AGG: A Tool Environment for Algebraic Graph Transformation*

Gabriele Taentzer**

Università di Roma, Italy, gabi@dsi.uniroma1.it

Abstract. AGG is a general tool environment for algebraic graph transformation 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, and an interpreter which is also usable without the graphical interface.

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 [UML99].

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. Graphs can conveniently be used to describe complex data and object structures. In this case, graph transformation defines the dynamic evolution of these structures.

Graphs have 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 unattributed, attributed by simple computations or by complex processes, depending on the abstraction level. AGG is not specialized to a certain kind of graph transformation application.

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

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

** On the leave of TU Berlin, Germany, gabi@cs.tu-berlin.de

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 single-pushout approach to graph transformation [EHK+97]. 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 cannot be covered by this formal foundation. Furthermore, the formal foundation of AGG offers verification possibilities which can be implemented on top of AGG directly.

2 AGG Concepts

Graph transformation based applications are described by AGG graph grammars. They consist of one graph, the start graph initializing the system, and a set of rules describing the actions which can be performed. The start graph as well as the graphs of the rules 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.

The way how graph rules are applied realizes directly the single-pushout approach to graph transformation as presented in [Löw93,EHK+97]. The formal basis for graph grammars with negative application conditions was introduced in [HHT96]. If the applicability of rules is restricted to the well-known *gluing condition*, the single and the double pushout approaches [CMR+97] yield the same results. (See also the comparison of both approaches in [EHK+97]).

The attribution of nodes and arcs by Java objects and expressions follows the ideas of attributed graph grammars as stated in [LKW93] and further in [TFKV99] to a large extent. The main difference is that here, Java classes and expressions are used instead of algebraic specifications and terms. The combination of attributed graph transformation with negative application conditions has been worked out comprehensibly in [TFKV99]. The AGG features follow these concepts very closely.

In the following, the main AGG concepts are presented in more detail. For the illustration we use a sample implementation of a graph-based HTML browser. This application first scans a given directory for HTML files and presents them as nodes in a file tree. Furthermore, it starts an HTML browser used to inspect the files. This example shows how AGG graph transformation can be used to access system resources like the file system and how to control computations on complex Java objects by graph transformation.

3 The Tool Environment

Figure 1 shows the graphical user interface of the AGG system. To the left, the window with the current graph grammars is shown. It is possible to have more than one graph grammar loaded. The grammars and their contents are visualized by a tree where the current graph grammar, start graph or rule is highlighted. The selected graph or rule is shown in the corresponding graphical editor on the right. The upper editor is for rules showing the left and the right hand sides, the lower for graphs. The attribution of objects is done in a special attribute editor that pops up when a graph object is selected for attribution.

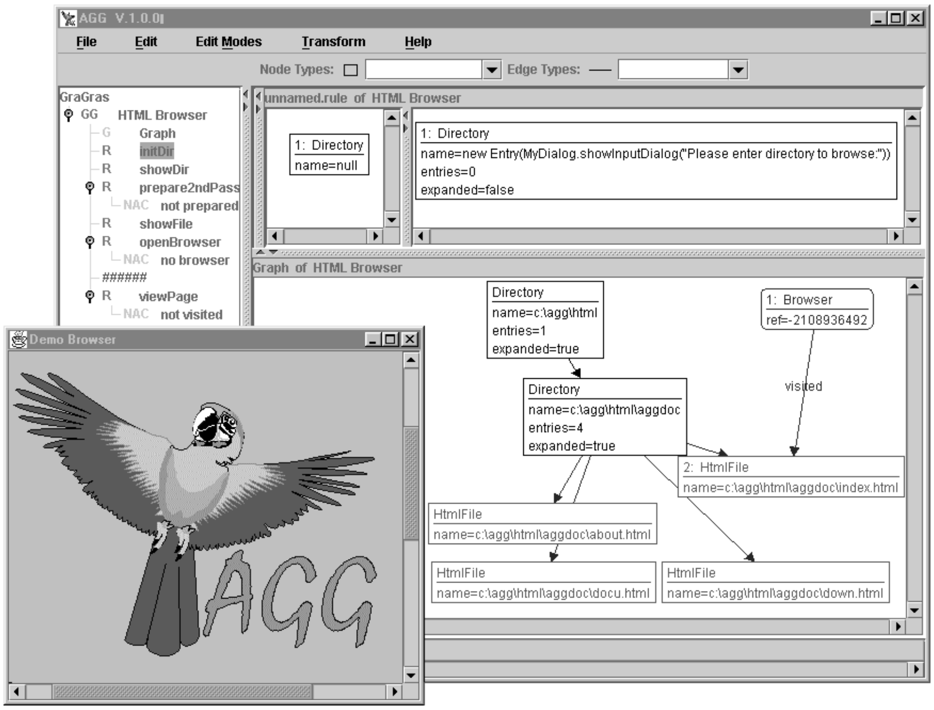

Fig. 1. Screen dump of AGG

In Figure 1 the graph grammar for an HTML browser is shown which contains six rules. The first rule is depicted in the rule editor. The graph editor does not show the start graph but the host graph after running the graph grammar. The start graph looks exactly like the left-hand side of the depicted rule. The graphs and rules are explained in the following sections.

4 Attributed Graphs

AGG graphs are directed and their nodes and arcs may be typed and attributed. A type can be composed from a string and the visual layout. It is possible that the string is empty and the type is determined only by the layout, i.e. different layouts mean different types. For nodes several shapes and colours are supported, arcs may also be differently coloured. Moreover, different arc styles are supported. There may be arbitrary many arcs between two nodes. In the graphical editors, the type name (if any) is shown inside the node shape and next to an arc.

The attributes are specified by a type, a name and a value. Each graph node and arc may have several attributes. All graph objects (nodes and arcs) of one type also share their attribute declaration, i.e. the list of attribute types and names. Only the attribute values may be chosen individually. From a conceptual point of view, attribute declarations have to be considered as an integral part of the definition of a node or arc type.

The attributes may be typed by any valid Java type. This means that it is not only possible to annotate graph objects by simple types like strings or numbers, but that we can also utilize arbitrary custom classes to gain maximal flexibility in attribution. However, the actual power of this concept will be shown in the following section, when we move ahead from the sole graphical description of states to the dynamic aspects of modeling state transitions by graph rules. We will then be able to use arbitrary Java methods to manipulate object attributes as well as to interact with the user or with the underlying system environment, while still specifying the structural aspects of a transition in a graphical way.

To visualize the directory structure to browse HTML files we use three different node types (cf. Figure 1), "Directory", "HtmlFile", and "Browser". A "Directory" node has three attributes: its name as absolute path, the number of entries which already have been recognized, and a boolean flag indicating whether the directory has already been fully expanded or not. "HtmlFile" nodes only have one attribute, the name of the file. A "Browser" node is attributed by the reference to the actual browser object. This is a complex object of a user-defined Java class. In the example, two arc types are distinguished, one is called "visited" and is only used between the browser and an HTML file. The other one is used to show the file structure and is not named. In this example there is no need to attribute the arcs additionally.

5 Graph Rules

Graph rules are used to describe graph transformation. They consist of a left and a right-hand side L and R and, moreover, a set of negative application conditions. The graphs occuring in a rule are typed and attributed, as mentioned above. But it depends on the rule side which kinds of attributes are allowed. Left-hand sides do not only contain concrete Java objects, but are allowed to have also variables. They are used to abstract the operation from concrete attribute values. Moreover, the right-hand sides may contain more complex Java expressions to express computations on the attributes. If an attribute is used only to check some value without changing it, it only has to occur on the left-hand side of a rule. If an attribute value is changed independent of the value it had before, it only has to occur on the right-hand side.

The left and the right-hand side of a rule are related by a partial graph morphism $L \to R$. Those graph parts related by this morphism are preserved by the rule, all the other graph objects in the left-hand side are deleted, all others in the right-hand side are newly created. To indicate which objects are mapped to one another in the graphs, we use numerical tags preceding an object's type name, separated by a colon.

The first rule of our example, called "initDir", is depicted in the rule editor in Figure 1. It does not change the graph structure but only some attributes. If the name of the directory is unset it can be set by the user creating a new dialog for this input. The name is then passed to a new "Entry" object. Since this directory has not been expanded yet, its number of entries is 0 and its flag is set to false.

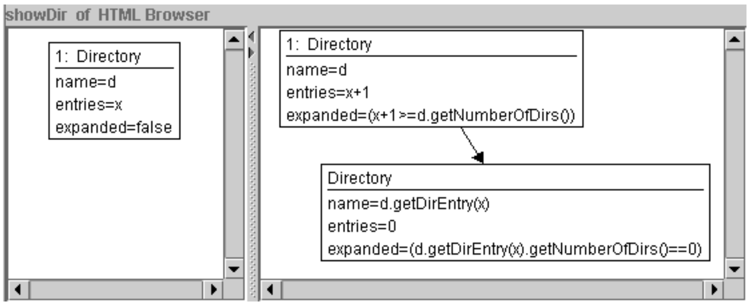

Fig. 2. Rule "showDir"

Rule "showDir" makes a further subdirectory visible. If a directory still is not fully expanded, this rule is applicable. It creates a new "Directory" node in the file tree, increases the number of entries in the parent directory, checks if this is now fully expanded, and sets the attributes of the newly shown directory. Consider especially the complex boolean Java expressions used to compute the values of the expanded attributes. After having shown the directories, the files have to be made visible in the file tree. This is done by rules "prepare2ndPass" and "showFile" in Fig. 1, not shown in detail. Thereafter, the application of rule "openBrowser" (also not shown in detail) starts an HTML browser.

Moreover, a rule may contain a set of negative application conditions (NAC) which are able to express that something *must not* exist for a rule to be applicable. Basically, this is realized by introducing another graph N to the rule which is to hold the negative conditions just like the left-hand side graph L contains the positive ones. Within an NAC, you specify exactly that fraction of a matching situation that you *don't* want to happen. Gathering all the negative application conditions in one graph means to formulate a disjunction of conditions. If only a small part of $N - l(L)$ cannot be mapped the whole condition is satisfied. To express that several graph parts must not exist independently of each other the need for several conditions occur. A rule is applicable if all these conditions are satisfied.

For an example of an NAC consider rule "viewPage" in Figure 3. The NAC is shown on the left, the left-hand rule side in the middle, and the right-hand side on the right. If a certain HTML file has not been shown by the browser,

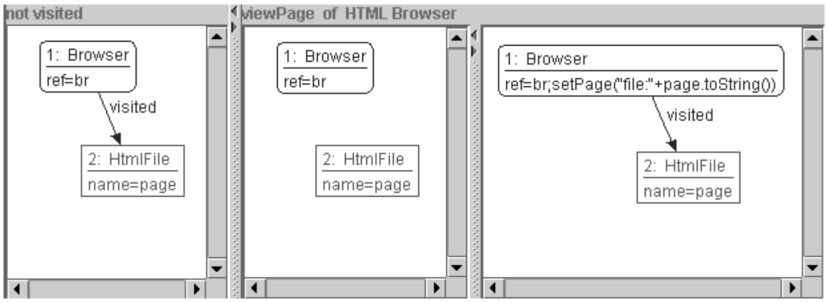

Fig. 3. Rule "viewPage"

this rule shows the file. In Fig. 1 the browser with a section of the AGG home page is visible in the lower left corner. We keep track of the browsing by inserting "visited" arcs in the graph. So if there is not a "visited" arc from the browser to a certain HTML file, it may be shown by the browser and such an arc is inserted. Consider the newly computed attribute of the browser. The reference of the browser should be the same afterwards, but a certain action, called "setPage", should be performed. With the address of the HTML file as input this method induces the browsing of this file. Since the reference should not be set to the return value of the method, we extended the Java syntax slightly by a new operator: the semicolon ";" between an object and its method call. If it is used, the return value is ignored and the attribute value is again the object reference.

Furthermore, rules may contain attribute conditions. They may be any boolean Java expressions and are checked before a rule is applied, i.e. during matching. Attribute conditions are allowed to include any variable declared for the corresponding rule. Clearly, the evaluation of an attribute condition is dependent of variable instantiations. Furthermore, rules may have parameters which are useful to determine e.g. matches and attributes of new graph objects by the user.

6 Rule Application

Rule application is performed in two steps: First we have to check if a rule is applicable to a certain host graph, i.e. we have to find a match $m : L \to G$. Afterwards the rule is applied at one of its matches.

A match is a total graph morphism, i.e. each graph object of L is embedded into Graph G. If a variable occurs several times in a rule's left-hand side, it has always to be matched with the same value. Note that in general, we may find multiple matches of the rule's left-hand side into the host graph, while on the other hand, there may be no matches at all. In the latter case, the rule is *not applicable* to the given host graph. A rule is applicable at a certain match, if all its NAC's and further attribute conditions are satisfied. An NAC is satisfied with respect to a given match $m : L \to G$ if we cannot find a total morphism $n : N \to G$ such that any object of L being mapped by l and n is mapped to the same object by m. We suppose to map the graph part $N - l(L)$ injectively by n, because this seems to be more intuitive.

The basic idea of what happens during a graph transformation is simple and intuitive: the matching pattern found for the rule's left-hand side is taken out of the state graph and replaced by the rule's right-hand side. But let us examine the effect more closely on a per object basis. Since a match is a total morphism, any object o of the rule's left-hand side L has a proper image object $m(o)$ in the host graph G. Now if o also has an image $r(o)$ in the rule's right-hand side R, its corresponding object $m(o)$ in the state graph is *preserved* during the transformation, otherwise it is *removed*. Objects exclusively appearing in R without an original object in L are *newly created* during the transformation. Finally, the objects of the host graph which are not covered by the match are not affected by the rule application at all; they form the so-called *context* which is always preserved during transformations. There is one thing to watch out for when a rule is deleting nodes. To get a graph again, dangling arcs are implicitly removed by a transformation as well, even though they belong to the context which is normally to be preserved.

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. But in AGG, we are not limited to applying simple arithmetic operations on attributes. In fact, we may call arbitrary Java methods in an attribute expression, as long as the overall type of the expression matches the type of the attribute whose value it represents.

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 automatical 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, e.g. to improve the layout of the new graph.

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 order of the rules to be applied is defined by their order in the grammar tree. Starting the interpretation, each rule is applied as often as possible, until no more match for this rule can be found. Then, the next rule is chosen and applied if possible. The graph transformation stops when all rules in the rule list have been applied in the given order as often as possible. The graph transformation finally stops if either there is no more rule applicable, or the user has stopped the transformation process.

7 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. Applications of AGG may be of a big variety because of its very flexible attribution concept relying on Java objects and expressions. The development group of AGG at the Technical University of Berlin will continue implementing concepts and results concerning verification and structuring of graph transformation, already worked out formally. AGG is available at: `http://tfs.cs.tu-berlin.de/agg`.

References

[CMR+97] 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 [Roz97], pages 163 - 246.

[EHK+97] 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 [Roz97], pages 247 - 312.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[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.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[Roz97] G. Rozenberg (ed.). *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1996.

[TFKV99] 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 (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution.* World Scientific, 1999.

[UML99] Rational Software Cooperation. Unified Modeling Language. Available at `http://www.rational.com`, 1999.