Introduction to the Language Concepts of AGG

Michael Rudolf, Gabriele Taentzer
TU Berlin

November 10, 1999

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. The attribution of nodes and arcs by Java objects and expressions follows the ideas of attributed graph grammars 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 AGG graphs and how they are attributed are presented in Sections 1 and 2. Graph rules and their application to graphs are discussed in Section 3. Section 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.

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.

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 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.

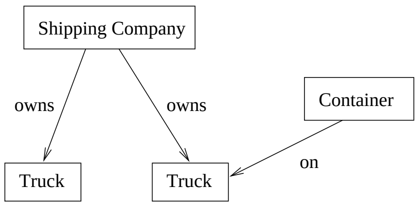

Figure 1: A shipping company modeled as a simple graph.

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 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.

2 Attributes

Let us reconsider the graph model of the shipping company in Figure 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 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 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.

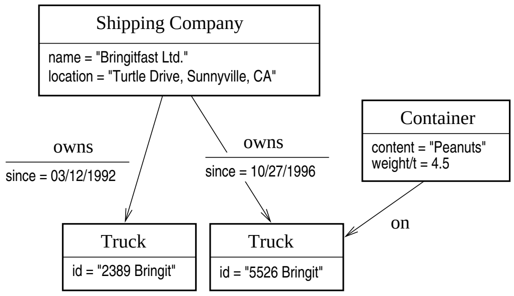

Figure 2: The example from Figure 1 augmented with attributes.

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 [1] in the paradigm of object-oriented

[1] sometimes also called *class attributes*

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 [2] to manipulate object attributes as well as to interact with the user or with the underlying system environment, while still specifying the structural aspects of a transition in a graphical way. Simple, but non-trivial sample attributions based on user-defined Java classes are shown in our examples.

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.

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

[2] In the object oriented paradigm, procedures or functions associated with a class are called *methods* of this class.

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 3. Once having figured out the "before" state, specifying 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. 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 To indicate which objects are mapped

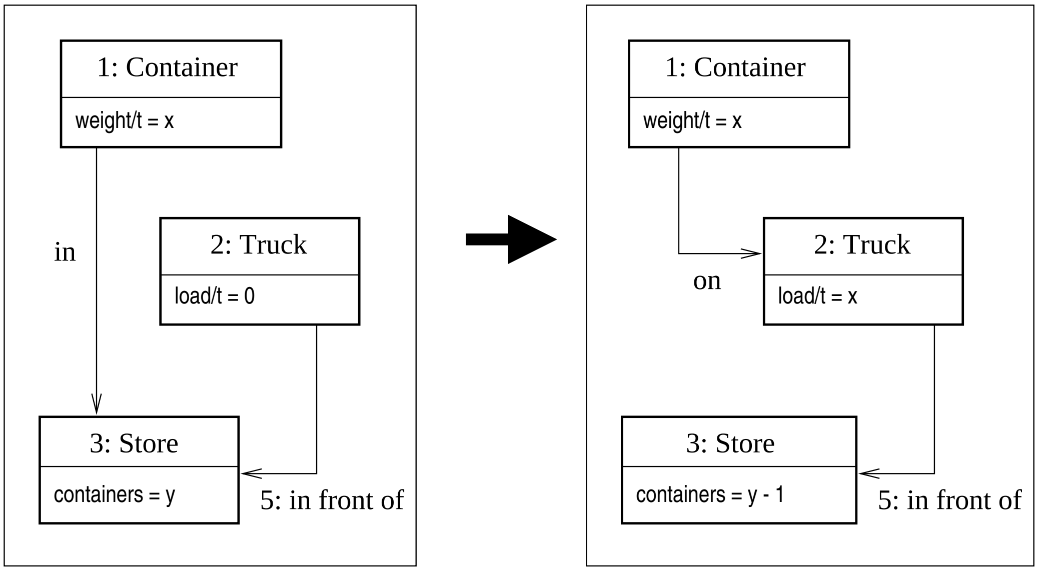

Figure 3: Graph rule to load a container onto a truck.

to one another in the figures, we use numerical tags preceding an object's type name, separated by a colon. Figure 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 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 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 rule for selling a truck described in a way that the "truck"-node is deleted. Applying this rule to node (6) in Figure 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 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

	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 1: Restrictions to attribute expressions in different graph contexts.

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. Please have a closer look at the sample method invocations in our examples.

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 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 nodes at the single resulting node. Consider the example in Figure 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.[3] In additon, Figure 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 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 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.[4]

---

[3]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.

[4]With the well-defined exception of the implicit removal of dangling arcs and conflicting

local : Only the fraction of the state graph covered by the match is actually affected by the transformation.[5]

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 yield the same results.

To ensure compliance with the formal semantics definition, AGG's transformation engine is based on a library designed to perform arbitrary colimit computations 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. 4, as AGG performs transformation steps *in-place*, i.e. the rules modify the state graph directly.

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 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 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

---

objects.

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

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 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 instance, in Figure 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 $n \circ l = m|_{dom(l)}$).

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 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, 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 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.[6] Under 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.[7] 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 8 shows diagrammatically how a conjunction of multiple constraints is constructed.

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

[6] 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.

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

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 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 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".

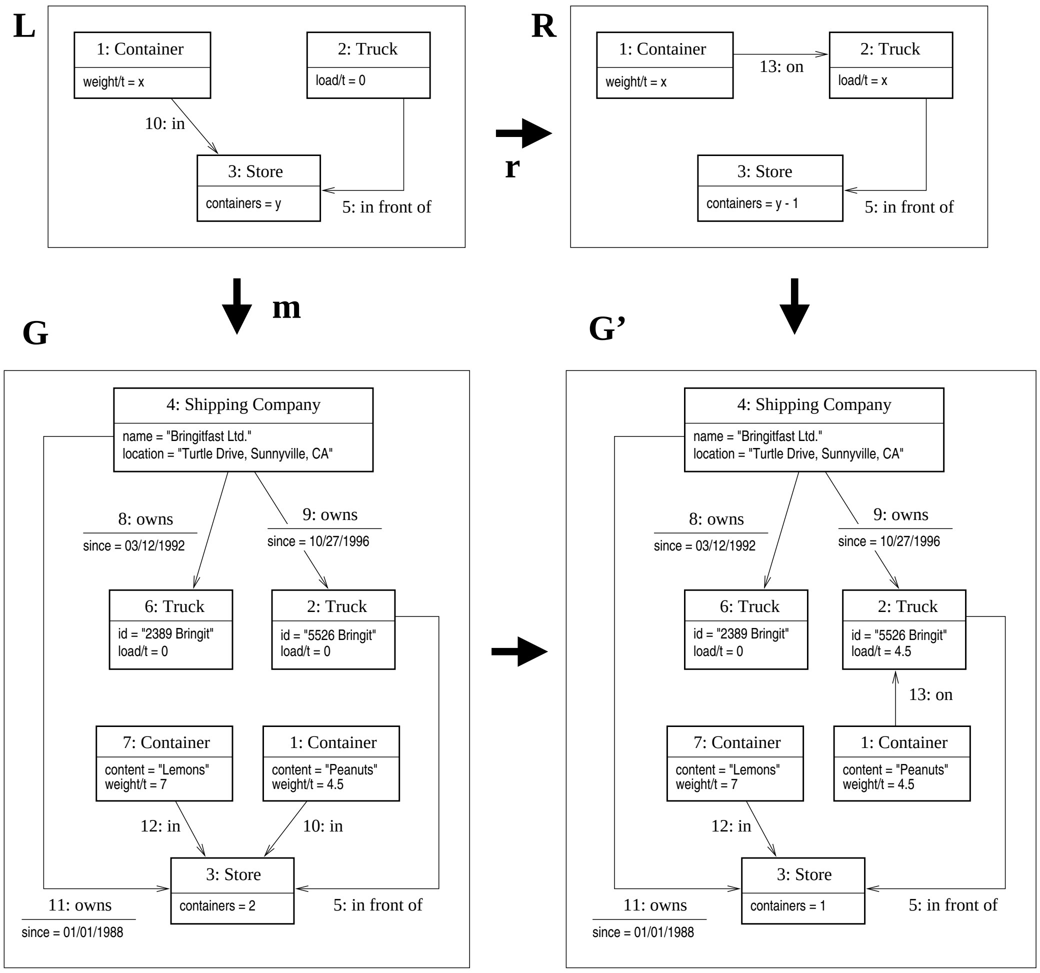

Figure 4: A transformation step by applying the rule from Fig. 3 to a state graph G at a match m.

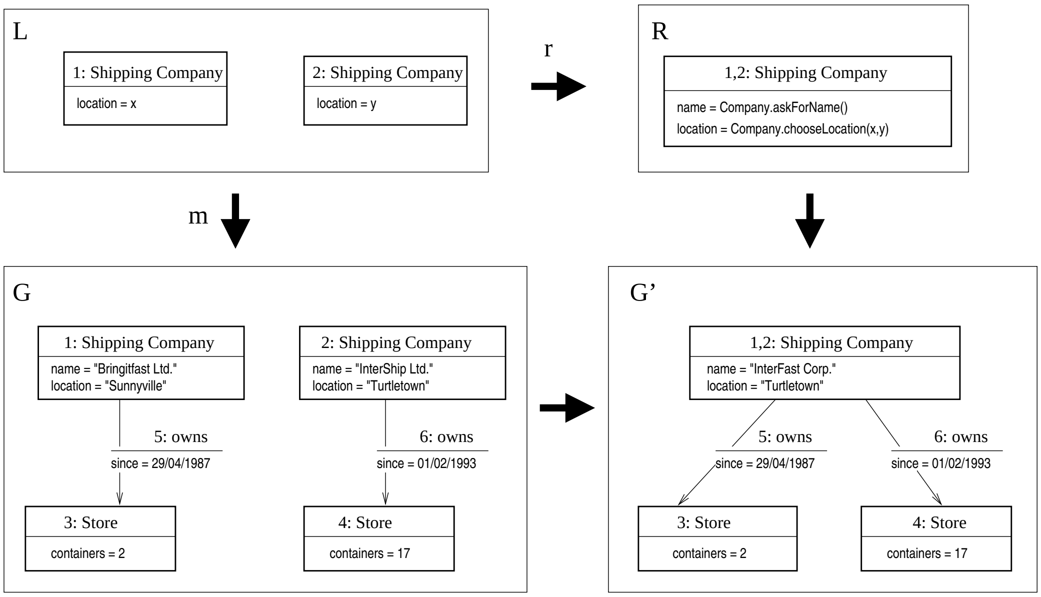

Figure 5: Graph transformation showing the fusion of two shipping companies.

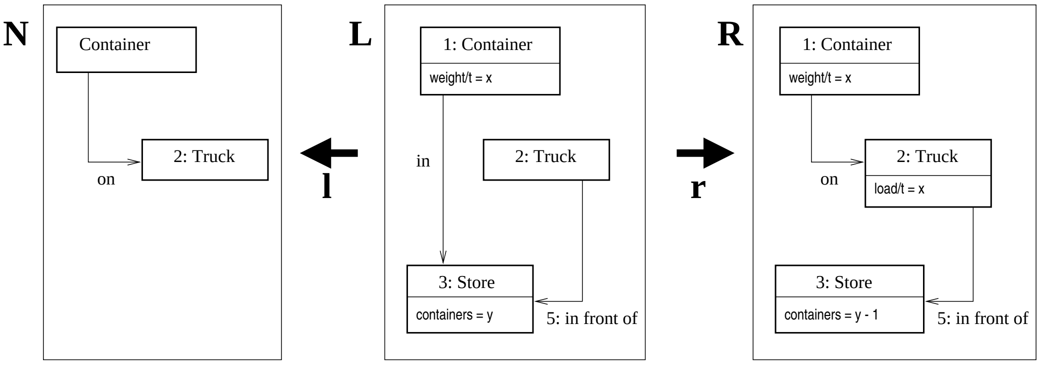

Figure 6: A negative application condition preventing a truck to be loaded if it is already occupied.

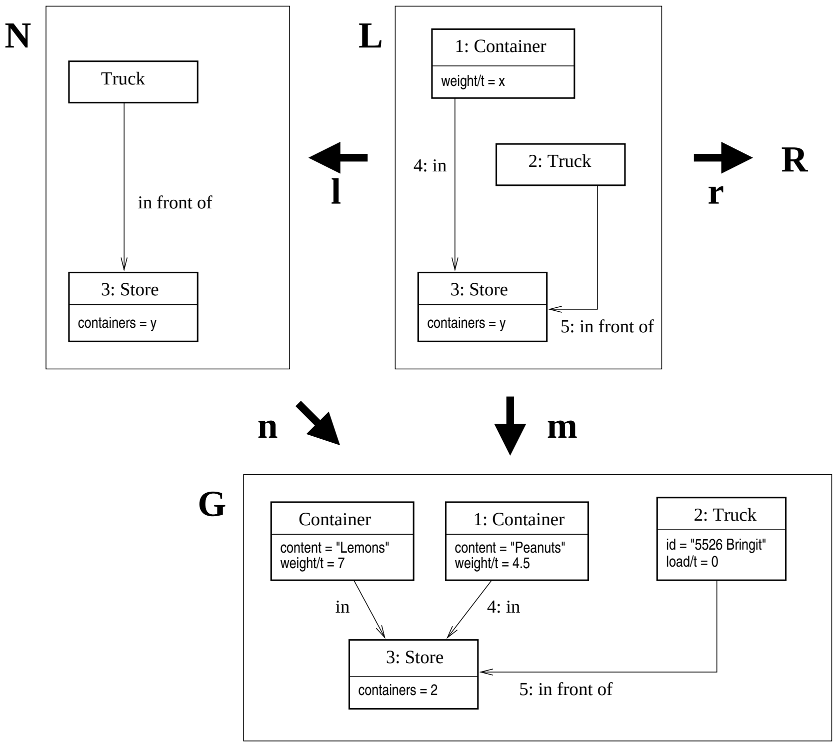

Figure 7: An inconsistent negative application condition causing the rule never to be applicable.

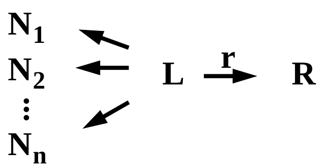

Figure 8: Scheme of a rule with multiple NACs.

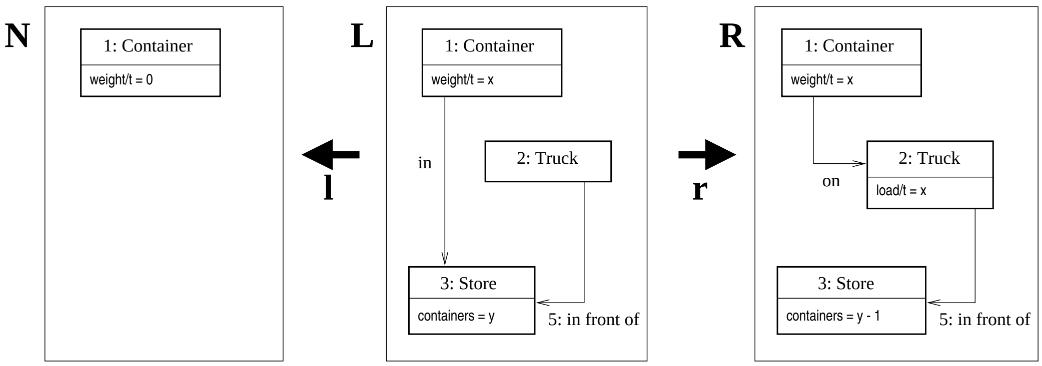

Figure 9: Using a NAC to avoid particular variable instantiations.