Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 152 (2006) 207–222
www.elsevier.com/locate/entcs

Improving the Usability of a Graph Transformation Language

Attila Vizhanyo[1], Sandeep Neema, Feng Shi,
Daniel Balasubramanian, Gabor Karsai

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Model transformation tools implemented using graph transformation techniques are often expected to provide high performance. For this reason, in the Graph Rewriting and Transformation (GReAT) language we have supported two techniques: pre-binding of selected pattern variables and explicit sequencing of transformation steps to improve the performance of the transformation engine. When applied to practical situations, we recognized three shortcomings in our approach: (1) no support for the convenient reuse of results of one rewriting step in another, distant step, (2) lack of a sorting capability for ordering the results of the pattern matching, and (3) absence of support for the distinguished merging of results of multiple pattern matches. In this paper we briefly highlight the relevant features of GReAT, describe three motivating examples that illustrate the problems, introduce our solutions: new extensions to the language, and compare the approaches to other languages.

Keywords: Model Transformations, Graph Transformations, Usability

1 Introduction

Model transformation tools are expected to work reliably and provide high performance. When model transformation is used for tool integration (as in [1]), or when it is used in an interactive tool to assist a modeler, we expect that the transformation is rapidly executed. On the other hand, the developers (the "programmers") of the model transformation also expect high perfor-

[1] viza{,sandeep,fengshi,daniel,gabor}@isis.vanderbilt.edu

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2005.10.026

mance from the language: they want to express complex transformations in a concise, yet understandable form, without having to deal with low-level implementation details, and with relying on the constructs of the transformation language.

In the first generation of the GReAT (Graph Rewriting and Transformations) language [2], we have provided a number of capabilities to address the first concern, at the expense of the second. The performance improvements were made using the following techniques: 1. To cut down on the search in order to find matching subgraphs in the host graph, we support the pre-binding of selected graph pattern elements to well-known nodes in the host graph. These pre-bound elements change the search into a local graph search problem. Furthermore, the pre-bound elements and new elements created by the rewriting rule can also be reused in a subsequent rewriting rule. 2. To eliminate the search for rule activations at execution time, we explicitly sequence the rewriting rules. The sequencing is specified by passing the bound host graph elements discussed above from rule to rule. Special rules, called "tests" are used to provide conditional execution paths. 3. To improve performance further, we generate executable code in a procedural language (C++) that is compiled with a standard compiler [3]. Techniques 1 and 2 will be described in the following section, which details the semantics of the GReAT language.

While these techniques provided us with a language in which we could implement transformations executing with acceptable performance, there were a number of shortcomings that we discovered in practice. This paper will introduce three motivating examples that illustrate what the problems were, along with the improvements we made to the language to address those problems and the implementation of the languages improvements. This is followed by a short comparison with related work, and the summary and conclusions.

2 GReAT

GReAT is suitable for the formal specification of model transformations, where UML class diagrams are used to represent the abstract syntax of the input and the output models of the transformation. The abstract syntax of a model defines its modeling concepts, their relationships, and their integrity constraints [8]. We regard the UML classes as the nodes of a type-graph [9], and the various associations between these classes as the edges of the graph. In this paper only the necessary language constructs are explained, [4] describes the full approach and support tools. The operational semantics of GReAT is formally defined in [5].

The applicability of graph transformation techniques to model transfor-

mation problems can be understood by regarding the models as vertex and edge labeled multi-graphs, where the labels are referring to the corresponding types in the type-graph, that is the UML class diagram. If we represent the models with graphs, then the graphs can be considered as object networks, whose "schema" is represented using UML class diagrams. UML class diagrams can play the role of a graph grammar in that they can describe all the "legal" object networks that can be constructed with the domain. Therefore, it is plausible to formulate the model transformation problem as a graph transformation problem.

Model transformations in GReAT are represented as explicitly sequenced elementary rewriting operations, called productions or rules. A production contains a pattern graph that consists of pattern vertices and edges. GReAT uses the UML class diagram notation to specify the pattern vertices and edges. Each pattern element in a GReAT rule can play one of three roles: Bind, Delete, or CreateNew. The role specifies how an element is used during the transformation. The execution of a rule involves matching every pattern object marked as either Bind or Delete. The pattern matcher will return the set of all possible matches for the given pattern that are found in the host graph. The matches are then evaluated with respect to the optional guard condition, and matches for which the guard evaluates to false are discarded. Then, for each match which satisfies the guard condition, graph objects marked as Delete are deleted from the graph, and objects marked as New are created. Finally, the attributes of the graph objects can be manipulated by an optional AttributeMapping (AM) specification. The attribute mapping uses programmatic access to the graph objects through a well-defined C++ API.

Consider the simple rule shown in Figure 1. "Container" has the Bind role, "LP" and the "Container/LP" composition have the Delete role, and "Inport" and the "Container/Inport" composition have the CreateNew role. The semantics of this rule is: find all "CompoundComponent"-s, along with all of the "LocalPort"-s contained inside each "CompoundComponent". Next, evaluate the guard condition. Let this condition be, "Container.name == LP.name". Thus, only matches in which the name of the "LocalPort" is the same as the name of the Container will pass the guard, and the rest will be discarded. Finally, the "LocalPort" objects of the valid matches will be deleted, and a new "InputPort" will be created in the "CompoundContainer". As stated in the introduction, GReAT supports the pre-binding of selected graph pattern elements to well-known nodes in the host graph. This provides a rule with an initial partial match of its pattern, which significantly reduces the search space. The initial matches are provided to a transformation rule with the help of input ports that form the input interface of a transformation

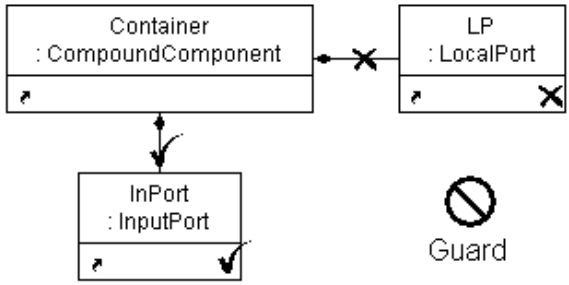

Fig. 1. A Simple Rule in GReAT

step. Similarly, output ports can be used to pass matched or newly created objects to a subsequent rules. For a transformation rule to be applied to the host graph, graph objects must be supplied to each input port of the rule; if there are no objects supplied to a rule's input ports, the rule is not executed.

While specifying a transformation involving multiple domains (i.e. multiple, unrelated type-graphs), it is often necessary to associate objects of different domains with each other. GReAT supports the specification of associations between UML classes belonging to different domains by allowing the user to specify crosslink diagrams. Such associations cross-cutting individual domains (referred to as "crosslinks" in GReAT) can then be created and manipulated during the execution of the transformation, just like ordinary associations.

To illustrate the use of ports and crosslinks, consider the rule shown in Figure 2. (Here, we are assuming that the UML classes "LocalPort" and "Transmitter" belong to different domains.) In1 is used to provide an initial binding for "LocalPort", and In2 is used to provide an initial binding for "Actor". In other words, when the rule fires, selected "LocalPort" and "Actor" graph objects are readily available for pattern matching. The rule then finds all "Transmitter"-s contained in the "Actor", and creates crosslinks between all matched "Transmitter" and the "LocalPort" objects, which is denoted by a tick mark right next to the association in the figure. Finally, matched "Transmitter" objects are passed along to the subsequent rule via the output port Out1. The dataflow in GReAT also implies an execution order for the rules as follows. If the output ports of RuleA are connected to the input ports of RuleB, then RuleA must execute before RuleB. Each rule can have an arbitrary number of ports, and each port is supplied with graph objects from (possibly various) sources. If one source rule supplies objects for an input port of a destination rule, it must supply objects for all input ports of the destination.

In order to manage the complexity of transformations, GReAT provides

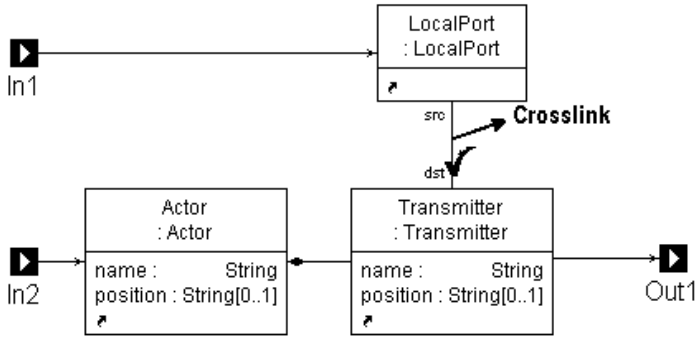

Fig. 2. Input Ports and Crosslinks

the user with higher-level constructs, such as hierarchical rules and control structures. High-level rules can be created through the composition of a sequence of primitive rules. There are two kinds of high-level rules in GReAT: Blocks and ForBlocks, both of which contain primitive rules. The difference between the two is that a Block passes all inputs to the first contained rule, the output created by this first rule are passed to the second rule, and so on. A ForBlock passes one input all the way through its contained rules, then passes the next packet, and so on.

3 Global container

As introduced above, the control flow language of GReAT specifies an execution order of the elementary transformation steps. A transformation step always starts the pattern matching with an initial context. By context we mean the pre-binding of some pattern variables to host graph vertices. This context (i.e. the initial binding) is passed along from rule to rule via ports during the transformation, similar to parameter passing in procedural languages.

The main weakness of this approach is that the programmer needs to specify the context passing through several rules, even if the context is actually used only in one remote, non-adjacent, step. To simplify development, we have introduced the concept of the global container. The input(s) and the output(s) of the transformations reside in containers that hold objects and links of the input and output graphs. These containers are selected at the beginning of the transformation, and each production matches/deletes/creates objects within these implicit containers.

We defined the *global container* as an object network whose "root" object (that contains all objects within the container) is accessible throughout the entire transformation, and thus it is not necessary to pass the root object along as the context. The global containers consist of temporary, non-persistent

objects that exist only during the execution of the transformation. It is the programmer's task to define the "syntax" (i.e. the type-graph) of the container by drawing a UML class diagram, much like when defining the input and output domain(s) of the transformation. The programmer can specify the syntax of an arbitrary number of global containers, and GReAT will manage the instances (i.e. the containers) of these. One can create an arbitrary type system for a global container, including defining new classes with attributes, associations, etc. with all the capabilities of a standard UML class diagram. The only globally accessible object per container is a singleton instance of the root type, called *global root object* or *global object*. From this single global object, other model objects can be reached via pattern matching, whose type in turn can be either a new type or be part of any type-graph defined for the transformation (including that of the input and output domains).

Global containers are most useful in large transformations, when they can eliminate a large portion of context passing, or recurring complex pattern matching. One example is generating code from models, where components are used to model the functional decomposition of a system. Suppose that the transformation consists of several rules, and the leftmost and rightmost rules are as shown in Figure 3. Furthermore, suppose that the model contains hundreds of components, and some of them were incorrectly named for code generation (e.g. their name begins with a number). We need to perform multiple operations on the set of these incorrectly named components in different transformation steps. The number of such components is negligible compared to the total number of components, so it makes sense to reuse the pattern matching results to cut down on the search time. However, it is also inconvenient to specify context passing all over dozens of rules, especially when the context is used only in the last rule. To save the programmer from doing the excess work, the global container feature can be used.

The left rule in Figure 3, (1) associates components that have invalid names with a global object "NSRootObject", (2) counts them using the counter attribute of "NSRoot". The right rule, which can be far away, in a distant part of the transformation program, renames these selected components to conform to the target language naming conventions. Notice that the pattern matching finds the components through the "NSRootObject-Component" association starting from "NSRootObject", as there is no pattern containment relationship specified starting from "RootFolder".

The disadvantage of the global container feature is that it still uses pattern matching to find previously found results, although the search time is considerably lower (as in our example). In comparison, context passing does not need pattern matching, but it requires memory space to store the results.

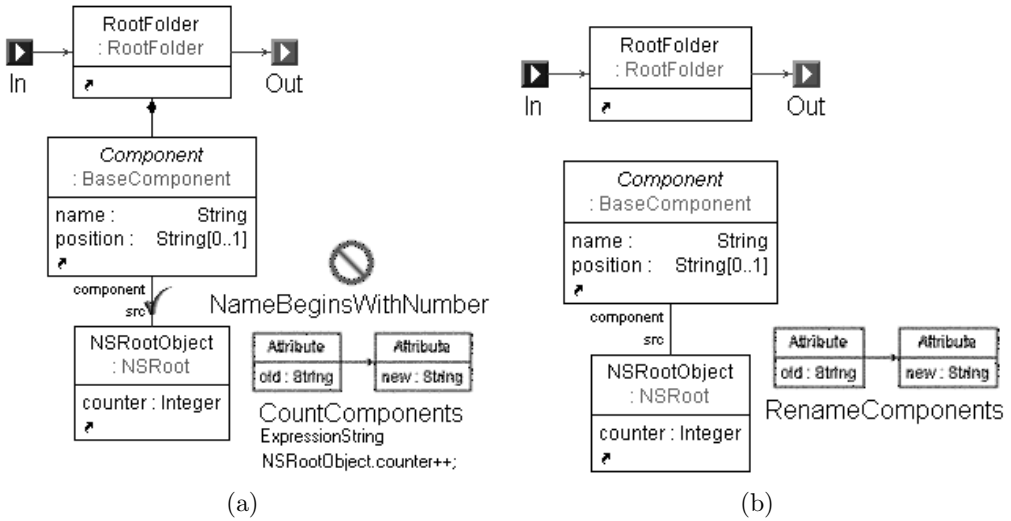

Fig. 3. Using global containers

The primary advantage of using global containers is reduced development time through supporting a convenient reuse of results of one rewriting step in remote step. Global containers also help to remove superfluous details from the transformation for the sake of a cleaner program structure.

4 Sorting

GReAT is built upon a graph pattern specification language and pattern matching. The pattern matching is deterministic in the sense that it returns the set of all the valid matches for a given pattern, and that set of matches will always be the same for a given pattern and host graph. Pattern matching is non-deterministic in the sense that the order of the elements (matches) in the set may vary between different executions. This kind of non-determinism is not acceptable in some model transformations, where certain elements need to be processed in a fixed order. Recall that the rule performs the actions on the set of matches in the same order as the pattern matching found them.

One example when deterministic order of matches is compulsory is interpreting hierarchical concurrent state machines, e.g. Matlab's StateFlow [6]. The operational semantics of StateFlow prescribes that parallel (AND) states are evaluated and executed from left to right and top to bottom. In other words, every concurrent (AND-decomposed) compound state has multiple active sub states, and is responsible for executing all its children in this specific order during performing a state machine step.

In one of our previous works, we have implemented a StateFlow to C code

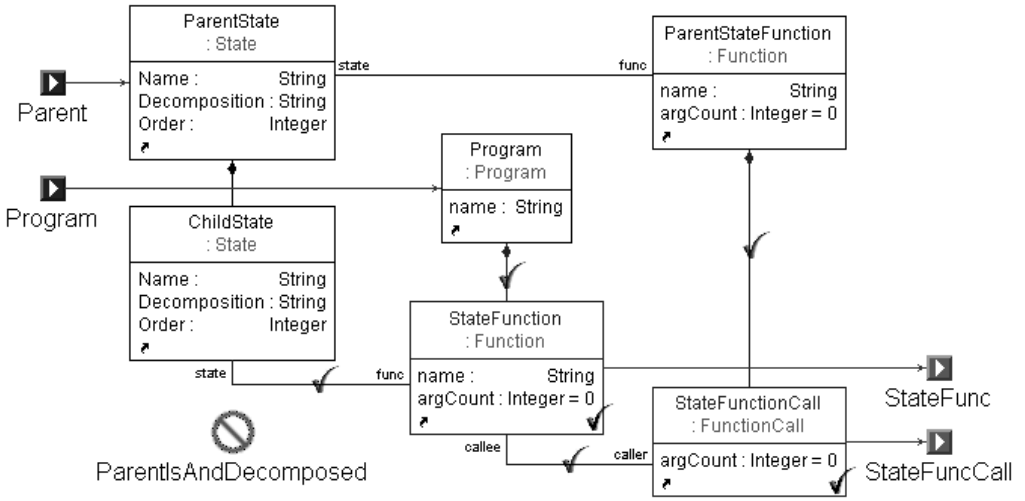

Fig. 4. Create State execution functions

generator in GReAT [7]. The code generator essentially produces a C function definition for each StateFlow state. For concurrent compound states, the generated C function is responsible for calling the functions representing the child states. For example, suppose that the concurrent state "Parent" contains three parallel states from left to right and top to bottom: "ChildA", "ChildB" and "ChildC". Then the generated state execution function is expected to look like:

```
void ParentExec() { ChildA(); ChildB(); ChildC(); }
```

Note that the first "ChildA" is executed, then "ChildB" and last "ChildC", and any other state evolution order does not conform to the StateFlow operational semantics. In our implementation, we created a metamodel that captures the abstract syntax of a stylized subset of C. Then, during the transformation steps we create model elements representing C code segments. The resulting model is then printed out into C text format in a post-processing step. Figure 4 shows our original solution for concurrent state code generation. The specified pattern finds all children of "ParentState", and the associated state function definition for "ParentState" (through a cross link generated earlier in the transformation). Then those matches with sequential (non-concurrent) compound states are discarded by evaluating the guard condition. Finally, state execution function definitions and function callers referring to the functions definitions are created in the last step. The problem with this pattern specification is that the function callers are created in a random order, so nothing will enforce the correct execution order of the parallel states. In the case of the previous example, if the pattern matcher returns with the set: {*(Parent, ChildB, Prg, ParFunc), (Parent, ChildA, Prg, ParFunc), (Parent,*

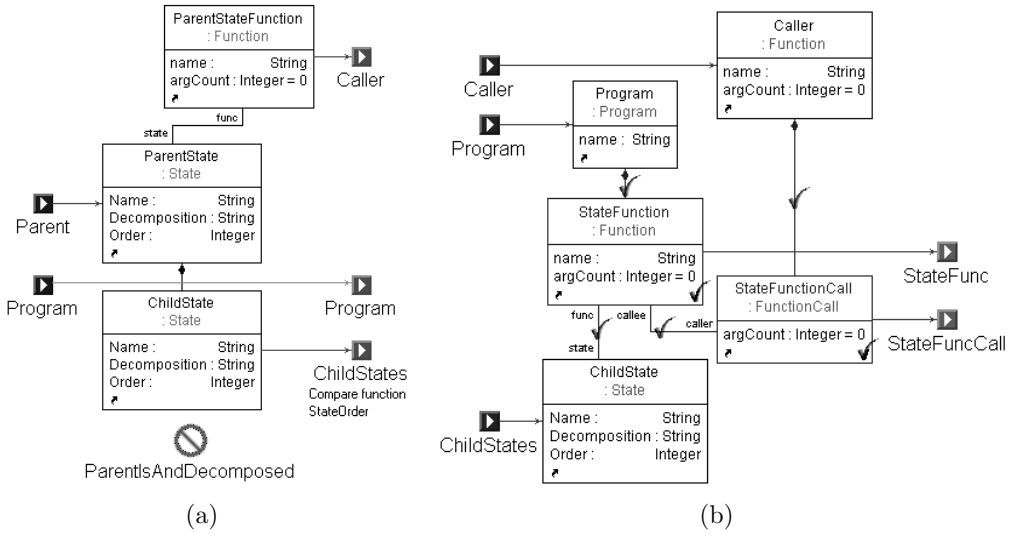

Fig. 5. Create ordered State execution functions

ChildC, Prg, ParFunc)}, then the generated code will look like:

```
void ParentExec() {ChildB(); ChildA(); ChildC();}
```

Clearly, we need to specify an ordering of parallel states, based on the "Order" attribute of the "ChildState"-s. In other words, we want to reorder the set of matches using some ordering criteria, e.g. sort the matches based on the "Order" attribute of "ChildState"-s. This sorting step should take place after pattern matching, or even after the effector, so that newly created objects can also be reordered. This implies that the sorted matches can be used only in subsequent transformation steps. The sorting step is the last operation executed before leaving the rule.

The GReAT programmer can specify sorting by setting the attribute "Compare function" of an output port in a rule. In Figure 5, the compare function "StateOrder" is used to order the parallel states. The compare function is coded in a procedural language (here: C++), and its two arguments are the two objects to compare. An example compare function is shown in Figure 6. The programmer needs to specify only the function name and the body, the function signature is automatically generated. If sorting is required, the matches produced by the pattern matching are reordered and sorted such that ordering relation specified by the compare function holds between the elements. In our example, {*(Parent, ChildB, Prg, ParFunc), (Parent, ChildA, Prg, ParFunc), (Parent, ChildC, Prg, ParFunc)*} is reordered yielding {*(Parent, ChildA, Prg, ParFunc), (Parent, ChildB, Prg, ParFunc), (Parent, ChildC, Prg, ParFunc)*}. These matches are then passed along in the

```
template <class T>
bool StateOrder (const T& lhs, const T& rhs) {
return lhs.Order() < rhs.Order();
}
```

Fig. 6. Predicate for comparing States by using the Order attribute

same order to the next rule as shown in Figure 5. Finally, "StateFunction"-s and "StateFunctionCall"-s are created for each match in the correct order, ensuring that the generated C function will execute the parallel states in the correct order.

As a side note, there was another implementation initiative to integrate sorting into the pattern matching, such that the pattern matching should find the matches in the specified order. This proposal would have permitted the use of sorted matches inside the same rule. However, we have dropped the idea, because this change would have required the pattern matching to traverse the pattern graph in a specific direction. This requirement involves a preprocessing step of analyzing the pattern graph and computing a traversal path, which in turn introduces a substantial computational overhead in the pattern matching. Clearly there is also a runtime overhead associated with reordering the matches, but (1) this overhead is typically smaller than computing the traversal path (2), reordering needs to be performed only when sorting is specified vs. the traversal path would be required to be computed in all cases for each pattern graph.

5 Distinguished merging

The fundamental approach to managing the inherent complexity of a program is to separate the program into distinct subsystems (a.k.a. module, library or component). The separated parts are connected only by a well-defined interface. In graphical languages, subsystems typically expose their interfaces with the help of ports. For example, in signal flow languages, data ports indicate the types of signals the component accepts. The programmer specifies the signal flow by connecting the output ports of the source component to the input ports of the destination component. As the interface gets larger, the number of connected ports increases rapidly. Connecting each pair of source-destination ports manually becomes cumbersome and tedious, thus the process necessitates automation.

Notice that this problem can be regarded as a model transformation problem: new signal flow connections need to be added between selected ports specified by some criteria. The solution is not trivial using graph transformations, however. The pattern matching algorithm must find and return pairs of

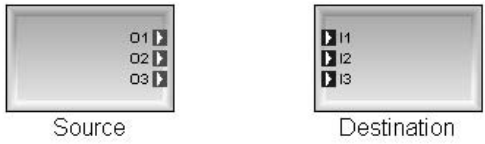

Fig. 7. The Port Connection Problem

ports that need to be connected. The search space is the cross product of all output ports of the source by all input ports of the destination. The challenge lies in how to match those distinguished pairs of porst to be connected. For example in Figure 7, the total search space is the all possible combinations of ports, that is {*(O1,I1), (O1,I2), (O1,I3), (O2,I1), (O2,I2), (O2,I3), (O3,I1), (O3,I2), (O3,I3)*}. What we need is some pattern and/or guard condition that discards the unwanted combinations, and keep only those matches that represent the ports to be connected. A general-purpose pattern matcher cannot figure out automatically which are the ports to be connected, so some restricting criteria are needed to specify the acceptable combinations. A simple criterion can be given by using the physical layout ordering of the ports: the topmost output port should be connected with the top most input port, the second output port from the top should be connected with the second input port, and so on. Using this criteria the search space is restricted to {*(O1,I1), (O2,I2), (O3,I3)*}, and this set indeed represents the desired port connections.

While this criterion is clear and simple, it is very difficult to specify it in stateless graph transformation languages. The problem is that the criterion involves comparing matches with each other, and thus introduces memory requirements for the pattern matching. The pattern matching must remember if it has already found a topmost port, and if so, discard the current match (e.g. I2, I3 must be discarded for O1, because I1 has already been found). Even if we introduce memory into the pattern matching algorithm, the language also needs to be augmented with new elements that let the user refer to and discard matches previously found by the pattern matching. We found that these new language features would be extremely complex compared to the existing language semantics, so we tried to find another solution which would have a natural syntax and an intuitive semantics.

As described above, what we really need is to find a distinguished subset of matches in the original search space. The two most prominent properties of this subset are: (1) each port occurs only once in the set (2) the ports are sorted: they are paired up to form elements in a specific order. Next we will show that these two properties together are sufficient to select the distinguished subset. The selection algorithm, called distinguished merging, is

Input: {*(O1,I1), (O1,I2), (O1,I3), (O2,I1), (O2,I2), (O2,I3), (O3,I1), (O3,I2), (O3,I3)*}
After step 1: *X*={*O1,O1,O1,O2,O2,O2,O3,O3,O3*},
Y={*I1,I2,I3,I1,I2,I3,I1,I2,I3*}
After step 2: *X*={*O1,O1,O1,O2,O2,O2,O3,O3,O3*},
Y={*I1,I1,I1,I2,I2,I2,I3,I3,I3*}
After step 3: *X*= {*O1,O2,O3*}, *Y*={*I1,I2,I3*}
After step 4 & output: *D*={*(O1,I1), (O2,I2), (O3,I3)*}

Fig. 8. Connecting Ports

outlined below:

Name: Distinguished merging
Inputs: **P(I x O)** : Cross product of input and output ports, or any subset of **P**; **s**: sorting criteria
Output: **D(I x O)**: Distinguished subset of the cross product, where the elements represent the wanted connections
Algorithm:
Step 1: Break apart the elements of the cross product and create two corresponding bags for input and output ports.

```
X = first(P); // first elements of the pairs in P
Y = second(P); // second elements of the pairs in P
```

Step 2: Sort the elements of the input port bag and output port bag independently using the sorting criteria.

```
X = sort(X,s); Y =sort(Y,s);
```

Step 3: Remove duplicates of consecutive elements with the same value.

```
X = remove_duplicates(X); Y = remove_duplicates(Y);
```

Step 4: Compute the distinguished subset, by forming pairs from the elements of the ordered sets.

```
D = form_pairs(X,Y);
```

Figure 8 demonstrates the algorithm with the ports shown in Figure 7, and sorting ports based on their vertical position.

Distinguished merging can easily be extended to multiple sets. Suppose we want to execute the algorithm on N sets. Then we have an input set containing N-ary tuples as elements, which is broken down into N individual bags during step 1. After sorting and creating 'unique' bags, we form N-ary tuples by taking the elements of each set vertically.

Note that the algorithm fails if the number of elements of the individual sets differ after step 3. Indeed, step 4 needs sets of the same size to complete

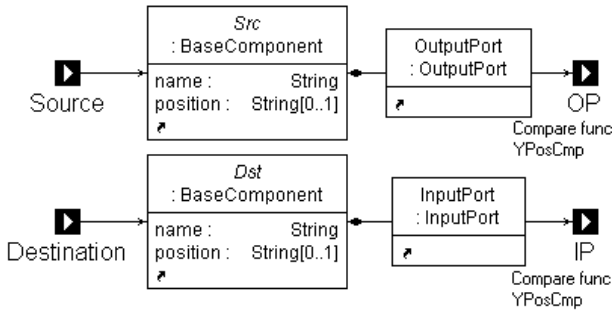

Fig. 9. Rule that selects connecting ports

forming tuples, otherwise it will produce incomplete tuples. In GReAT, it is the user's responsibility to ensure that each set will contain the same number of elements after step 3. For the port-connection example this means that the source must have as many output ports as the destination has inputs.

Also note that the algorithm does not necessarily select a subset in the input, but rather computes new elements out of the constituent parts, i.e. the graph objects. For example, consider the following input *(I1,O2),(I2,O1)*. The algorithm output is going to be *(I1,O1),(I2,O2)*, demonstrating that neither of the elements of the distinguished subset is contained in the input set. Instead, graph objects have been reorganized or merged to form new elements, hence the name distinguished merging.

Next we discuss how GReAT implements the distinguished merging algorithm. The pattern matching is left intact, because the distinguished merging is performed as a separate step in the last phase of the rule execution. The input set is provided by the pattern matching, but the sorting criteria must be specified by the programmer. He can provide the sorting criteria by compare functions, like in the case of sorting, only that all the output ports must have a compare function specified, so that each set can be sorted and made unique during steps 3 and 4, respectively. Figure 9 shows a pattern that selects "OutputPort"-s and "InputPort"-s in "BaseComponent"-s. Here, the same predicate "YPosCmp" used as a compare function for both output ports. In addition, a rule attribute "distinguished" must be set to true (not shown in the figure). The selected component ports are connected in a subsequent rule.

It is interesting that distinguished merging can also be used for selecting the source and destination components in another rule earlier in the transformation. Figure 10 shows a rule that selects component pairs that are laid out horizontally right next to each other. Figure 11 shows how the rule works. Let us denote the four input components to connect as C1, C2, C3 and C4. The pattern matching first finds all possible ordered pairs from the set of input components (there are 4!/2!=12 such pairs). To ensure that component

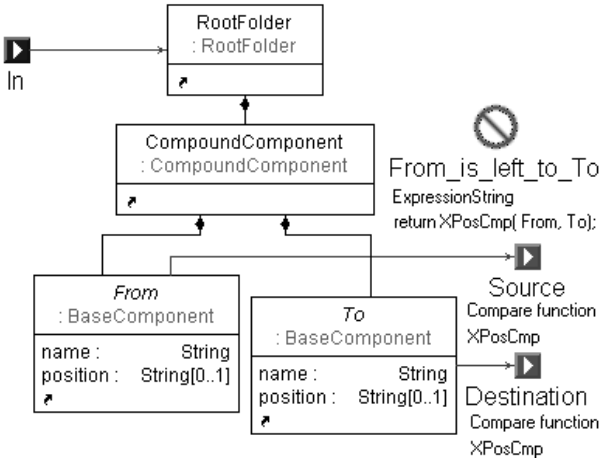

Fig. 10. Rule that selects adjacent component pairs

Input: {*(C1,C2),(C1,C3),(C1,C4),(C2,C1),(C2,C3),(C2,C4),*
(C3,C1),(C3,C2),(C3,C4),(C4,C1),(C4,C2),(C4,C3)}
After Guard: {*(C1,C2),(C1,C3),(C1,C4),(C2,C3),(C2,C4),(C3,C4)*}
After step 1: X={*C1,C1,C1,C2,C2,C3*}, Y={*C2,C3,C4,C3,C4,C4*}
After step 2: X={*C1,C1,C1,C2,C2,C3*}, Y={*C2,C3,C3,C4,C4,C4*}
After step 3: X={*C1,C2,C3*}, Y={*C2,C3,C4*}
After step 4 & output: D={*(C1,C2), (C2,C3), (C3,C4)*}

Fig. 11. Selecting source-destination rule pairs

"From" lies to the left of component "To", we utilize the compare function "XPosCmp" in a guard condition "From_is_left_to_To", which discards all pairs where "From" is right to "To". Finally, the distinguished merging gets rid of all non-adjacent pairs. These source-destination component pairs are then passed along one-by-one to the rule depicted in Figure 9 shown above.

We believe that distinguished merging is a powerful concept and its application is not limited to a specific problem. Besides selecting ports to be connected and identifying adjacent components, we used distinguished merging to form pairs of data read and write blocks, where blocks with identical names were selected to implement object (un)marshalling between remote components. Formally, the distinguished merging defines a bijective mapping between the elements of two sets. In the language of graph theory, the distinguished merging finds a perfect match in a complete bipartite graph.

We realize that restricting the resulting sets to be of the same size is a strong constraint, and we plan to relax this by allowing the user to specify an action that will be executed when the sets are of uneven size (e.g. ignore incomplete tuples, throw an exception, etc.)

6 Related work

Similar graph transformation tools such as AGG [9], PROGRES [10], Fujaba [11] and VIATRA [12] offer various levels of support for the features we have described here. The concept of a global containers is readily supported in all of these languages because the search for a given pattern graph is always performed on the entire host graph. However, this also implies that the search for a given pattern graph will be exponential in the number of objects contained in the pattern graph in the worst case.

In regards to sorting and distinguished merging, there is no direct support for either of these features in the above tools, though it seems possible to sort matches with a complex sequence of ordered rules. VIATRA provides a similar control flow language for explicit rule sequencing, and also allows the pre-binding of incoming objects, but does not directly support the sorting of pattern matches. AGG, Fujaba and PROGRES provide capabilities for deterministic rule sequencing, but do not support the pre-binding of objects to rules, so a sequence of rules could possibly be used to implement sorting.

Also, most of the above tools allow the user to specify whether a given pattern in the pattern graph should be an isomorphic or non-isomorphic image in the host graph, but the isomorphic matching is done non-deterministically; thus, the deterministic distinguished merging as we have described above would have to be implemented in several rules.

In general, our experience brings up a highly relevant question for the graph transformation-based languages: if we want to implement complex transformations in them, what is the minimal set of capabilities that is simple enough to be part of the core semantics of the language, yet powerful enough such that complex transformations can be implemented using them in a compact manner? We believe answers to these questions are not readily available and require further research.

7 Summary and Conclusions

In this paper we have identified three shortcomings with the original design of a graph transformation language and showed how the language could be improved to address those problems. The improvements were made by introducing new constructs in the language, and were orthogonal: all programs written with the earlier version preserved their semantics. We believe these extensions were necessary because the early version either simply did not have support for them, or they could be used only in a very inconvenient way. We have applied the extensions in developing some specific model transformation

solutions and their application lead to smaller and cleaner programs.

A number of research questions remain that we would like to address in future research. These questions are related to the extensibility of the transformation language, and include such issues as: How to provide ways to extend a transformation language with constructs similar to the ones presented here? How can one precisely describe the semantics of the extensions? How can we extend the execution environment (interpreter and the code generator) to support the extensions? We hope that answering these questions will lead to a new degree of extensibility and capabilities for model transformation tools.

References

[1] Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration, Vol 4. No1, DOI: 10.1007/s10270-004-0073-y, Journal of Software and System Modeling, 2004.

[2] Karsai, G., Agrawal, A., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Simple Language for Graph Transformations, in review for Journal on Software and System Modeling, preliminary version is available from www.isis.vanderbilt.edu

[3] Vizhanyo, A., Agrawal, A., Shi, F.: Towards generation of high-performance transformations.In: Proc. Generative Programming and Component Engineering. Lecture Notes in Computer Science, Springer-Verlag (2004)

[4] Agrawal A., Karsai G., Shi F.: Graph Transformations on Domain-Specific Models, Technical report ISIS-03-403, Vanderbilt University, November, 2003.

[5] Karsai G., Agrawal A., Shi F., Sprinkle J.: On the Use of Graph Transformations for the Formal Specification of Model Interpreters, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.

[6] Mathworks Stateflow semantics documentation, http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/semantics.html

[7] Neema, S., Kalmar, Z., Shi, F., Vizhanyo, A., Karsai, G.: A Visually-Specified Code Generator for Simulink/Stateflow , in review for IEEE Symposium on Visual Languages and Human-Centric Computing, 2005.

[8] Agrawal, A., Karsai, G., Ledeczi, A.: An End-to-End Domain-Driven Software Development Framework. Conference on Object Oriented Programming Systems Languages and Applications, 2003.

[9] Taentzer, G.: AGG: A Tool Enviroment for Algebraic Graph Transformation. In Proc. of Applications of Graph Transformation with Industrial Relevance, Kerkrade, The Netherlands, LNCS, Springer, 2000.

[10] Schurr, A.: PROGRES for Beginners. 1997. http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-download_file.php?fileId=232

[11] Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment, Proc. ICSE: The 22nd International Conference on Software Engineering, Limerick, Ireland, ACM Press, 2000.

[12] Varró, D., Pataricza, A.: Mathematical model transformations for system verification. Tech. rep., Budapest University of Technology and Economics, 2001.