type graph can contain additional application conditions for rules, since it offers the possibility to set multiplicity constraints for arc types. Type graphs have first be introduced in [6].

The way how graph rules are applied realizes directly the algebraic approach to graph transformation as presented in [1, 2]. The formal basis for graph grammars with negative application conditions (NACs) was introduced in [8].

Besides manipulating the nodes and arcs of a graph, a graph rule may also perform computations on the objects' attributes. During rule application, expressions are evaluated with respect to the variable instantiation induced by the actual match. The attribution of nodes and arcs by Java objects and expressions follows the ideas of attributed graph grammars as stated in [11] and further in [14] to a large extent. The main difference here is the usage of Java classes and expressions instead of algebraic specifications and terms. The combination of attributed graph transformation with negative application conditions has been worked out comprehensibly in [14]. The AGG features follow these concepts very closely.

Graph transformation can be performed in two different modes using AGG. The first mode to apply a rule is called *Debug* mode. Here, one selected rule will be applied exactly once to the current host graph. The matching morphism may be (partially) defined by the user. Defining the match completely "by hand" may be tedious work. Therefore, AGG supports the automatic completion of partial matches. If there are several choices for completion, one of them is chosen arbitrarily. All possible completions can be computed and shown one after the other in the graph editor. After having defined the match, the rule will be applied to the host graph once. The result is shown in the graph editor that is, the host graph is now transformed according to the rule and the match. Thereafter, the host graph can immediately be edited.

The second mode to realize graph transformation is called *Interpretation* mode. This is a more sophisticated mode, applying not only one rule at a time but a whole sequence of rules. The rule to be applied and its match are nondeterministically chosen. Starting the interpretation, all rules are applied as often as possible, until no more match for any rule can be found. Please note that in general the result graph is not unique, since the application of one rule may avoid the application of another rule.

The basic concepts of AGG are presented comprehensively in [7]. In the following, the presentation of AGG concentrates on its new features which cover mainly validation possibilities.

3 Validation Support

Besides editing and interpretation facilities, AGG also offers support for model validation. Since the main AGG concepts rely on a formal approach to graph transformation, i.e. the algebraic approach, validation techniques developed formally for this approach, can be directly implemented in AGG. In the following, three main techniques are presented.

3.1 Graph Parsing

The AGG graph parser is able to check if a given graph belongs to a certain graph language determined by a graph grammar. In formal language theory, this problem is known as the membership problem. Here, the membership problem is lifted to graphs. Three different parsing algorithms are offered by AGG, all based on back tracking, i.e. the parser is building up a derivation tree of possible reductions of the host graph. Leaf graphs are graphs where no rule can be applied anymore. If a leaf graph is isomorphic to the stop graph, the parsing process finishes successfully. Since simple back tracking has exponential time complexity, the simple back tracking parser is accompanied by two further parser variants exploiting critical pair analysis for rules.

Critical pair analysis can be used to make parsing of graphs more efficient: decisions between conflicting rule applications are delayed as far as possible. This means to apply non-conflicting rules first and to reduce the graph as much as possible. Afterwards, the conflicting rules are applied, first in uncritical situations and when this is not possible anymore, in critical ones. In general, this optimization reduces the derivation tree constructed, but does not change the worst case complexity.

A parsing process might not terminate, therefore so-called layering conditions are introduced. Using layers for rules and graph types such that each rule deletes at least one graph object, which is of the same or a lower layer, creates graph objects of a greater layer only, and has negative application conditions with graph objects of the current or lower layers only, it has been shown in [5] that a parsing process based on such a layered graph transformation always terminates.

3.2 Critical Pair Analysis

Critical pair analysis is known from term rewriting and usually used to check if a rewriting system is confluent. Critical pair analysis has been generalized to graph rewriting. Critical pairs formalize the idea of a minimal example of a conflicting situation. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies.

A critical pair is a pair of transformations both starting at a common graph G such that both transformations are in conflict, and graph G is minimal according to the rules applied. The set of critical pairs represents precisely all potential conflicts, i.e. there exists a critical pair like above if, and only if, one rule may disable the other one. There are three reasons why rule applications can be conflicting: The first two are related to the graph structure while the last one concerns the graph attributes.

1. One rule application deletes a graph object which is in the match of another rule application.
2. One rule application generates graph objects in a way that a graph structure would occur which is prohibited by a negative application condition of another rule application.

3. One rule application changes attributes being in the match of another rule application.

Please note that using a type graph with multiplicity constraints usually leads to considerably less critical pairs.

3.3 Consistency Checking

Consistency conditions describe basic properties of graphs as e.g. the existence of certain elements, independent of a particular rule. To prove that a consistency condition is satisfied by a certain graph grammar, i.e. is an invariant condition, a transformation of consistency conditions into post application conditions can be performed for each rule [10]. A so-constructed rule is applicable to a consistent graph if and only if the derived graph is consistent, too. A graph grammar is consistent if the start graph satisfies the consistency conditions and the rules preserve this property.

In AGG, consistency conditions are defined on the basis of graphical consistency constraints which consist each of a premise graph and a conclusion graph such that the premise graph can be embedded into the conclusion graph. A graphical consistency constraint is satisfied by a graph G if for each graph pattern equivalent to the premise also an extension equivalent to the conclusion can be found. Based on graphical consistency conditions a propositional logics of formulae has been defined, to be used to formulate more complex consistency conditions.

4 The Tool Environment

Figure 1 shows the main graphical user interface of the AGG system. To the left, a tree view with all graph grammars loaded is shown. The current graph grammar is highlighted. A selected graph, rule or condition is shown in its corresponding graphical editor on the right. The upper editor is for rules and conditions showing the left and the right-hand sides or the premise and conclusion, respectively. The lower editor is for graphs. The attribution of graph objects is done in a special attribute editor that pops up when a graph object is selected for attribution. In Fig. 1 a grammar for a sample requirement specification on shopping is shown. This grammar is presented in detail in [9] where conflict detection of such a requirement specification is analyzed by critical pair analysis. Here, we extend the specification by two consistency conditions to show how they can look like. The graph editor on the lower right shows the type graph of this sample application. It contains typical object types around shopping. On top of the type graph, an atomic consistency condition "UniqueGoodPropriety" is depicted which is part of consistency condition "Proprieties" being a logical formula shown in a separate formula editor on the lower left. Condition "Proprieties" is defined as conjunction of atomic conditions "UniqueGoodPropriety" and "RackPropriety". Condition "UniqueGoodPropriety" has two conclusions,

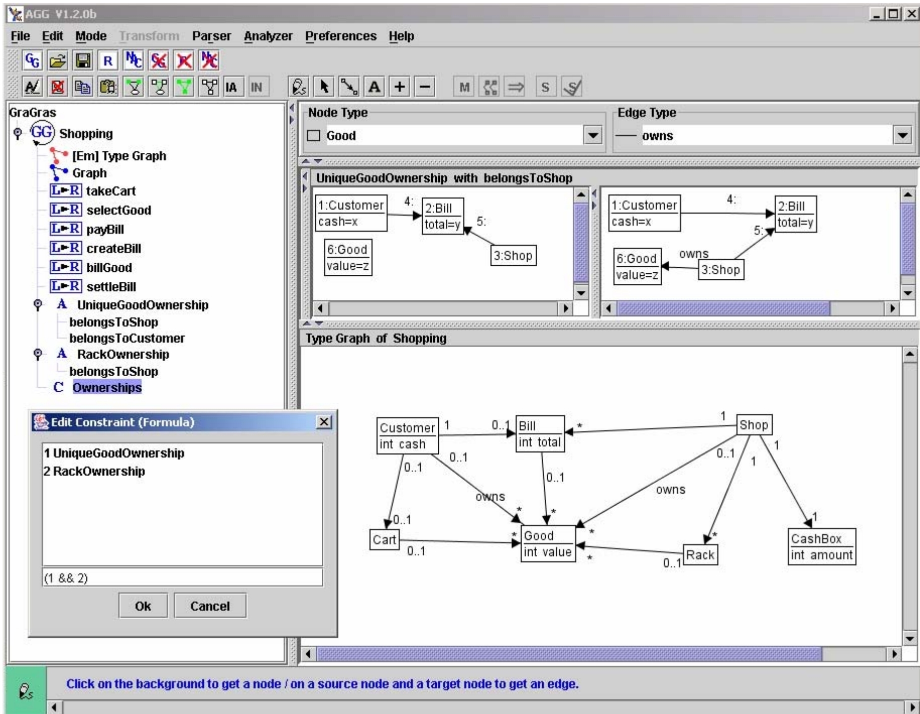

Fig. 1. Screen dump of AGG

i.e. this condition contains two sub-conditions (with the same premise) disjunctively connected. The one presented expresses that each good belongs to a shop. Otherwise, it has to belong to a customer.

Graphs and graph grammars can be stored as XML documents [15], especially AGG supports exchange in GXL [16], the quasi standard format for graphs.

5 Conclusion

This paper gives a rough overview on the graph transformation environment AGG. It consists of visual editors for graphs, rules and graph grammars as well as a visual interpreter for algebraic graph transformation. Moreover, standard validation techniques for graph grammars are supported. Applications of AGG may be of a large variety because of its very flexible attribution concept relying on Java objects and expressions. E.g. the application of AGG for visual language parsing [5] implemented in GenGEd [3], conflict detection in functional requirement specifications [9], consistency checking of OCL constraints in UML models [4] have been considered. AGG is not the only tool environment which is based on graph transformation. In this context we also have to mention PROGRES [13] and a variety of tools which apply graph transforma-

tion in a certain context, e.g. Fujaba [17], DiaGen [18], GenGED [3] and many more. But AGG is the only one which consequently implements the theoretical results available for algebraic graph transformation to support their validation. The development group of AGG at the Technical University of Berlin will continue implementing concepts and results concerning validation and structuring of graph transformation systems, already worked out formally. AGG is available at: `http://tfs.cs.tu-berlin.de/agg`.

Acknowledgement

Large parts of AGG have been developed by Olga Runge. She is carefully maintaining AGG, caring about all technical and documentation issues and puts a lot of efforts on integrating students' work smoothly into the whole project.

References

[1] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*, pages 163–246. World Scientific, 1997. 447, 448

[2] 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, editor, *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1996. 447, 448

[3] R. Bardohl. A Visual Environment for Visual Languages. *Science of Computer Programming (SCP)*, 44(2):181–203, 2002. 451, 452

[4] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In A. Evans and S. Kent, editors, *UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*. Springer, 2000. 451

[5] P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as technical report SI-2000-06, University of Rome. 449, 451

[6] A. Corradini, U. Montanari, and F. Rossi. Graph Processes. *Special Issue of Fundamenta Informaticae*, 26(3,4):241–266, 1996. 448

[7] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999. 448

[8] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996. 448

[9] J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In *Proc. of Int. Conference on Software Engineering 2002*, Orlando, USA, 2002. To appear. 450, 451

[10] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html. 450
[11] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993. 448
[12] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
[13] A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999. 451
[14] G. Taentzer, I. Fischer, M Koch, and V. Volle. Visual Design of Distributed Systems by Graph Transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*, pages 269–340. World Scientific, 1999. 448
[15] *The Extensible Markup Language (XML)* *http://www.w3.org/XML/*, 2003. 451
[16] *GXL* *http://www.gupro.de/GXL*, 2003. 451
[17] *Fujaba Project Group*, 2003. Available at http://www.fujaba.de. 452
[18] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(3):157 – 180, 2002. 452