Chapter 14

THE AGG APPROACH: LANGUAGE AND ENVIRONMENT

C. ERMEL M. RUDOLF G. TAENTZER

Technische Universität Berlin, Fachbereich 13 Informatik,
Franklinstr. 28/29, D-10587 Berlin Germany
E-mail: {lieske,mich,gabi}@cs.tu-berlin.de

The Attributed Graph Grammar system (AGG) provides a visual programming environment which is based on graph transformation and Java. The idea is to program the topmost object structures of an application as well as its dynamics visually. For this purpose, these structures are designed as graphs which may be attributed by Java objects and further arbitrary data. The application's behavior is described by graph rules using an *if-then* programming style. Application of a graph rule changes the structure graph. Basic data types as well as object classes already available in class libraries may be used to program the application "in the small". Moreover, new classes may be included. This part of the application is done textually using the modern object-oriented programming language Java. The visually and the textually programmed application parts are combined by attributing the structure graphs by attribute values. These values may be data as well as object references. The graph rules may be attributed by Java expressions which are evaluated during rule applications. It's up to the programmer's decision how much of the application is realized visually and how much textually. Combining visual and textual programming in this way, programming should become easier because of visual concepts. On the other side, the wheel is not reinvented again, since well elaborated programming concepts, tools and class libraries mirroring the state of the art of programming can be used. The tool environment provides graphical editors for graphs and rules as well as a textual editor for Java expressions integrated into the visual editors. Moreover, visual interpretation and debugging of AGG programs is supported.

Contents

14.1 Introduction

Graphs play an important role in many areas of computer science. 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 all the kinds of diagrams used in OO-modeling methods like UML (see `http://www.rational.com`).

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. If graphs are used to visualize control flow like e.g. in Petri nets, graph transformation just manages its execution or – on a meta level – graph transformation can also be used to redefine control flow. Moreover, graphs are useful to describe software components and their dependencies. Typical operations of software development tools can be described by graph transformation, too.

These are only some examples for the use of graph transformation. To use it as programming paradigm, graphs have the possibility to carry attributes. Graph transformation is then equipped with further computations on attributes. As the reader might have noticed, graph transformation can be applied on very different levels of abstraction. Depending on this level, graph transformation is just attributed by simple computations or – on the other side – by complex processes.

It is useful to program with graph transformation if complex data structures have to be considered. A set of objects referencing each others forms a graph, and visual debugging, i.e. performing graph transformation step by step, supports the understanding of its dynamics. Since graph transformation is rule-based, algorithms which are complex in imperative languages because of lots of loops and recursion may be more easily implemented by graph transformation. Elementary operations are described by graph rules and the implicit control flow which applies each rule as often as possible realizes looping. See Section 14.3 for a graph transformation example where six rules implement the branch-and-bound algorithm to find shortest paths.

In the last years, visual development environments such as VisualBasic, Delphi and VisualAge, became en vogue. All kinds of documents needed for software development are visually managed. But the proper application programming is still based on text, except for special areas like programming of graphical user interfaces and database queries. On the other side there are visual programming languages like Prograph [3] and PictorialJanus [10], which offer the possibility to describe the whole program except for pure data visually. But these languages are so strongly addicted to visual programming concepts, that they are very limited concerning the integration of textual programming. Moreover, most of them have unsufficient structuring possibilities.

Compared to visual programming languages, AGG may be understood as a hybrid programming language which offers visual as well as textual concepts. Graph transformation on the visual side and Java[1] on the other side are integrated. Both concepts can be used flexibly. It's up to the programmer how much of the program is visual and how much is textual. The advantage of complementing new graphical concepts with a common textual programming language is a broad availability of programming know-how concerning concepts, libraries and tools. We use Java for this purpose, because Java is a modern, interpretated language which runs platform independent. In AGG we also chose the interpretative approach for graph transformation which is very natural in this context to watch graph modification. Since AGG itself is written in Java we get platform independence for the whole application developed with AGG.

The AGG environment supports the visual administration of graphs and rules, structured into graph grammars. Visual editing, interpretation and debugging are integrated in the AGG environment.

AGG has a formal foundation based on the single-pushout approach to graph transformation [5]. 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 the possibility for software verification. Verification features are not yet available, but will be one of the next extensions of AGG.

This article is structured as follows: In Section 14.2 the AGG language concepts are presented. They are illustrated by a running example which describes some aspects of a shipping company (see also [4]). Thereafter, one part of the running example, i.e. the search algorithm for shortest paths underlying the tour scheduling in shipping companies, is shown in detail. This sample application is also used to explain the AGG environment in Section 14.4. This article concludes with future developments planned so far for AGG.

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

14.2 Agg Language Concepts

Agg programs consist of two main parts: a graph grammar attributed by Java objects which may come from user-defined Java classes. This set of classes forms the second part. Clearly, libraries of Java classes such as JDK are available, but are not considered to be part of an Agg program. Graph grammars contain a start graph and a set of rules which may have negative application conditions.

The way how graph rules are applied realizes directly the single-pushout approach to graph transformation as presented in [11,5]. The formal basis for graph grammars with negative application conditions was introduced in [7]. The attribution of nodes and arcs by Java objects and expressions follows the ideas of attributed graph grammars as stated in [13,1] and further in [19] to a large extent. The main difference is that here, Java classes and expressions are used instead of algebraic specifications and terms. As long as we restrict the use of Java expressions to the subset covered by algebraic specification techniques, the whole transformation semantics of AGG coincides with the formal foundations. In particular, this means that we would have to avoid methods having side effects. The combination of attributed graph transformation with negative application conditions was first considered in [8] and extended to more general attribution concepts and conditions in [19]. The Agg features follow these concepts very close.

The Agg graphs and how they are attributed are presented in Sections 14.2.1 and 14.2.2. Graph rules and their application to graphs are discussed in Section 14.2.3. Section 14.2.4 introduces rules with negative application conditions and shows how they are applied. Last but not least, the combination of these application conditions with attributes is discussed at several examples in some detail.

14.2.1 Graphs

The Agg-system supports two different notions of a graph, which are, however, closely related. This section and the following one are to give a short introduction to the notion of the so-called *simple graphs*, which is, as the name implies, quite simple and straight forward. The other idea of a graph known in Agg is slightly more sophisticated in that it is a hierarchical one. Since the user interface of the implemented system is currently restricted to the editing of simple graphs, we will concentrate on this simple notion throughout the article. For those who are interested, however, Section 14.4.3 gives a brief overview of the hierarchical graph concept and how it can be seen as a generalization of simple

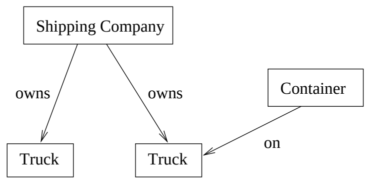


Figure 14.1: A shipping company modeled as a simple graph.

graphs[2].

A *simple graph* consists of two disjoint sets containing the *nodes* and the *arcs* of the graph. As a whole, the nodes and arcs are called the *objects* of the graph. Every arc represents a directed connection between two nodes, which are called the *source* and *target* nodes of the arc. To allow for some further classification of a graph object, any object may be associated with exactly one *label* out of a given label set. The labels are also called *types*, and if an object o has the label ℓ, we say "o is of type ℓ". Note that in our notion of a simple graph, we can have multiple arcs of the same type between a single pair of nodes, because every arc has an identity on its own, just like a node does. This fact distinguishes our view from another popular notion of a graph where an arc is described just as a relation between nodes.

What we have by now is a very basic idea of a graph, but it is yet sufficient to serve as an effective and intuitive means to some basic modeling tasks. For an example, consider Figure 14.1, where we have a small part of a shipping company modeled as a simple graph. The nodes of the graph are depicted as rectangles, while an arc is drawn as an arrow pointing from its source node to its target node. We have three types of nodes (*"Shipping Company", "Truck"*, and *"Container"*) and two types of arcs (*"owns"* and *"on"*). The interpretation of the graph is intuitively clear: We have a shipping company that owns two trucks, one of which is loaded with a container.

Note that in our idea of a graph, the position of a node or an arc in the plane holds neither syntactic nor semantic information, i.e., the layout of a graph is just a matter of presentation for the sake of readability to the user. Obviously, the layout of a graph may be considered as the equivalent to the indentation (or "pretty-printing") of a program in a conventional textual programming language like C: It is a well known fact that a program which is properly indented

[2]Actually, the hierarchical graphs are used for the internal representation of graphs throughout the system's implementation, and all the algorithms needed for graph transformation are based on this notion as well.

according to its logical structure is by far more comprehensible to the human reader than the same program put into one line, whereas to the compiler, both versions are equivalent. This experience is perfectly transferable to the layout aspect of a graph, and it makes clear that the layout is of considerable importance, although it does not bear any relevant information in the first place. Unfortunately, the problem to automatically compute a reasonable layout of a graph is much more complex than to pretty-print a textual program. In this article, we will not address the layout problem any closer, leaving it as an important issue for consideration in future time. de set, which serve as its source and target nodes.

14.2.2 Attributes

Let us reconsider the graph model of the shipping company in Figure 14.1. Obviously, our model is not very fine grained, as we are missing information about the name of the company, about the trucks' identity plates, and about the weight and content of the container, just to name a few. Doubtless, this kind of data is rather essential to the operations of a shipping company. As it is not feasible to represent such information purely by nodes and arcs, we need a means to attribute the objects of a graph with additional data like integer numbers or strings. We call this additional data *attributes*.

In AGG, an attribute is declared just like a variable in a conventional programming language: we specify a *name* and a certain *type* for the attribute, and then we may assign any *value* of the specified type to it. So to hold the name of our shipping company, we would define something like the following, and associate it as an attribute to our "Shipping Company" node:

```
String name = "Bringitfast Ltd."
```

Of course, there can be multiple attributes for one object, even of the same type. Assuming we want to keep information about the company's location as well, we may end up with the following attribute definitions:

```
String name = "Bringitfast Ltd."
String location = "Turtle Drive, Sunnyville, CA"
```

In this situation it becomes evident that the name of an attribute has to be unique among all the attributes of the same graph object; therefore, the name of an attribute is also known as its *selector*.

Figure 14.2 shows our model of the shipping company again, augmented with some sample attributes. In order not to overcrowd the display, type declarations are omitted in the representation of attributes. Of course the implemented system provides functionality to examine the complete definition of an

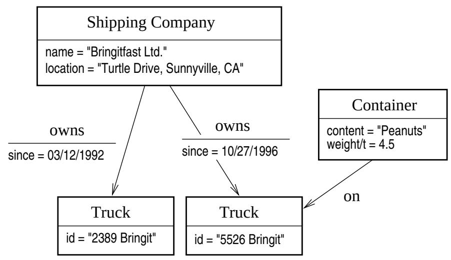


Figure 14.2: The example from Figure 14.1 augmented with attributes.

attribute on demand. The attributes named `since` associated with each of the two "owns" arcs show that arcs can be attributed in exactly the same way as nodes are. In this case, `since` has a value of type `Date` and specifies the date when the object pointed to by an "owns" arc has been acquired by the company.

Note that all graph objects of the same type also share their attribute declarations, i.e. the list of attribute types and selectors; only the values of the attributes may be chosen individually. For example, if we introduce another "Shipping Company" node to our model, it would have to have the same two attribute entries with the selectors `name` and `location` as the existing company, but the values may be freely chosen to reflect the actual name and location of the new competitor. From a conceptual point of view, attribute declarations have to be considered as an integral part of the definition of a type. This observation is not very surprising, since in many respects, the concept of a type with integral attribute declarations resembles the notion of a class with its member variables [3] in the paradigm of object-oriented programming.

An important question we did not yet deal with is: What types are actually available for the declaration of attributes? The answer is short and simple, yet emphasizes the affinity of the graphical AGG-approach to the object-oriented paradigm: The attributes may be typed with any valid Java type. This means that it is not only possible to annotate graph objects with 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 reveal 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 [4] to manipulate

[3] sometimes also called *class attributes*

[4] In the object oriented paradigm, procedures or functions associated with a class are

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. A simple, but non-trivial sample attribution based on a user-defined Java class will be explained in Section 14.3.

14.2.3 Graph Transformation

So far, we got an idea about how to model a situation of a given problem domain as an attributed simple graph. While graphical representation of states is nice, it is still not very exciting in the long run. Since the world is a moving place, we need a means to express dynamics, i.e. *operations* (or *actions*) on our graph model. For example, we may want to simulate some simple business processes of the shipping company modeled in our previous examples. To demonstrate the concept of how this is done in AGG, in this section we will see how the simple action of a container being loaded onto a truck can be expressed by *graph transformation*. Following the example, we will gain an intuition about the *single pushout approach* to graph transformation which is the formal foundation the AGG system is based on. A comprehensive description of the single pushout semantics for graph transformation can be found in Vol. 1 of this handbook [5], along with an overview of the main related theoretical results.

Graph Rules and Matches

An action can be viewed as a state transition, and obviously, a transition of states can be specified by giving descriptions of the states before an after the action in question. Since states are modeled as graphs in AGG, it follows that basically an action can be described as a pair of two graphs modeling the "before" and "after" states. In the "before" state of an operation, we collect all the preconditions that have to be met for the operation to take place. For our example operation "loading a container onto a truck", we may have the following requirements:

- There has to be a truck.
- There has to be a storehouse with at least one container in it.
- The truck to be loaded has to be in front of the store.

These requirements are easily assembled into a graph, as shown on the left-hand side of Figure 14.3. Once having figured out the "before" state, specifying

called *methods* of this class.

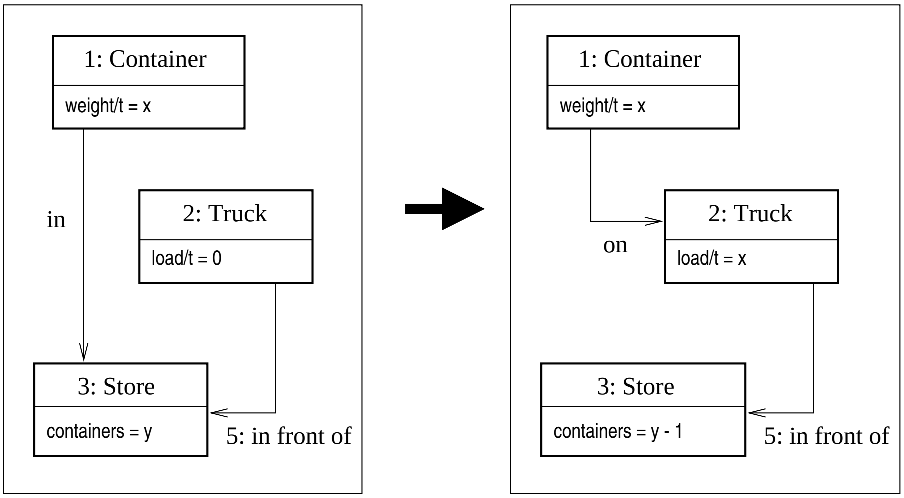


Figure 14.3: Graph rule to load a container onto a truck.

the appropriate "after" state is straight forward: it looks just like the "before" situation, just the arc representing the "in" relation between the container and the store is removed in favor of an "on" arc between the container and the truck. These two graphs describing the "before" and "after" states serve as the left and right-hand sides of a *graph rule* (Fig. 14.3) implementing our operation.

Note that the operation is *abstract* in that it does not specify a concrete container, truck, or store to operate on. Instead, the graph objects of a rule have to be considered as variables which are instantiated when the rule is *applied* to a concrete state graph. The most striking evidence for the abstract nature of the specification is that we use *variables* to refer to the attributes we want to operate on without regard to their concrete values. Irrelevant attributes like the ID of the truck are simply omitted from the specification.

As we have seen before, the left-hand side of a graph rule states the necessary conditions for the specified operation to take place: A rule can only be applied if its conditions are fulfilled by the current concrete state graph. Quite obviously, this corresponds conceptually to an *if-then* clause with an empty *else* branch in a textual programming language.

The checking of the conditions is accomplished by trying to find a *match* for the graph pattern given by the rule's left-hand side in the state graph, which is also called the *host graph* for the rule application. In terms of theory, matches and rules are *graph morphisms* , i.e. mappings of the objects of one graph to those of another with certain compatibility commitments concerning source and target mappings (cf. [5], Def. 1). To indicate which objects are mapped to one another in the figures, we use numerical tags preceding an object's type name, separated by a colon. Figure 14.4 shows an example application of our "load" rule to a concrete state graph at a given match m. The match determines exactly which of the containers found in the state graph is to be loaded onto which truck. At the same time, the attribute variables associated with the rule's objects are instantiated with the corresponding values of the state graph's objects they are mapped to. If a variable occurs several times in a rule's left-hand side, it has always to be matched with the same value.

You may notice that for every node or arc out of the rule's left-hand side, the match morphism m provides a corresponding object in the state graph G, whereas the rule morphism r does not map the arc (10) to any object of the right-hand side of the rule; we say that m is a *total* morphism while r is *partial*. This situation is typical for the so-called *Single Pushout (SPO) approach* to graph transformation pursued by AGG: a match has to be total because we want all the conditions imposed by the rule's left-hand side to be fulfilled, whereas the partiality of the rule is used to distinguish the objects to be deleted from those to be preserved.

Note that in general, we may find multiple matches of the rule's left-hand side into the state 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 state. In our example, we have two possible matches actually: In Figure 14.4, we may equally well choose (7) as the container to load instead of (1). However, we have no choice in which truck is to be loaded, since truck (6) is not "in front of" the store as the rule demands. In the case of multiple matching possibilities, one match has to be chosen. It fully depends on the application context whether this selection is done randomly or by preference, e.g. by user interaction.

Rule Application

The effect of a rule application at a given match is a state graph transformation, also called *derivation* or *graph transformation step*; in our example, it is the transformation from the state graph G to G' in Figure 14.4. The basic idea of what happens during a derivation 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 state 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 derivation, otherwise it is *removed*. Objects exclusively appearing in R without an original object in L are *newly created* during the derivation. Finally, the objects of the state 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 derivations.

There is one thing to watch out for when a rule is deleting nodes. Imagine a

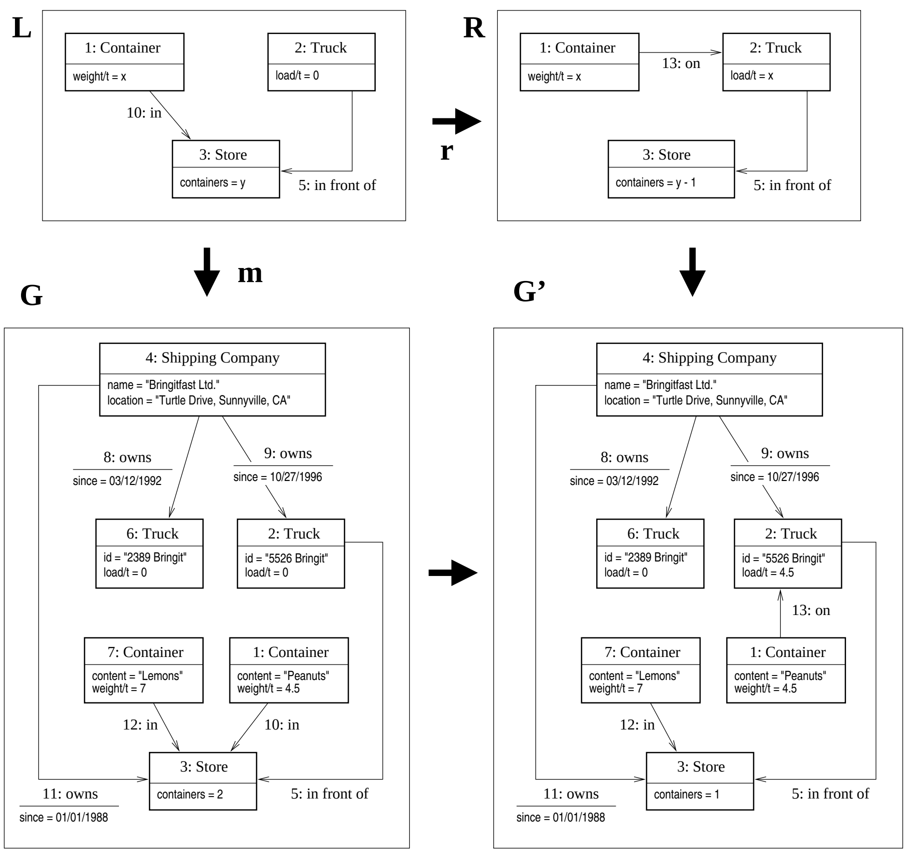


Figure 14.4: A transformation step by applying the rule from Fig. 14.3 to a state graph G at a match m.

rule for selling a truck described in a way that the "truck"-node is deleted. Applying this rule to node (6) in Figure 14.4 deletes not only this node, but also arc (8), since it would not have a target node anymore. Thus, dangling arcs are implicitly removed by a derivation 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. For example, the right-hand side of the rule depicted in Figure 14.3 contains the expression `y - 1` as a value for the integer attribute `containers` of the store. 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. From interfacing databases to calling sophisticated mathematical libraries or invoking interactive dialogues, everything is possible. Actual invocation of a method occurs whenever the expression is evaluated, i.e. every time the corresponding rule is applied. We are going to have a closer look at a sample method invocation in Section 14.3.

Note that the use of complex expressions to denote attribute values is limited to the right-hand rule side, whereas in the left-hand side, only variables and constant values are allowed. The expressions of a right-hand rule side may only refer to variables introduced in the left-hand side of the same rule. Finally, in the host graph we may only have constant values as attributes. Of course, "constant" does not mean that we cannot change these values: applying a rule which operates on an attribute simply means to replace one constant value by another one, which may be the result of the evaluation of a complex expression. Table 14.1 sums up the restrictions to the use of attribute expressions in different contexts. These restrictions may be less stringent in future versions of the system. For example, it may be useful to state expressions like "x+2" in the left-hand rule side; matching would then mean to solve a simple equation in order to obtain the proper instantiation value for "x". When we want to do meta modeling – i.e., applying rules to other rules –, it would also be necessary to allow arbitrary attribute expressions in the host graph. In this case, however, we would want to consider the host graph's attribute expressions as purely syntactical entities.

Please note that AGG is not limited to injective morphisms, as the impression may arise when looking at the simple examples presented by now. Any rule or match morphism may map two or more different objects to one single image object. Using non-injective rules, for example, it is easy to write a rule merging multiple nodes into one, contracting all the in- and outgoing arcs of the merged

	Constant	Single variable	Complex Expression
Example	42	x	$7 * x - 13$
Host graph	yes	no	no
Left-hand rule side	yes	yes	no
Right-hand rule side	yes	yes, if declared	yes, if all variables declared

Table 14.1: Restrictions to attribute expressions in different graph contexts.

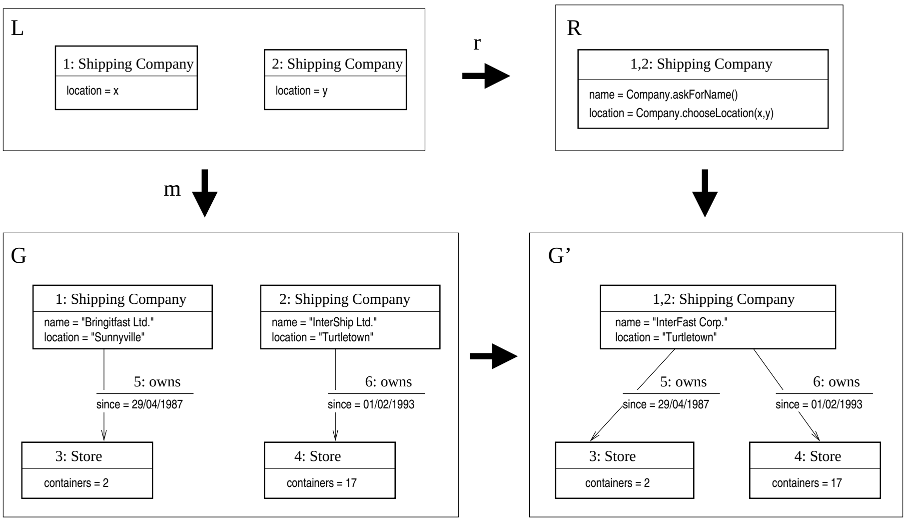


Figure 14.5: Graph transformation showing the fusion of two shipping companies.

nodes at the single resulting node. Consider the example in Figure 14.5 where the merging of two shipping companies is described by a graph transformation. Please notice that gluing of nodes also affects the arcs (5) and (6) which have a common source node (1,2) afterwards.[5] In additon, Figure 14.5 shows the usage of Java methods to determine the new name and location of the resulting shipping company. Java class `Company` is user-defined and offers the methods `askForName()` and `chooseLocation()` which are to pop up suitable dialogs for user interaction.

Moreover, also the rule matches may be non-injective, i.e. two or more graph

[5]Another way to model such a merging would be to delete one of the "Company"-nodes (1) and (2). However, applying such a rule to graph G would have the effect that one of the arcs (5) and (6) is deleted, too, which is not adequate.

objects in the left-hand side of a rule may be mapped to a single object in the state graph. But be aware that non-injective matches may cause conflicts between deletion and preservation: we could have, for instance, two graph objects in the left-hand side, one to be deleted and one to be preserved, and both mapped to the same image object in the state graph. In conflicts like this, deletion has precedence over preservation.

In the previous paragraphs, we gave an informal description of the semantics of rule application. An informal description like this is very helpful to get an intuition about what is going on, but it tends to be incomplete and ambiguous. That is why it is important to have a formal definition of the semantics in the first place. Another point is that a calculus based on formal semantics is by far more amenable to theoretical analysis and formal verification techniques. The formal semantics of rule application is given in terms of category theory, by a single categorical construction known as a *pushout* in an appropriate category of attributed graphs with partial morphisms - hence the name *Single-Pushout (SPO) approach*. Those who are familiar with category theory may have recognized that Figure 14.4 is a pushout diagram in the category of graphs and graph morphisms. The single pushout semantics for graph transformation ensures some of the most desirable properties for a derivation, in particular it guarantees that the transformation effect is

complete : Any effect specified in the rule is actually performed in the concrete derivation.

minimal : Nothing more is done than what is specified in the rule.[6]

local : Only the fraction of the state graph covered by the match is actually affected by the transformation.[7]

If dangling arcs and conflicting graph objects are forbidden and, moreover, all graph objects to be deleted are mapped injectively to the state graph, the well-known *gluing condition* is satisfied. In this case, the single and the double pushout approaches [2] yield the same results. (See also the comparison of both approaches in [5]).

To ensure compliance with the formal semantics definition, AGG's transformation engine is based on a library designed to perform arbitrary colimit computations [21] where pushouts fit in as a special case. Note, however, that AGG will never face you with a full pushout diagram like the one in Fig. 14.4, as

[6]With the well-defined exception of the implicit removal of dangling arcs and conflicting objects.

[7]Actually, the coverage has to be extended to include potentially dangling arcs.

AGG performs transformation steps *in-place*, i.e. the rules modify the state graph directly.

14.2.4 Negative Application Conditions

We have already seen that the left-hand side of a rule states the necessary conditions that the current state must fulfill so that the rule can be applied. Therefore, we may also call them *application conditions*. In fact, we're facing a *conjunction* of application conditions, since, considering Figure 14.3 for an example, there has to be a truck *and* a store *and* the truck has to be in front of the store *and* so on. Checking for all these conditions to be true simply means to find a match for the left-hand rule side into the state graph.

Quite frequently though, the need arises to express that something *must not* be there for a rule to be applicable. For instance, we might want to request that in order to be loaded, a truck must not carry a container already. We cannot directly express a *negative application condition (NAC)* like this with the means we already know. Sometimes, we can work around this problem by finding a positive formulation for the request. In Figure 14.3, we simply ask for a `load` value of 0 for the truck's weight to ensure that it is empty. As workarounds quickly grow to be inconvenient or inaccurate, we introduce a dedicated concept for the direct expression of NACs. Basically, we do this 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. Now, if we just put one container node into the negative condition graph N, it would mean "there must not be a container at all". But what we want to state is the condition "there must not be a container on the truck to be loaded", so we just put that whole constellation into N: a truck, a container, and an "on" arc connecting these two nodes. To express that the truck in N refers exactly to the truck to be loaded by the rule, we establish a morphism l between L and N which maps the truck in L to the one in N. Figure 14.6 shows the complete rule equipped with the negative application condition. A rule of thumb says:

> Within an NAC, you specify exactly that fraction of a matching situation that you *don't* want to happen.

Regarded this way, the representation of an NAC as a graph together with a morphism originating at the left-hand side of the rule becomes perfectly reasonable: Just consider N as a fraction of a potential host graph, and l as a part of a match into that host graph. Being a negative condition, the meaning of the whole situation is then "in a situation like N, don't match like l". For

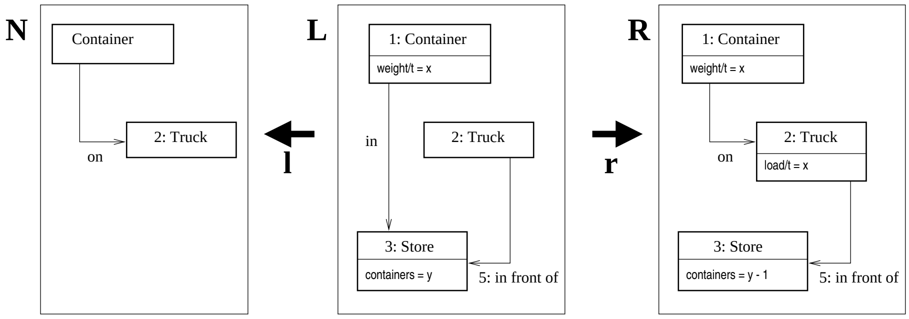


Figure 14.6: A negative application condition preventing a truck to be loaded if it is already occupied.

instance, in Figure 14.6 we have "when there is a truck carrying a container, don't take that truck as the one to be loaded".

Of course, the semantics of negative application conditions is again formally defined, which is especially important since complex NACs can become quite confusing. Considering the formal semantics brings ultimate clarification in these cases. Basically, the formalization says that 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 two graph objects being mapped to one another by l are mapped to the same object in the host graph G (more formally: there must be no total morphism $n : N \to G$ such that $n \circ l = m|_{dom(l)}$).[8] In the following paragraph, we are going to consider another sample NAC, and we will use the above definition to check if it actually does what we intend.

In our "load" rule, we may also want to postulate that no other truck than the truck to be loaded may occupy the loading bay in front of the store. Following the rule of thumb, we might come up with an NAC as shown in Figure 14.7. Given the state graph G depicted therein, our intention is obviously that the NAC should be fulfilled, since the only truck located in front of the store is the one to be loaded. However considering the formal definition, we can easily find a total morphism $n : N \to G$, mapping the store in N to the store (3) in G, the truck to (2) and the "in front of" arc to (5). Furthermore, the only object being mapped by l at all is the store (3), which is obviously mapped to the same store of the host graph both times. Thus according to our definition,

[8]Note that this definition is slightly differing from Def. 31 of [5] in that we allow partial NAC morphisms $l : L \to N$ while the original definition demands totality. Both representations are equivalent, though, and can easily be transformed into one another. We find the partial representation to be more compact and intuitive.

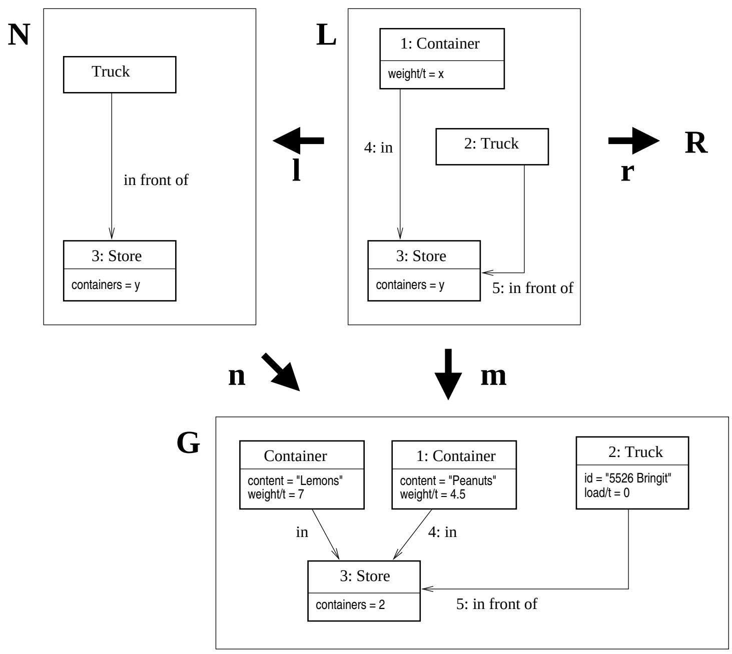


Figure 14.7: An inconsistent negative application condition causing the rule never to be applicable.

the NAC is *not* fulfilled, with the consequence that the rule is not applicable in the given situation. In fact, the rule will never be applicable at all, since we are facing an example for how an NAC may contradict the prerequisites stated in the left-hand rule side.

The problem with the NAC as stated in Figure 14.7 is that it actually says "there must not be a truck in front of the store *at all*" instead of "there must not be any *other* truck in front of the store". The solution to this problem is to include another "Truck" node into the NAC, and mapping the truck (2) of L which is the truck to be loaded via l to this new node. Additionally, we claim that the total mapping n of the NAC into the host graph has to be injective on that part not coming from L, so that the two trucks of the NAC may never be mapped to one and the same truck in the host graph.[9] Under

[9]Restricting NAC mappings in this way is reasonable since the effects of non-injective satisfaction are counter-intuitive quite frequently. Formally spoken, the restriction is that graph part $N - l(L)$ has to be mapped injectively by morphism n.

these preconditions, the rule with the augmented NAC is now applicable when it should be, as can be easily verified.

By now, we developed two separate NACs for the "load" rule. Most likely though, we want *both* NACs to be satisfied for each application of the rule, i.e. *neither* the truck shall be already loaded, *nor* shall there be another truck blocking the loading bay. Cast into logical notation, the condition we want to express is of the form $\neg N_1 \wedge \neg N_2$, i.e. a conjunction of NACs. But how can we express this coherence in graphical terms? We might assemble the positive aspects N_1 and N_2 into one graph representing the conjunction $N_1 \wedge N_2$ of these conditions, just like the left-hand side of a rule represents a conjunction of positive application conditions. However, using such a graph as an NAC will have the effect of $\neg(N_1 \wedge N_2) = \neg N_1 \vee \neg N_2$, yielding a disjunction of NACs since the negation applies to the NAC graph as a whole. Instead, we allow a negative application condition to consist of multiple NAC graphs N_i and morphisms $l_i : L \to N_i$, where the pairs (N_i, l_i) are called the *constraints* of the NAC.[10] We then define an NAC to be satisfied if all its constraints are satisfied, where the above definition about the satisfaction of NACs turns out to be about constraints, actually. Figure 14.8 shows diagrammatically how a conjunction of multiple constraints is constructed.

$$\begin{tikzcd} \mathbf{N_1} & & \\ \mathbf{N_2} & \mathbf{L} \arrow[l] \arrow[ul] \arrow[ddl] \arrow[r, "\mathbf{r}"] & \mathbf{R} \\ \vdots & & \\ \mathbf{N_n} & & \end{tikzcd}$$

Figure 14.8: Scheme of a rule with multiple NACs.

Up to now, we only considered NACs with injective mappings from L to N. But there is one special situation where it comes very handy that we are not restricted to injectivity: in the case that we allow non-injective matches for rules in general, using non-injective NACs allows us to enforce injective matching just for particular objects. Recall the rule of thumb: by mapping two objects of L to the same object in N, we can specify that we don't want this identification to happen in an actual match.

So far, we deliberately ignored the role of attributes in negative application conditions. And in fact, the most common use of NACs is to state graphical conditions where the attribute values do not matter, as was the case for all of our previous examples. You may have noticed that we quietly copied the

[10] Most of the time, we will sloppily speak of a rule having multiple NACs.

attribute values of the left-hand side graph objects to the corresponding objects of the NAC graphs, which is, by the way, exactly the behavior you will observe when defining NACs within AGG. This is to ensure that any two objects mapped to one another by an NAC morphism l_i can safely be mapped to the same object in a host graph. But it is worthwhile to mention that we can also use NACs to express some simple negative conditions on attribute values: whenever we define a variable (or no value at all) for an attribute's value in the left-hand rule side, we may specify a concrete value for that very same attribute in a corresponding object of an NAC graph, meaning that we do not want this specific instantiation to happen. Again, this fits nicely into the intuition gained by the rule of thumb presented above. Figure 14.9 shows a sample NAC ensuring that the container to be loaded will not be empty. However, currently there is no convenient way to express conditions like "do not exceed the maximum admissible load of 12 t". Section 14.5 gives an outlook on how future developments will address these shortcomings.

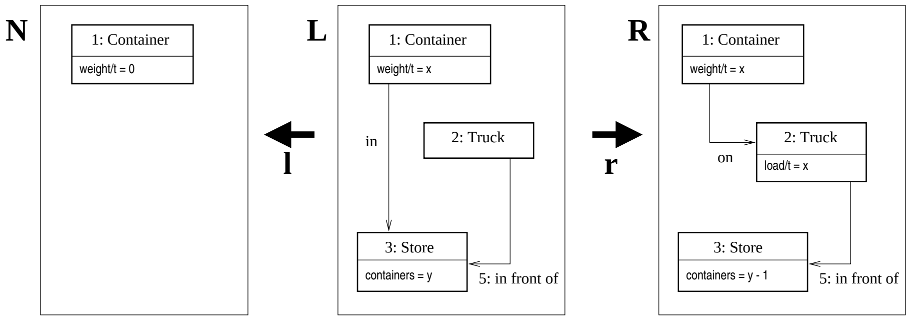


Figure 14.9: Using a NAC to avoid particular variable instantiations.

14.3 Sample Application: Finding the Shortest Path

In this section, we present a small, but non-trivial sample graph grammar to illustrate the features of the AGG system and the advantages of visual programming. For this purpose, we choose the well-known "shortest path" problem which is also an essential part of the tour planning in shipping companies. Search procedures are ubiquitous, popping up whenever alternatives have to be explored to solve a problem. Here, we illustrate a route finding procedure. Given a map (in our example, the map shows some cities, connected via highways), the procedure finds the shortest path from a start position (a certain

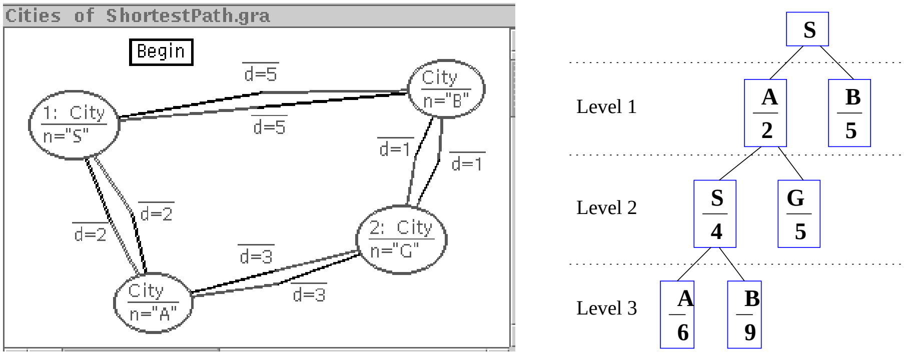


Figure 14.10: The Branch-and-Bound Algorithm

city) to a goal position (another city).

The most obvious way to find a solution is to devise a bookkeeping scheme that allows orderly exploration of all possible paths. It is worthwhile to note that the bookkeeping scheme must not allow itself to cycle in the graph. It would be senseless to go through a sequence like *Start* - *City*1 - *Start* -*City*1 ... over and over again. With cyclic paths terminated, graphs are equivalent to trees. Therefore, we will generate the search tree corresponding to our road map graph. Drawing the children of one tree node is called *expanding* the node. Tree nodes are said to be *open* until they are expanded, whereupon they become *closed*. The *length* of a path in the tree is given by adding the distances between the path nodes.

To solve our problem we have to look for the shortest path in our search tree from the root node to the goal node. We define the problem by drawing the graph to be explored as the start graph of our graph grammar. Nodes in the start graph are cities (type *City*), a directed edge models the highway connection between two cities, the length of a highway is encoded as edge attribute. Of course, there always have to be two edges with opposite directions and the same distance value between two cities to denote that highways can be used in both directions. The left picture of Figure 14.10 shows an example road map as typed, directed, attributed graph, drawn with the AGG graph editor. Directions of edges are denoted by colours in AGG: The edge part connected to the source node is drawn black, the part connected to the target node has a lighter colour. Labelling *City* nodes is realized by assigning values to the `String` attribute `n`.

We use the *Branch-and-Bound* algorithm, as described in [20] that builds a search tree by tracing through possible paths beginning at the start node, compares the length of the partial paths already found and expands the open node at the end of the shortest partial path. Expansion continues until there is a path reaching the goal that is of length equal to or shorter than all incomplete paths terminating at open nodes (leaves).

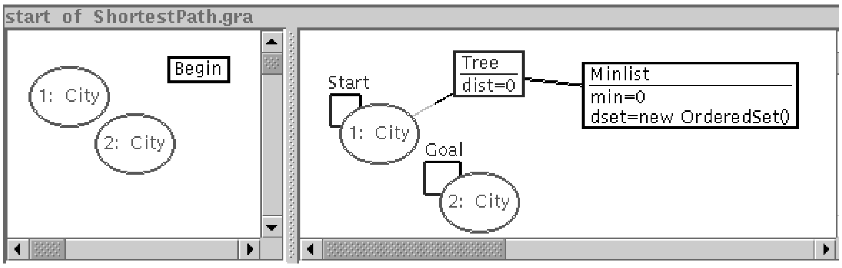
Let us consider the example in Figure 14.10 and assume we want to find the shortest path from the start node labelled "S'' to the goal node labelled "G''. Looking only at the first level of the corresponding search tree on the right-hand side of Figure 14.10, the distance from S to node A is clearly less than the distance to B. Expanding A at level 2 leads to a path containing the goal node. But it reveals at the same time another open path with a shorter length (so far), namely the way back to node S. Therefore, this node has to be expanded once more, leading to the two open paths in level 3. Here we see that there is no point in calculating the path length for any of the three open paths through the nodes A or B since any of the path's length is at least 5 already, and hence longer than the path to G for the known solution through A.

We see, the algorithm terminates in spite of circles in the original graph. Optimizations of the basic branch-and-bound search algorithm are given in [20]. Here, we are interested in demonstrating an intuitive implementation of the basic algorithm using attributed graph grammars in the AGG environment.

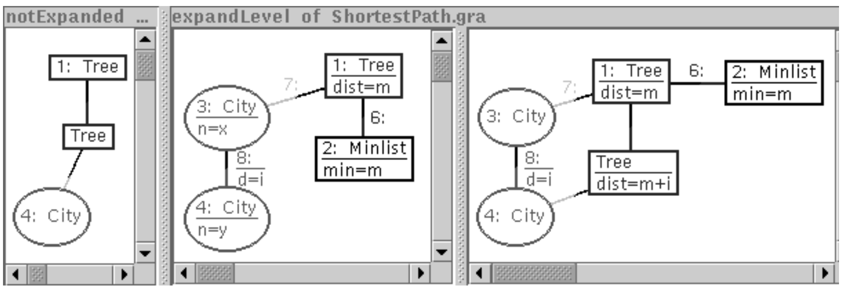
The road map shown in Figure 14.10 is the start graph of our graph grammar. The rules to be applied to the host graph realize the branch-and-bound algorithm. Each rule is applied as often as possible, transforming the host graph "in place". If a rule is not applicable any more, the next rule is tried. Matches (i.e. mappings from the left-hand side to the host graph) are computed automatically or defined by the user. Our graph grammar *ShortestPath.gra* consists of the following six rules:

1. Before applying the *start* rule, the user identifies the start and goal nodes of the connection he wants to know the shortest traveling length of by defining a partial match that is, by mapping the two $City$ nodes of the left-hand rule side to two $City$ nodes of his choice in the search graph. Rule and match morphisms are denoted in AGG by equal numbers for each mapping. Thus, the $City$ node with the number 2 will be marked by a $Start$ loop, whereas $City$ node no. 3 is defined as the goal node of the search procedure.

   The rule generates the root of the search tree and the initial bookkeeping node (type $Minlist$) that contains a list of the current path lengths and stores the current minimum value after each expansion step. Tree nodes are connected by edges with their corresponding $City$ nodes. The deletion of the *Begin* node makes sure that the *start* rule can be applied to the start graph only once.



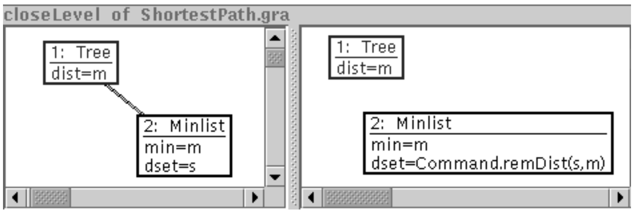
2. The *expandLevel* rule expands a node with the current minimum value that is, it generates the next level of the search tree. The negative application condition (short NAC, shown left to the left-hand rule side) ensures that no child is generated twice. The (partial) mappings from the left-hand rule side to the NAC are again shown via equal numbers. This rule has to be applied as often as possible, until all children of all nodes with the current minimum value have been found.



3. The rule *closeLevel* deletes the connection between the *Minlist* node and the previously expanded node as only values about paths to open nodes are stored, and the previously expanded node is now closed. At the same time, the list entry of the path to the previously expanded node is deleted from the distance list `dset` at the *Minlist* node. The deletion is realized by a call to the Java `OrderedSet` class method `s.remove` from the Java library `com.objectspace.jgl`[11].

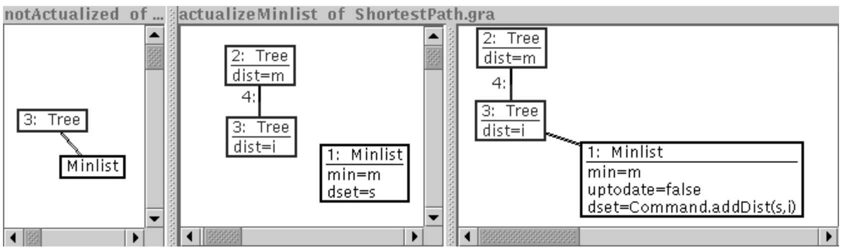
   When the rule *closeLevel* is applied as often as possible, all previously expanded nodes are closed.

[11] Java Generic Collection Library



4. The rule *actualizeMinlist* generates an edge between the *Minlist* node and each new leaf node in the tree that has just been generated by the *expandLevel* rule. The NAC ensures that only one edge will be generated between the *Minlist* node and each new leaf node. The distances between the start node and the new leaf nodes are stored in the *Minlist* attribute `dset` containing all distances of partial paths. Again, a Java `OrderedSet` class method is called to add distances to `dset`, namely the method `s.add`.

   Setting the *Minlist* attribute `uptodate` to `false` means that the computation of the minimum of the values in the distance list is now out of date. The correct minimum will be computed in the next rule *calcMinimum*. The rule *actualizeMinlist* has to be applied as often as possible, until the *Minlist* node is connected to all leaves in the search tree.



5. Applying the rule *calcMinimum*, the current minimum of all open paths, stored in the *Minlist* attribute `min` is actualized accordingly by calling the Java method `calcMin(OrderedSet s)` from the user-defined class `Command` which is shown in Figure 14.11. To ensure that the rule can be applied only once, right after the actualization of the distance list, the attribute `uptodate` is set to true again after one rule application.

6. The *solution* rule realizes the termination condition: If the goal node is reached in a current path *and* the distance from the start node to the goal node is equal to the minimum stored in the *Minlist* node attribute `min`, then the minimum is the solution, and the path in the search tree from the start node to the goal node represents the shortest connection between the two *City* nodes in the graph. The solution goal node in the search tree is marked by a *Solution* loop. This rule can be applied only once, because the *Minlist* node is removed when the solution is found.

If the *solution* rule cannot be applied (because no path containing the goal node is yet found, or because the distance to the goal is longer than the length of other partial paths), the search must be continued by jumping back to the *expandLevel* rule. This jump is realized in AGG by starting the graph transformation again with the current host graph. As the start rule is applicable only once (because of the *Begin* node being there only in the beginning), we then actually realize a loop over all other rules that is, a jump back to the *expandLevel* rule.

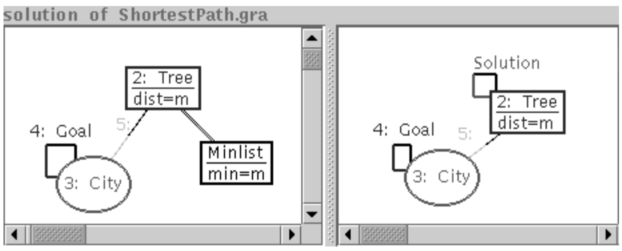


Figure 14.12 shows the rule application strategy just described. The application mode (*) means *apply as often as possible.* For three rules, we actually realize the mode (1) meaning *applicable only once.*

A comparison of our *Shortest-Path* graph grammar solution with a possible textual implementation of the algorithm given in [20] in an imperative programming language yields the following advantages of our visual approach:

The problem which is given as a graph (city/ road map), can be implemented directly and needs not be translated to textual data structures like linked lists, matrices, etc.

```
import java.util.*;
import com.objectspace.jgl.*;

public class Command
{
 public static OrderedSet addDist(OrderedSet s, int i)
 { s.add(new Integer(i));
  return s;
 }
 public static OrderedSet remDist(OrderedSet s, int i)
 { s.remove(new Integer(i));
  return s;
 }
 public static int calcMin(OrderedSet s)
 { Enumeration e=s.elements();
  int min = ((Integer)e.nextElement()).intValue();
  return min;
 }
}
```

Figure 14.11: User-Defined Java Class Command.java

Rule name	*Application*	*Description*
start	**(*) = (1)**	initializes search tree
expandLevel	**(*)**	generates all leaves for all open minimal paths
closeLevel	**(*)**	deletes old min. value and disconnects Minlist from all closed nodes
actualizeMinlist	**(*)**	connects Minlist to all new open nodes and actualizes distance list
calcMinimum	**(1)**	computes minimum of the current distance list
solution	**(*) =(0)|(1)**	if match: shortest path is found if not: continue with rule expand

finished (no more rule is applicable)

Figure 14.12: Rule Application Strategy for the Find-Shortest-Path-Example

The underlying structure of the solution algorithm is a tree, and hence can also be implemented visually, which makes the algorithm easy to understand and to debug, especially as the algorithm steps are realized by rule applications, and their effects are shown directly in the current state graph during interpretation.

The computation of the minimal path length is not done visually in our example but by using the sophisticated Java class `OrderedSet` from the Java library `JGL`. A visual realization would be rather complex and result in several additional rules. These rules should be hidden from the main algorithm rules to obtain a well structured implementation. Up to now, a hierarchy of rules cannot be realized in AGG.

Control Structures like loops are given implicitly by the AGG inherent rule application strategy, i.e. by multiple rule applications and by jumping back to the first grammar rule after all rules have been applied. A loop termination condition is reached if there are no more matches for the current rule. The program terminates if there are no more matches for any of the rules.

14.4 AGG Environment

The AGG-environment is designed as a tool to edit directed, typed and attributed graphs in order to define a graph grammar (i.e. a start graph plus a set of transformation rules) as input for the graph transformation component of the system. Moreover, visual interpretation and debugging of attributed graph grammars is supported. While debugging means performing direct transformation steps for user-selected productions and occurrences in the single-pushout approach to graph transformation, a whole transformation sequence is executed in the interpret modus. In the following, first the editing and afterwards the interpretation facilities are presented at the sample application introduced in Section 14.3.

14.4.1 Editing

The editing environment of the AGG system provides a comprehensive functionality for the input and modification of attributed graph grammars with a mouse/menu-driven user interface. There are four different editors: two graphical editors for graphs and rules, a textual editor for Java expressions as attributes which is integrated into the graphical editors, and a window to edit the names of the graph grammars and their start graphs and rules. The rule editor supports the editing of rules consisting of a left hand and a right hand rule side and (optionally) one or more negative application conditions (NACs).

A rule may be applied to the start graph in the graph editor which then is transformed according to the rule and the match into a so-called host graph, to which further rules may be applied.

The AGG Graphical User Interface

Figure 14.13 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 file loaded. This is visualized by a GraGra tree, the current graph grammar, start graph or rule being highlighted. In Figure 14.13 there is only one graph grammar, namely *ShortestPath.gra* consisting of the start graph *Cities* and six rules (two of them having NACs). In the GraGra tree window the names of graph grammars, start graphs, rules and NACs can be edited and the rule or graph to be edited or used for transformation steps can be selected here by clicking on its icon. The selected graph or rule is then shown in its graphical editor.

The File menu contains means to load or save a graph grammar file, to create or delete its components and to exit the system. Each graph grammar consists of one start graph and a number of rules. The menus Edit and Edit Modes allow to edit graphs, rules and attributes. The Transform menu is visited for interpreting and debugging graph transformation. Here, the definition of a match and the execution of a transformation step or sequence is possible. The screen shot in Figure 14.13 shows the first rule *start* of our Shortest-Path example in the rule editor and the result of the *start* rule having been applied to the road map shown in Figure 14.10 in the graph editor.

In this chapter, we describe how to create and delete graph grammars and their components, how to define types for graph objects, how to draw nodes and edges in the graphical editors, how to edit rules, and how to add attributes making use of the textual attribute editor and externly defined Java classes.

Creating and Deleting Graph Grammars and their Components

In the File menu, a new graph grammar may be created by clicking on the menu entry New and choosing the item GraGra. A new graph grammar named *GraGra*0 is created then, together with the start graph named *Graph* and one rule named *Rule*0 by default. These new components are shown in the tree view window where the default names can be changed now by clicking on the name text field twice. The default edit mode is the *Draw* mode, and the new empty start graph and the empty rule *Rule*0 have been loaded into their graphical editors where graphs may be drawn.

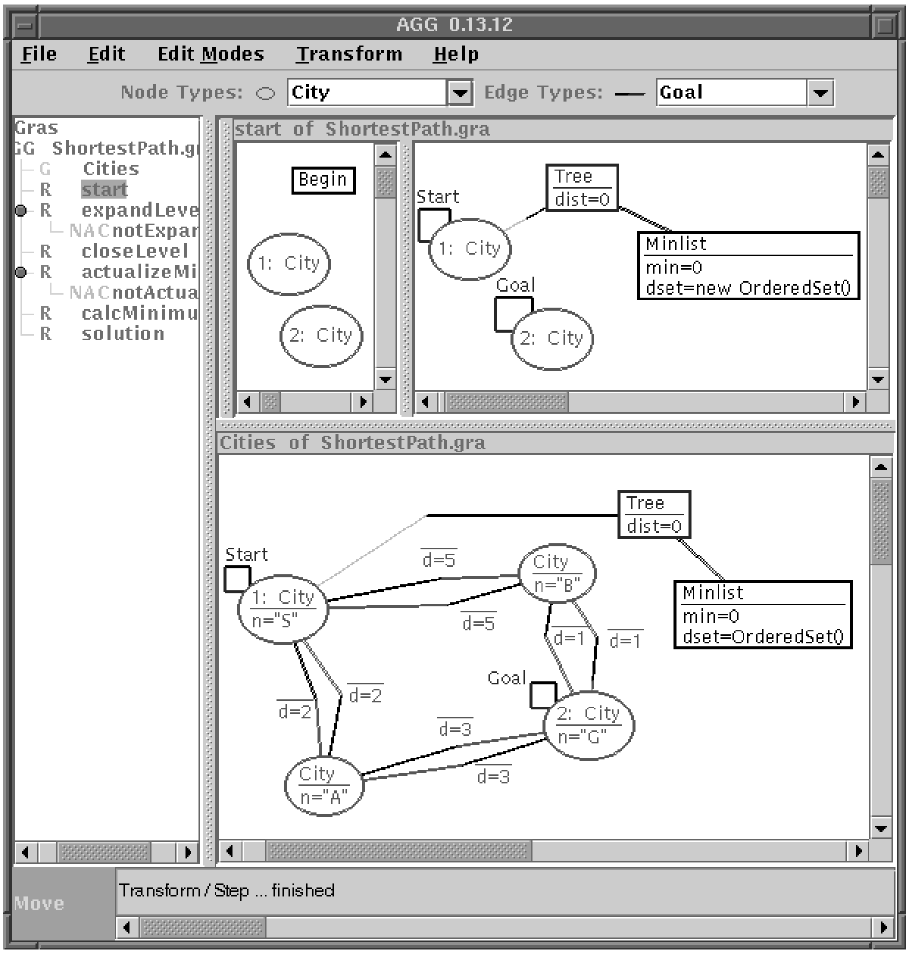


Figure 14.13: The AGG system Graphical User Interface

To create a new rule for the currently selected graph grammar or a new NAC for the currently selected rule, the menu entry New from the File menu has to be clicked followed by choosing one of the items Rule or NAC. Again, the newly created items are added to the tree view where they can be renamed or selected for editing.

A graph grammar, a rule or an NAC is deleted by selecting the File menu item Delete while the graph grammar, rule or NAC to be deleted is selected (highlighted in the tree view).

Defining Types for Graph Objects

The graphical editors have a limited set of elementary graphics to represent nodes and edges. Nodes may be shaped as rectangles, rectangles with rounded corners, ovals and circles. Edges may be solid, dotted or dashed lines. Both nodes and edges may have different colours. A type name may be added to identify a graph object's type but is not necessary. For example, there might be defined the node types *red rectangle*, *blue rectangle*, *red circle with type name* City and *red rectangle with type name* City. These are four different node types. The type of an object has to be defined before actually drawing the object. A type is defined by clicking on the graphical Node Type (or Edge Type) symbol and selecting a graphical shape and/or colour in the context menu of colours and shapes/styles. Then the type name must be entered into the Node Types or Edge Types text field. If no type name is wanted, then at least the return key must be pressed in the Node / Edge Types text field. Then, the new type is store in the Node Types or Edge Types list. It can be accessed by mouse click whenever a new object of this type is to be drawn. The current type is indicated by the graphical symbols next to the Node / Edge Types text field and the name inside the text field. In the graphical editors, the type name (if any) is shown inside the node shape or next to an edge.

Drawing Nodes and Edges

Having defined node and edge types, the user changes to the *Draw* mode by selecting *Draw* in the menu Edit-Modes. Now nodes may be drawn by clicking the left button on the panel of the graph or rule editor. An edge between two nodes is created by clicking at first on the source and then on the target node of the edge using the left mouse button. Edges are half black to indicate the source and half another color to indicate the target. An edge may contain one bend. There are two possibilities to draw a bend: Firstly, the bend can be inserted when creating the edge by clicking on a place in the panel between clicking on source and target nodes. Secondly, the bend can be created or moved being in the Move mode of the editor by grabbing the edge with the left mouse button at the point of color change and dragging it to the desired place in the panel. Bends can be removed by selecting one or more edge with a bend and choosing the item Straighten_Edges in the Edit menu.

Being in the Select mode, a graph object can be selected by clicking on the object. A selected object is colored green in the editor. Selecting an object twice means deselecting it. Selected objects may be duplicated, moved, deleted or attributed (all Edit menu operations). An alternative to change between Edit Modes is to use the context menu Edit Modes that pops up when clicking the right mouse button with the cursor placed on a free space in the editor panel. Here, the user can switch between general Edit modes (like Draw, Select, Move, ...). The mode then is valid for the local editor panel. If an object is moved outside the current editor panel, the size of the panel will be enlarged. The scroll bars then allow to access the invisible object. Another context menu containing graph object specific operations is evoked by a click on the right mouse button if the cursor is placed right onto a node or the point of color change of an edge. The operations here (like evoking the attribute editor, Delete, Duplicate, Select) are valid only for the graph object the menu had been opened for. A third operations menu pops up when the graph object the cursor is on is selected before evoking the menu. Figure 14.14 shows the different context menus.

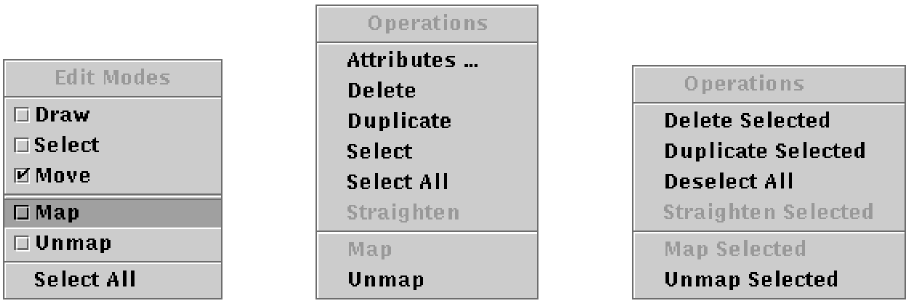


Figure 14.14: Context Menus for Editing Operations

Editing Rules

Editing a rule, it is advisable to draw the objects of the left-hand rule side first, and then choose the menu item Identic-Rule in the Edit menu. Thus, the graph objects of the left-hand rule side are copied to the right-hand rule side, and an identic rule morphism is generated. The rule morphism is indicated in the editor by showing identic numbers for each graph object on the left-hand side and its image on the right-hand side of the rule. Now, having two identic rule sides, the graph objects that are to be deleted by the rule, have to be deleted in the right-hand rule side, and graph objects that are to be created by the rule have to be drawn there additionally. Another possibility to draw the same rule would be to draw the part of the left-hand rule side first that is to be preserved by the rule, then choose Identic-Rule and afterwards add those graph objects to the left-hand rule side that are to be deleted and those graph objects to the right-hand rule side that are to be created by the rule.

An alternative to define a rule morphism is to draw both rule sides first, and then change to the edit mode Map. In this mode, the rule morphism can be defined elementwise by clicking on a graph object in the left-hand rule side first, and then clicking on a graph object of the same type in the right-hand rule side. This method is useful when there is already a rule morphism and the user wants to change it.

Analogously, an NAC may be entered by drawing the NAC graphical objects into the NAC graphical editor and then defining the mappings from the left-hand rule side to the NAC being in the Edit Mode Map by clicking elementwise on a graph object in the left-hand rule side first, and then clicking on a graph object of the same type in the NAC graph. The attribute values then are copied from the left-hand rule side to the NAC. The Edit menu option Identic NAC is deactivated in the current AGG version due to problems with the mapping of variables and attributes, but will be available soon.

Adding Attributes to a Graph Object

Attributes of graph objects have three components: the attribute type, its name and its value. Taking Java objects as attributes, the corresponding Java class defines the attribute type. The object itself is the attribute value. A survey of the different attribute values that are allowed for the host graph and the rule sides is given in table 14.1 on page 564.

In AGG, the attribution of objects is done in a special text editor that pops up when a graph object is selected in the graphical editor and the Edit menu item Attributes... is clicked. Another possibility to invoke the attribute editor is to switch to the mode Attributes in the Edit Modes menu and then click onto the object whose attributes are to be edited. The mode Attributes is also available in the pop-up menu Edit Modes, see Figure 14.14. The attribute editor allows to declare a *type*, a *name* and a *value* for an attribute. There may be multiple attributes for one graph object. As the underlying attribute handler interprets expressions in Java syntax, the types for attributes are either primitive Java types (i.e. byte, short, char, int, double, long, float, boolean) or Java objects. Java expressions are arithmetic expressions (e.g. `x + 1`), boolean

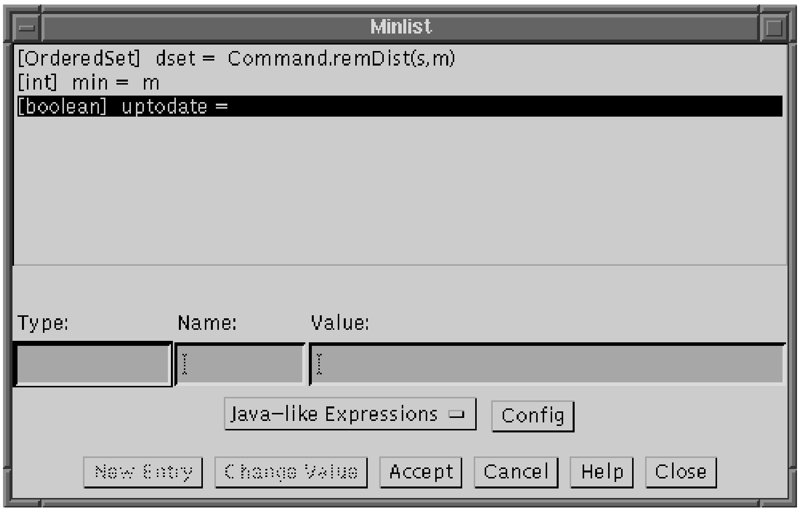


Figure 14.15: The Attribute Editor

expressions (e.g. `width <= 7`), bit manipulations, conditional expressions (e.g. `v>8?7:0`), 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. `java.lang.Math.sin(0.5)` or `v.addElement(new Integer(i))`). Apart from the standard Java library *JDK*[12], other user-defined classes can be used for attribution. The CLASSPATH environment variable and the class libraries have to be adapted accordingly. Alternatively, a configuration editor can be used.

An example for a user-defined Java class is the class `Command` of our Shortest-Path application, defining the operations for adding a distance value into the list of all open path lengths (method `addDist`), for removing a value from the list (method `remDist`) and for calculating the minimum value of all list elements (method `calcMin`) in Figure 14.11.

Figure 14.15 shows the attribute editor for attribute declaration and value definition. Here, the type of the graph object whose attributes are edited, is *Minlist*, the type of the bookkeeping node managing the list of path lengths and calculating the current minimum value after each expansion step. The attribute editor has been invoked for the *Minlist* node in the right-hand side of the rule *closeLevel* shown in the previous section. Type *Minlist* has two

[12] Java Development Kit

attributes set to variables or expressions, the ordered set `dset` itself (attribute type `OrderedSet`) and the minimum `min` (attribute type `int`). The value of `min` is set to the variable `m` (which has been defined in the left-hand side and is bound to a concrete value when determining a match for the rule). The value of `dset` is the return value of the method `remDist(s,m)` which is the set `s` without the element `m`. The third attribute `uptodate` is not bound because it is not needed in this node. In the current version of AGG, the order of attributes in the attribute editor cannot be changed.

Let us shortly explain the other window elements of the attribute editor: The field Java-like Expressions shows the current attribute handler. Other available attribute handlers may be selected from this list if available. For details concerning attribute handling see Section 14.4.3 on implementation issues.

Pressing button Config the configuration editor pops up and the addition of new class packages can be declared. In our Shortest-Path grammar, for example, we here declared the Java class package `com.objectspace.jgl`. Clicking on New Entry, a new attribute (consisting of name, type and value) may be edited. Change Value allows to change the value of an attribute that is selected in the attribute list on top of the window (the same effect has a double click on the attribute). With Accept, the input of an attribute is confirmed and finished (same as pressing the return key after an entry), with Cancel, the attribute input is stopped. The field Close closes the window and all accepted attributes are shown under the type name of the respective graph object in the graphical editor. Whenever for another graph object of the same type the attribute editor is opened, the names and types of all attributes edited for this type before are shown in the list. Only the values have to be entered again for each new graph object.

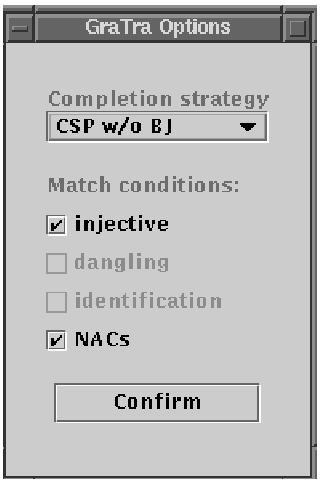
14.4.2 Interpretation and Debugging

In this section, we explain the definition of match morphisms and the different modes for rule application to the host graph in the context of our running example.

The Debug Mode

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 match morphism in the *Debug* mode may be defined by the user analogously to a rule morphism:

From the menu Transform, we choose the menu item MatchDef and define the match morphism elementwise by clicking on a graph object in the left-hand rule side first, and then clicking on a graph object in the host graph. Of course, the corresponding graph objects must be of the same type, and there must be a consistent mapping of the attributes. Another possibility to define a mapping is to choose the Map option for one graph object from the Edit Modes context menu and then click on the corresponding object in the host graph. Of course, a mapping also can be deleted by choosing Unmap from the same menu. Defining the match completely "by hand" may be tedious work. Therefore, AGG allows to complete a partial match by choosing the Transform menu item NextCompletion after having defined some (or none) arbitrary mappings of the match. The partial morphism will be completed then to a total match automatically. If there are several choices for completion, one of them is chosen arbitrarily. Calling NextCompletion again, another completion is computed if possible. In this way, all possible completion are computed and shown one after the other in the editor using equal numbers for the source and target object of each mapping. After having defined the match, we choose the Transform menu item Step, and 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, for instance to improve the layout of the new graph. Afterwards, any other rule may be selected and applied to the new graph in the *Debug* mode as described above.



For a transformation, options can be chosen by clicking on the Options item in the Transform menu. Here, a completion strategy may be defined (Constraint

Solving Problem, Constraint Solving Problem without BackJumping, Simple Backtracking, for details see Section 14.4.3) and match conditions may be fixed (injective or not, with NACs or not). This option allows to adapt the completion strategy to the graph grammar. For a very simple host graph (for example without attributes), the simple strategy Simple Backtracking might be the fastest, whereas for our Shortest-Path example we chose Constraint Solving Problem without Backjumping due to problems with attribute evaluation in NACs using the default strategy Constraint Solving Problem.

Let us illustrate the transformation procedure transforming the start graph of our Find-Shortest-Path example using the rules introduced in chapter 1. We will go through a transformation session in *Debug* mode in order to show all important intermediate transformation steps.

The transformation options chosen for our example are to use injective matches, rules with NACs and the completion strategy *Constraint Solving Problem without BackJumping*:

A Sample Run

Figure 14.10 shows the start graph of grammar *ShortestPath.gra*. To determine the start and goal cities the user defines a partial match for rule *start* by matching the two cities in the left-hand side only. After completing this partial match the transformation result is shown in Figure 14.13. The match defined City S to be the start and City G to be the goal of the search. The initial tree

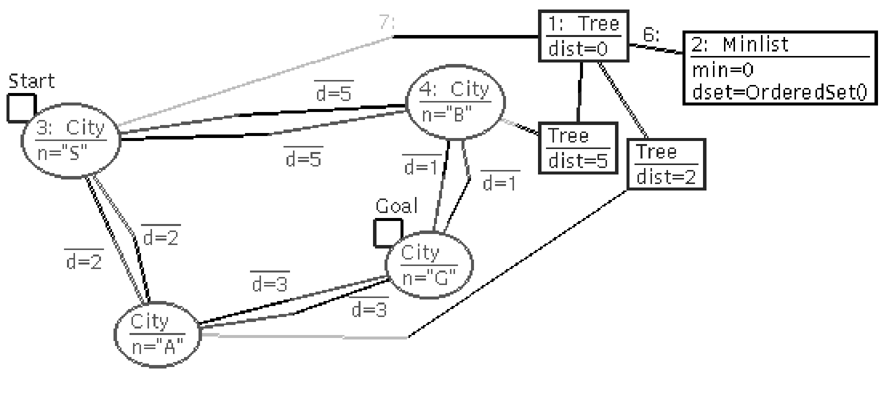


Figure 14.16: Host Graph after applying the *expandLevel* rule twice

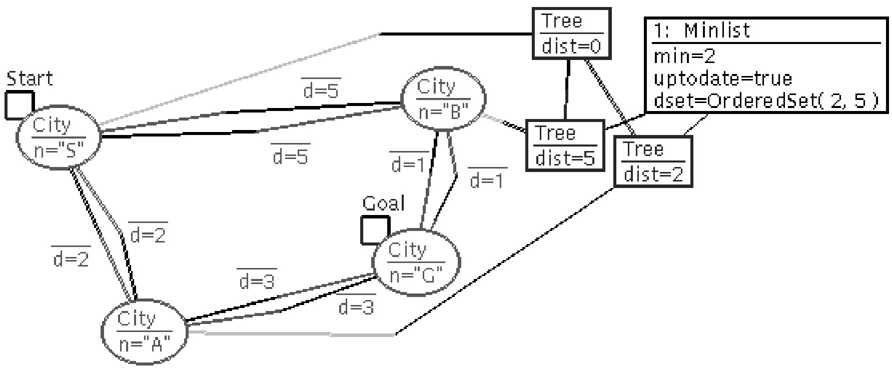


Figure 14.17: Host Graph after applying the rules *closeLevel*, *actualizeMinlist* and *calcMinimum*

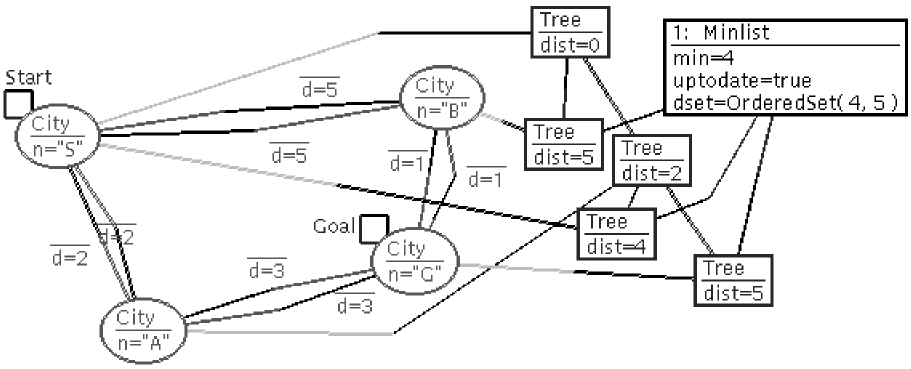


Figure 14.18: Host Graph after the computation of the second search tree level

node, connected to the start $City$ node, and the empty $Minlist$ node have been generated, whereas the $Begin$ node has been deleted from the start graph.

Figure 14.16 shows the host graph after applying the *expandLevel* rule twice. The two possible matches of the rule realize the expansion of the root node of the tree: For both neighbours of the *Start* City S (that is, for the Cities A and B) the corresponding tree nodes are generated as sons of the root node of the tree. The newly generated tree nodes carry the information of the path length from the *Start* City and are connected to their corresponding City nodes.

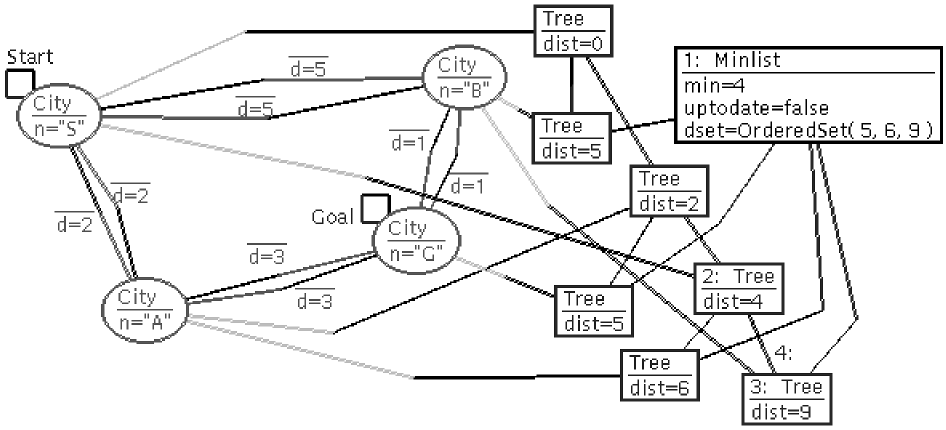


Figure 14.19: Host Graph after the computation of the third search tree level

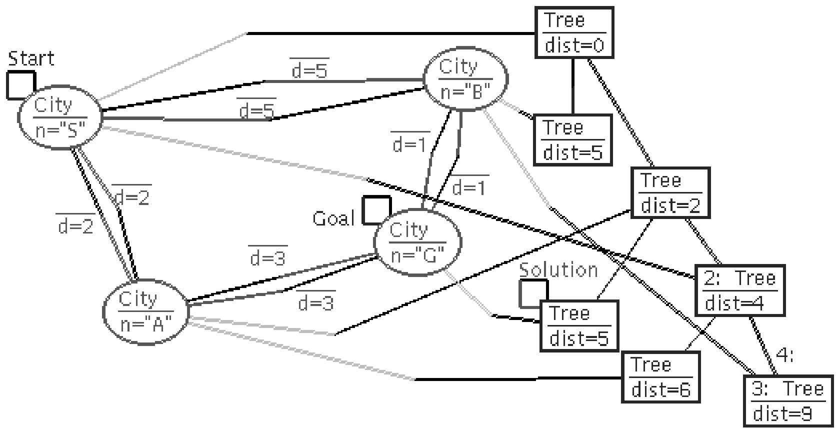


Figure 14.20: Resulting host graph after computation is finished

In Figure 14.17 we see the host graph after the application of the rules *close-Level*, *actualizeMinlist* and *calcMinimum*. The bookkeeping-node *Minlist* now contains the current path lengths list.

Figures 14.18 and 14.19 are screen shots of our host graphs after the second and the third tree levels have been expanded. The list of path lengths is updated and the minimum is calculated for each level.

Figure 14.20 shows the result of the complete execution of our graph grammar that is, it shows the host graph after no rule is applicable any more. The solution node is marked by a loop labeled *Solution*, and it contains the distance of the shortest path from the *Start* node S to the *Goal* node G in the variable *dist*.

The Interpretation Mode

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 rule list at the left side of the AGG environment main window. Starting the interpretation by clicking on the Start button in the Transform menu, each rule is applied as often as possible, until no more match for this rule can be found. Then, the next rule is tried 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. Starting again by pressing the Start button once more leads to a new application of all rules, beginning with the first rule, to the current host graph. The graph transformation finally stops if either there is no more rule applicable, or the user has pressed the Stop button in the Transform menu.

14.4.3 Implementation Issues

The first version of AGG [12] was mainly a graphical editor for unattributed hierarchical graphs with a small transformation component. It was implemented in Eiffel and presented at the CeBIT'92. The new version is implemented in Java and comprises not only graph, rule, attribute and grammar editors, but also an interpreter with debugging facilites. Figure 14.21 shows an overview of the system's structure , i.e. its main packages and their "uses" relations. The grey ellipses express a further partitioning of the system into one *core system* and an arbitrary number of *application components* built on top of it. The core system may be viewed as a library realizing the concepts of attributed graph transformation as close to the mathematical theory as possible. Most notably, the core system does not incorporate the layout aspects, i.e. the graphical representation of graphs, nor does it provide a graphical user interface for editing or transformation of graphs. Any application program making use of the functionality provided by the core system may be considered as an application component. Possible candidates for application components are components dedicated to layout or animation, graph editors, or a component to analyze graph grammars for certain properties, just to name a few. Since a convenient way to edit graphs and rules is essential, an editor component belongs to the AGG package as a standard component.

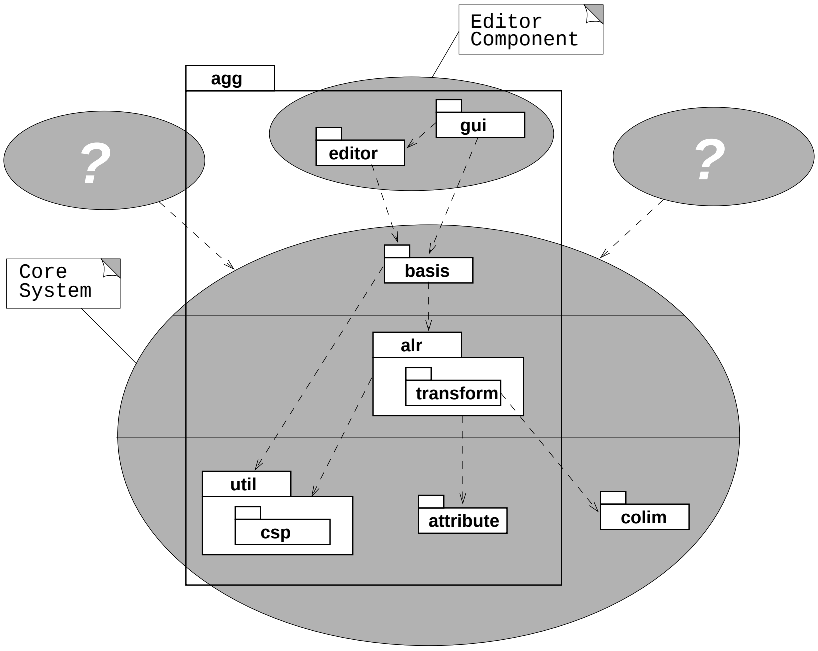


Figure 14.21: Overview of AGG system's structure.

Another interesting example for an application component is the GenGEd system which is currently in development. It is aimed at generating syntax-driven editors for any kind of diagram. It uses the AGG core system to specify and implement the operations of an editor to be generated. For more details about the GenGEd tools, compare for Chapter 3 of this volume.

As Figure 14.21 suggests, the core system consists of three layers. The packages of the bottom layer are distinguished by their potential reusability apart from the AGG context. The colimit library [21] was in fact developed independently and therefore it is not part of the root package `agg`. The middle layer implements all the data types and functions to perform attributed graph transformation, based on a hierarchical graph model called *ALR graphs*; a short description of this graph model is given later in this section. Finally, the top layer mirrors the middle layer functionality for the simple graph model as presented in Section 14.2.1. Its implementation is based on the fact that simple graphs can be considered as a special case of the ALR model.

AGG adopts the Observer pattern [6] to allow for peaceful cooperation of independently developed application components. The idea is that the core system notifies the application components about the changes of its state. This way, each application component can safely keep up with these changes, without even knowing about the mere existence of other components. For example, we could have an animation component visualizing every transformation step triggered by AGG's editor component by a movie sequence.

In the following paragraphs, we give a short summary for each of the key features of the system's implementation, supplemented by the references to more detailed information.

Hierarchical graph model

The implementation of the core functionality of the AGG system including graph matching and transformation is based on a hierarchical graph model called *ALR graphs*[13] [12]. Within this model, every node can be refined to a graph, and every arc can be refined to a bunch of arcs. Furthermore, we are able to represent arcs not only between nodes but between arbitrary graph objects. Special integrity conditions ensure that an ALR graph can be partitioned into so-called abstraction levels, each of them representing a simple graph – with the additional possibility to have arcs between arcs etc. The integrity conditions also ensure that an ALR graph has a unique top node, and that the transitive closure of the refinement objects of a node is again an ALR graph. Figure 14.22 shows an example ALR graph with three abstraction levels and a top node h, where the dashed arrows represent the abstraction function, or the refinement relation when interpreted inversely. Because every node in an ALR graph represents an ALR graph itself, we can also consider n_1 as the top node of an ALR graph containing the objects n_1, n_3, n_4 and e_2, for example. Note that the transformation and attribution concepts as presented for simple graphs in the previous sections apply analogously to the ALR graph model.

The basic idea behind the definition of ALR graphs is that we want to be able to represent graphs and morphisms within one and the same data structure. In fact, even a whole graph grammar is implemented as a single ALR graph in AGG. This opens up the way for meta-modeling, since such a representation enables us to perform "graph grammar transformation" by applying

[13]named after its inventors, R. Arlt, M. Lwe, and M. Rder

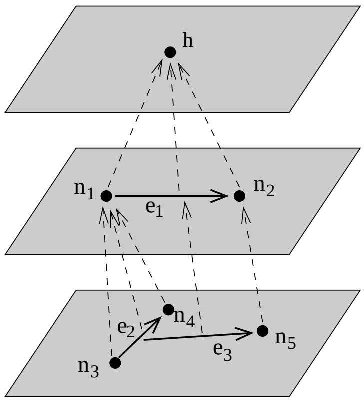


Figure 14.22: A sample ALR graph.

graph rules to the ALR graph representing the grammar. Apart from that, the refinement facilities provided by ALR graphs are of course a useful means for the vertical structuring of complex data models. For more information about ALR graphs, their use in AGG, and their relation to simple graphs (see `http://tfs.cs.tu-berlin.de/publikationen/public1997.html`).

Note that with the current version of the system, the ALR graph layer is completely hidden from the user, and the functionality of the editor component is solely based on simple graphs. An editor for the hierarchical ALR graph model may be offered in future versions.

Attribute Handling

The `attribute` package provides all the functionality needed to deal with attributes in order to perform attributed graph transformation. In particular, it offers facilities for editing and matching of attributes as well as the possibility to perform operations on attributes during a transformation step. The attribute subsystem consists of one *attribute manager* and an arbitrary number of *attribute handlers*. Each handler implements the syntactical and semantical domains of a specific type of attribute. For example, we could have one handler for boolean expressions and another one for real numbers, where each one provides its own parser defining the class of accepted expressions and an interpreter to compute the value of an expression. With a graph object having multiple attribute entries, each one may be handled by a different attribute handler. The attribute manager hides this diversity of handlers away from the client components and keeps track of which handler is responsible for which attribute. Ideally, this modular approach allows to add new handlers when the need arises without any changes to the client components. Introducing a new handler is just a matter of implementing a dedicated interface.

By default, the attribute component offers only one attribute handler, which is able to cope with arbitrary Java expressions. It turns out that this handler is so versatile that there's rarely a need for any other handler. But of course there may be some special cases where the Java handler may not be appropriate. Detailed information about the design and implementation of the attribute component and the Java expression handler can be found in [15].

Graph Matching

Finding a match for a given rule is an instance of the *subgraph homomorphism problem* which is known to be NP complete [14]. As a consequence in practice, finding a match for a given rule has a worst-case complexity which is exponential to the size of the rule's left-hand side. Since the actual transformation step can be performed in about linear time (cf. [21]), it is obvious that matching performance is decisive for the overall performance of the system. Fortunately, it is possible to reduce at least the average-case complexity of graph matching to a suitable measure by using optimized backtracking algorithms. Knowing that the overwhelming majority of theoretical as well as empirical studies on the optimization of backtracking algorithms is done in the context of constraint satisfaction problems (CSPs), AGG utilizes this knowledge by implementing graph matching as a CSP. The most important advantage of this approach is that we can apply well-elaborated CSP solution algorithms right "out of the box". Another benefit is that the actual solution algorithm becomes independent of the concrete graph model, allowing us to change the model without affecting the algorithm; we only have to adapt the translation step from the graph model into the CSP representation. More about how to represent and solve the graph matching problem as a CSP can be found under `http://www.cs.tu-berlin.de/~mich/matchcsp.ps.gz`. Details about the design and implementation of the CSP framework and the other parts of AGG's matching subsystem are presented in [16].

14.5 Future Developments

So far, we presented the current state of the AGG system to some extent. Since its realization in Java is quite recent, lots of wishes are open to obtain a sound visual language and environment, especially for the design of distributed systems. Among a variety of possible extensions the most urgent are: advanced attribute features capturing attribute conditions and rule parameters, language elements for controling rule applications, and wrt. our aim mentioned above the incorporation of distribution issues.

14.5.1 Advanced Attribution Features

The current attribute component of AGG consists of one attribute manager and an arbitrary number of attribute handlers. One attribute handler is used to treat attributes of a specific type. Currently, one attribute handler is available in AGG which interprets Java expressions, i.e. it enables the integrated execution of arbitrary Java programs with graph transformation. Java expressions are edited by a built-in attribute editor.

In a future extension of AGG, attribute handlers are to supply their own customized editors, e.g. considering an attribute handler for images, an image editor would be more appropriate than a textual one. If several attribute handlers are used within one application the usage of attributes held by different attribute handlers should be possible.

Up to now, rules cannot have parameters which are useful to determine e.g. matches and attributes of new graph objects. A future version of AGG will include input as well as output parameters for rules. Output parameters are useful to pass information between rule applications. There is often the situation that an operation cannot be described by only one rule, but by a so-called trigger rule which is applied only once. Thereafter, the application of a set of following rules applied as long as possible ensures the execution of the whole operation. Passing parameters between these rules can restrict the application area of the following rules to the state graph enormously.

Another important extension of AGG is the possibility to formulate attribute conditions which have to be checked before a rule is applied, i.e. during matching. Attribute conditions should be allowed to include any variable declared for the corresponding rule. Clearly, the evaluation of an attribute condition is dependent of variable instantiations. For our Java expression handler, the formulation of any boolean Java expression will be possible, i.e. these expression may also include method invocations.

14.5.2 Control Flow and Structuring

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 Java can serve for this purpose. Concerning imperative controling, it does not make much difference whether the language chosen is Java or another language with imperative controling. The advantage of using Java 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 transformations modelled by this graph grammar without any rule application control. This class should offer methods at least to execute rules. The test on applicability of a rule to the runtime graph is a further nice service.

After generating such a class from a graph grammar, this class can be extended or further classes using this graph transformation class can be established to implement the control flow on rule applications. New methods which are provided by these new classes can implement complex operations on graphs not describable by one rule. For example, shortest path computation (see Section 14.3) is not realizable by only one rule application. Implementing e.g. the well-known branch-and-bound algorithm (as done in Section 14.3) a number of rule applications have to be performed to compute a (restricted) decision tree step by step. Implementing this search method the basic methods of the graph transformation class generated, called *ShortestPath*, may be used.

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.

In Figure 14.23, a skeleton of class `ShortestPath` generated from the sample graph grammar given in Section 14.3 is shown. Constructor `ShortestPath()` has to create a working graph from the given start graph. The method `start(String s, String g)` applies rule *start* at a given partial match consisting of the names of the start and the goal cities. This generation step assumes that attribute parameters can be determined for rules as sketched in the previous section. All the other methods are intended to perform the application of the corresponding rule at an arbitrary match. The return value of these methods is `true` if one step could be performed. Otherwise it returns `false`. In this skeleton we omit methods to test the applicability which may be very helpful in certain algorithms.

```
public class ShortestPath {
      public ShortestPath(){ }
      private boolean found = false;
      private boolean applicable = true;

      public boolean start(String s, String g){ }
      public boolean expandLevel(){ }
      public boolean closeLevel(){ }
      public boolean actualizeMinlist(){ }
      public boolean calcMinimum(){ }
      public boolean solution(){ }

      public void computeShortestPath(String s, String g){
            while (!found){
                  start(s,g);
                        do { applicable = expandLevel();}
                  while(applicable);
                        do { applicable = closeLevel();}
                  while(applicable);
                        do { applicable = actualizeMinlist();}
                  while(applicable);
                  calcMinimum();
                  found=solution();
            }
      }
}
```

Figure 14.23: Generated Java Class Skeleton *ShortestPath* with new method `computeShortestPath`

To compute the shortest path between two cities we define a new method `computeShortestPath()` which implements the control flow presented in Figure 14.12.

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

14.5.3 Distribution Issues

Distributed graph transformation is structured graph transformation on two abstraction levels: the network and the local level. The network level contains the description of a system's network topology by a *network graph* and its dynamic reconfiguration during runtime by *network rules*. The application of such a rule to a network graph follows the double-pushout approach to graph transformation as presented in [2]. The rule-based approach is well suited to describe network reconfiguration in distributed systems, because it inherently models the asynchrony and non-determinism of reconfiguration control.

On the local level, graph transformation is used to manipulate local data structures. To describe distributed data, replication and migration of data to remote systems as well as synchronization and communication between distributed systems, network graph transformations are combined with local transformations which leads to distributed graph transformation. Distributed graphs can be nicely used to model a system's network and data structures which are distributed over different local systems. All kinds of local and distributed system interactions as well as dynamic network reconfiguration can be described by distributed graph transformation.

Distributed Graphs

A distributed graph consists of a network graph where each network node is equipped with a local graph describing a local system state. Each network edge is equipped with a total graph morphism relating local graphs to each others. These graph morphisms are used to describe replicated data structures. Each node in a distributed graph may be typed and attributed by further data types. Usually, each local system contains different sets of node, edge and attribute types. The interfaces may contain any subsets of types for communication. (Compare [19] for more details to attribution of distributed graphs.)

Network nodes being the source of a network edge may be interpreted as interfaces. See Figure 14.24 for an example of a distributed graph. In the following, we distinguish import and export interfaces. Export interface graphs offer public parts of local graphs whereas import graphs and their connection morphisms to export graphs show the imported data structures.

Distributed Graph Rules and Their Application

Distributed graph rules are useful to describe dynamic network reconfiguration by a network rule as well as dynamic local data structures in each component

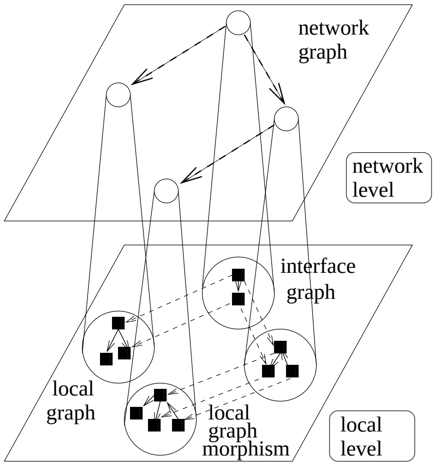


Figure 14.24: Example of a distributed graph

by a set of local rules. A distributed graph rule consists of a network rule and local rules for all network nodes which are preserved by the network rule. All newly created network nodes as well as those which should be deleted are equipped with a local graph. Local rules have to be compatible with local graph morphisms. See Figure 14.25 for a schematic representation of a distributed rule. In this example, one network node, its local graph as well as the network edge and its local morphism are deleted. The local graph determines the state in which such a network node is allowed to be deleted. One network node is preserved. Its local graph is manipulated by the given local rule. On the right-hand side a new network node has to be created and its initial state is given

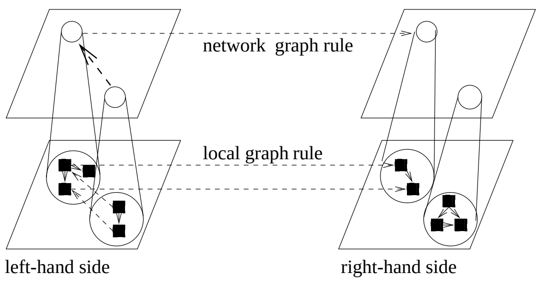


Figure 14.25: Example of a distributed rule

by its local graph.

Distributed graph rules can be used to describe all kinds of dynamics in distributed systems. These are manipulations of local data structures, synchronization and communication as well as dynamic reconfiguration of distributed systems.

To apply a distributed rule, several application conditions have to be checked to ensure that the resulting graph structure is again a distributed graph, i.e. no dangling edges occur and no local morphism becomes partial after application. Moreover, the application of distributed rules should not produce implicit side-effects on the graph structure.

A distributed graph transformation is performed by a number of simple graph transformations on both abstraction levels of a distributed graph. Besides the network transformation, a number of local graph transformations may be executed in parallel. A local transformation is performed in each network node which is preserved by the network transformation. It may contain also a number of attribute computations using variable instantiations determined by the corresponding match. Relations between transformed local graphs are induced by the transformation steps. Furthermore, network nodes and edges may be deleted and newly created as indicated by the applying rule. The context network remains unchanged.

As first extension of AGG concerning distribution we plan to support the definition of import and export interfaces for graphs and rules as substructures of those. The network level will not yet be considered explicitly, since networking reconfiguration by graph transformation will still be restricted. The main attention is directed to the synchronization of (first two, later more than two) local graph transformations related by subtransformations. This kind of distributed graph transformation describes the dynamics of some client-server systems.

The extension is designed such that several AGG processes can run concurrently on different hosts. Each AGG process has to register at an application server. Moreover, a broker will be available to each AGG process to answer queries on graph grammars and export interfaces.

14.6 Conclusions

In AGG, visually and textually programmed application parts are combined by attributing the graph objects by data values or Java objects. A second way of integrating graph transformation and Java is the use of Java to describe the application control of graph rules. This can be achieved by encapsulating a graph grammar and its transformation process in a graph transformation class. Control flow building up on graph rules can be realized by new methods in such a graph transformation class. This offers a high flexibility in using both paradigms in one application and surely increases the acceptance of graph transformation as programming paradigm.

AGG 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 next extensions planned. AGG is available at `http://tfs.cs.tu-berlin.de/agg` and on the CD-ROM in this book. There you will also find the shipping and shortest path examples shown in this article and further examples, e.g a simple browser for HTML-pages.

In Chapter 13 the first main graph transformation based language PROGRES is presented. Both languages, AGG and PROGRES, show lots of similarities, but they differ in the underlying graph model and graph transformation concepts. Whereas AGG implements the algebraic approach to graph transformation, PROGRES is based on the algorithmic approach (compare also [17]). Concerning attribution concepts, the ways how graph transformation is integrated with other programming paradigms is similar in AGG and PROGRES. Both languages and their environments are compared to some extent within Chapter 3 on the application of graph transformation to visual languages.

Several ideas remain open for future work: First of all, the type concept in AGG is still implicitly defined. A type graph containing node, arc and attribute types describes the static system aspects. It is very useful to state integrity conditions on the graphs to be considered. For the single-pushout approach, a notion of consistency conditions, logical formulas on the existence and non-existence of graph structures, and a constructive approach to ensure these conditions are presented in [9]. It is due to future work to develop a verification tool on this basis for AGG where static integrity constraints are proven to be valid for a graph grammar before run-time.

Secondly, it is worthwhile to extend the matching facilities in a way that all possible matches of a given rule set are searched for a better support of applications in the database area. Graph rules are used in that context to state queries or to formulate transformations on a database described by a graph. By gluing all (copies of) rules for matches which have been found a new amalgamated rule [5] can be constructed that describes the total effect/query constructed from elementary ones.

The proper aim we are heading for AGG is a visual design tool for distributed systems. The idea is to let AGG run distributively on several hosts. The concepts are based on distributed graph transformation as presented in [18] and [19]. Graphs and rules are distributed over graph grammars and may share common parts which are useful for communication. Several rules may be applied synchronously on their graphs sharing certain subrules which describe common effects. AGG should help developing distributed applications by offering a visual high-level language for this purpose.

Acknowledgements The AGG team consists of Boris Melamed, Olga Runge and the authors. The development of AGG is discussed and performed within this team. Hartmut Ehrig and Albert Zündorf gave valuable hints to round off this article.

References

1. 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.
2. 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.
3. P.T. Cox, F.R. Giles, and T. Pietrzykowski. Prograph. In M. Burnett, A. Goldberg, and T. Lewis, editors, *Visual Object-Oriented Programming*, pages 45 – 66. Manning Publications, 1995.
4. H. Ehrig and R. Bardohl. Specification Techniques using Dynamic Abstract Data Types and Application to Shipping Software. In *Proc. of the International Workshop on Advanced Software Technology*, pages 70–85, 1994.
5. 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, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1996.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

7. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.
8. R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Attributed graph transformations with controlled application of rules. In G. Valiente and F. Rossello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science.* Technical Report B - 19, Universitat de les Illes Balears, 1995.
9. 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.
10. K. Kahn and V.A. Sarswat. Complete visualization of concurrent programs and their execution. In *Proceedings of "IEEE Workshop on visual Languages, Skoje, IL"*, pages 7 – 15, 1990.
11. M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.
12. M. Löwe and M. Beyer. AGG — an implementation of algebraic graph rewriting. In *Proc. Fifth Int. Conf. Rewriting Techniques and Applications, '93, LNCS 690*, pages 451–456. Springer Verlag, 1993.
13. 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.
14. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness.* Springer Verlag, 1984.
15. B. Melamed. Grundkonzeption und -implementierung einer Attributkomponente für ein Graphtransformationssystem. Studienarbeit, Technical University of Berlin, Dep. of Comp. Sci., 1997.
16. Michael Rudolf. Konzeption und implementierung eines interpreters für attributierte graphtransformation. Master's thesis, Technical University of Berlin, Dep. of Comp. Sci., 1997. `http://tfs.cs.tu-berlin.de/publikationen/public1997.html`.
17. A. Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations.* World Scientific, 1996.
18. G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin, 1996. Shaker Verlag.

19. G. Taentzer, I. Fischer, M Koch, and V. Volle. Visual design of distributed systems by graph transformation. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1998. to appear.
20. P.H. Winston. *Artifical Intelligence*. Addison Wesley, New York, 1984.
21. Dietmar Wolz. *Colimit Library for Graph Transformations and Algebraic Development Techniques*. PhD thesis, Technical University of Berlin, Dep. of Comp. Sci., 1998.