Applying a Grouping Operator in Model Transformations

Daniel Balasubramanian, Anantha Narayanan, Sandeep Neema, Benjamin Ness, Feng Shi, Ryan Thibodeaux, and Gabor Karsai

Institute for Software Integrated Systems, 2015 Terrace Place, Nashville, TN 37235
{Daniel,Ananth,Sandeep,BNess,FengShi,RThibodeaux,
Gabor}@isis.vanderbilt.edu

Abstract. The usability of model transformation languages depends on the level of abstractions one can work with in rules to perform complex operations on models. Recently, we have introduced a novel operator for our model transformation language GReAT that allows the concise specification of complex model (graph) rewriting operations that manipulate entire subgraphs. In this paper we show how the new operator can be used to implement non-trivial model manipulations with fewer and simpler rules, while maintaining efficiency. The examples were motivated by problems encountered in real-life model transformations.

Keywords: Model Transformation, Graph Transformation.

1 Introduction

Model-based development necessitates the use of model transformations. The cost of setting up a model-based development tool chain depends on how economical it is to implement possibly complex yet necessary model transformations on an ad hoc basis, and whose correctness is often essential for the usability of the toolchain. Thus, higher-level techniques for specifying model transformations have been proposed, and one promising conceptual framework for specifying model transformations is based on graph transformations [1].

The practical application of graph transformation-based model transformation approaches [10] has shown that while the high-level nature of the graph rewriting rules is very powerful, sometimes writing common operations is very tedious. Commonly, graph rewriting operations match a subgraph of a host graph and then create (or remove) nodes and edges in the model graph and possibly modify attributes. Due to the difficulty of performing equivalent operations using Java methods over some primitive graph API, using the graph-based specification has clear advantages.

A matched subgraph typically has a simple structure (pattern nodes and pattern edges are bound to host graph nodes and host graph edges), and it is hard to form closures over such subgraphs. The closure would group together multiple matches of pattern nodes and edges into a graph that is treated as a unit in the context of some subsequent operation, typically un-gluing or gluing this graph with other nodes, copying the graph, or removing the group altogether.

A. Schürr, M. Nagl, and A. Zündorf (Eds.): AGTIVE 2007, LNCS 5088, pp. 410–425, 2008.

In our graph transformation-based model transformation tool, GReAT [1], we have introduced support for such closures over pattern matches. We call it the 'grouping operator', and we extended the semantics of our graph transformation rules with this new operator. This work has been reported in [3]. In this paper we briefly review the semantics of the operator and then give a number of examples —derived from real-life applications— that illustrate the use of the operator. The paper concludes with a review of related approaches and a summary.

2 Recap of Group Operator

In this section we briefly review the fundamentals of GReAT, highlighting the features of the grouping operator, first introduced in [3].

The Graph Rewriting and Transformation Language (GReAT) is a graphical language used for the specification and execution of model transformations defined using elements from the meta-models of the source and target languages. The entire language consists of three sub-languages: a pattern specification language, a transformation rule language, and a control-flow language. The pattern specification language is used to define the patterns matched in the host graph. The transformation rule language allows the creation and deletion of objects in the host graph, along with the modification of object attributes. Finally, the control-flow language allows a transformation to be explicitly sequenced.

The basic rewriting unit of a transformation is a *Rule* that contains two pieces: 1) a pattern to match with the host graph (defined using the pattern specification language), and 2) an action to perform after the matches are found (defined using the transformation rule language). When a *Rule* executes, it first finds all valid matches of the specified pattern in the host graph. After all matches are found, the rule actions are executed. These rule actions can include deleting and creating new elements in the host graph, as well as modifying attributes. Finally, user selected elements are passed to the next rule in the sequence (specified using the control-flow language).

2.1 Motivation for the Group Operator

As described above, the result of the first stage of a rule execution is a set of matches, where each match is a unique binding for the pattern variables in the rule. The limitation with this is that all the matches (of a single pattern) are isomorphic. In other words, all the matches have the same, predetermined number of nodes and edges. It would be useful to form closures over such unique matches, such that more complex sub-graphs can be matched and manipulated using the same rule.

For instance, consider a chain of nodes of arbitrary length, as shown in the graph in Fig. 1. Suppose that we wish to select such chains of arbitrary length and move them to a different container. The pattern shown in Fig. 2 matches two connected nodes at a time. This will result in the matches (a—b), (b—c) and (c—d). While it is possible to use these matches to move the chain to a new container, the description would be cumbersome (especially since moving it in parts would create dangling edges). The group operator allows us to group all these matches together, so that the selected sub-graph is the entire chain, as opposed to pairs of nodes. Using the group operator, the entire chain can be selected and copied in a single rule.

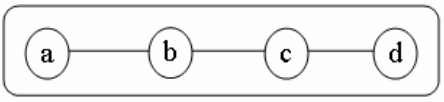

Fig. 1. A chain of arbitrary length

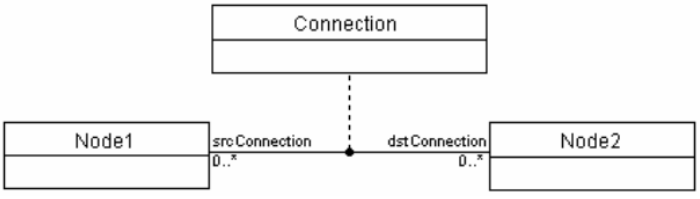

Fig. 2. Graph rule for matching a connection

Consider a container that contains a number of nodes, with a "name" attribute associated with each node. Multiple nodes are allowed to have the same "name", and the container can have any number of nodes, with any number of different "name" attributes. Fig. 3 (a) shows such a container. Suppose that we wish to sort the nodes with the same "name" into a single container, as shown in Fig. 3 (b). This involves creating an arbitrary number of containers (depending upon how many different "name" values are present), and moving an arbitrary number of nodes into each container. The group operator allows us to accomplish this in a single rule, when we use the "name" attribute as the grouping criterion. For the example shown in Fig. 3, the rule produces four matches of different sizes (one match containing four nodes, one with three, and two matches containing two nodes).

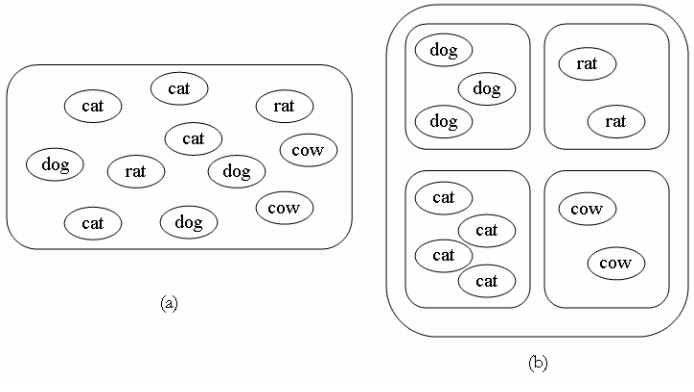

Fig. 3. Sorting items into containers

As shown above, it is useful to specify queries that can produce matches of different, arbitrary sizes in a single rule. The group operator is a new construct introduced into the GReAT language to allow the results of multiple matches to be combined so that larger graph patterns could be specified in a compact manner. In comparison, queries in PROGRES[12] allow the use of complex logic statements to construct a

result set, which can contain elements of different and arbitrary sizes. Our approach provides a graphical abstraction that is easier to visualize and use. Similar to GReAT, AGG[14] allows the specification of additional conditions on the attributes of pattern objects in a rule, but the matches are still isomorphic.

In general, a transformation can benefit from the group operator if the user must have the ability to specify a pattern that will match a variable number of objects. This often includes chains of objects with connections between them, such as the first example above, or large groups of objects that need to be separated into smaller groups based on common attributes, as in the second example given above. Also, a group operator should be used if sub-graphs (composed of multiple matches) need to be moved or copied into different containers, as these patterns can be difficult to specify in GReAT without this construct.

Specifying a rule with a group operator is relatively simple. The user adds one group operator to the rule, and then specifies two additional items:

1) Which elements of the overall pattern will be used to form the subgroups that are the subgraphs formed from the individual matches. For instance, in **Fig. 2**, *Connection*, *Node1* and *Node2* must be selected. The purpose of the rule should make the selection of which pattern objects are needed obvious.
2) The Boolean expression that will determine when two matches should be placed into the same subgroup (we use the prefixes "the_" and "other_" to identify the matches). In Fig. 2, this would simply be true, since all the connections form a single group. In Fig. 3, we would use "`the_Node.name() == other_Node().name`". This step is the crucial part of using the group operator. Matches from individual rules are often inserted into subgroups based on an attribute value of one of the objects, as in the second example above. In cases such as these, the Boolean expression is quite simple, and can even be written by novices who have limited experience with GReAT. Examples of both simple and complex Boolean expressions for subgroup formation are given in the following sections. Detailed information about subgroup formation can be found in [3].

After all matches are found, each match is placed into precisely one subgroup by evaluating the Boolean expression using this match and the matches already placed in subgroups. If the expression evaluates to true, then the current match is placed into the subgroup against which the Boolean expression was evaluated. If the expression yields a "false" for all of the matches in existing subgroups, then the current match is placed in a newly created subgroup. Finally, the rule's action is performed on a per-subgroup basis instead of a per-match basis. In this manner, one can effectively combine several matches into one "larger" match and then perform actions on that larger match. Additionally, the user can also choose to move or copy the elements of the subgroups to another parent container.

The next several sections give examples of the use of the group operator. The examples presented here were derived from actual experience with real-life modeling and model transformations. However, for the sake of brevity they have been simplified to a presentable form. Note also that model transformations are often applied to legacy modeling paradigms that are imperfect, yet cannot be discarded because of the investment in the models.

3 Separating a System into Its Subsystems

A system may be comprised of various subsystems which share common components. For instance, in a building, the electrical, plumbing, and networking subsystems all use rooms as a common component for distribution hubs and endpoints. Because of this, a model for such a system will necessarily have all subsystems represented, overlapping one another on top of their common components. However, at some point it may be necessary to separate the individual subsystems for the purposes of verification, construction, or clarity. GReAT, with its group operator, can be used to specify such a transformation compactly. To demonstrate this, both a "building" meta-model and a building model based on this meta-model, are presented, along with a GReAT transformation rule that makes use of the group operator to separate the model-building's electrical subsystem from its other subsystems.

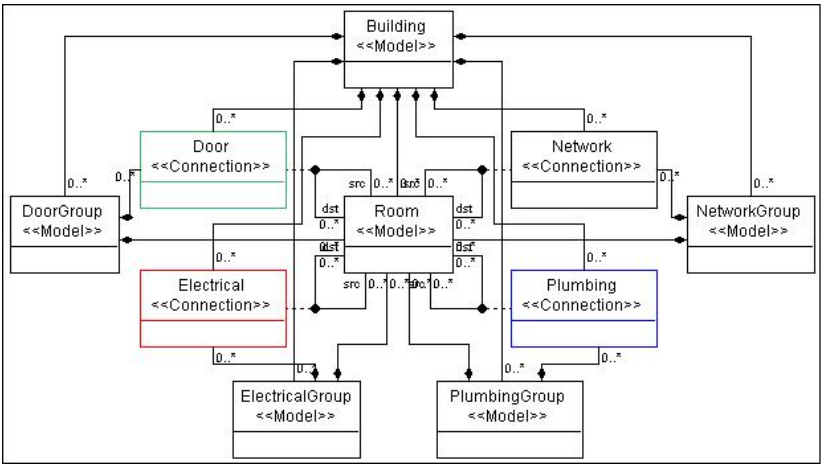

Fig. 4. Building Meta-Model

Fig. 4 shows a meta-model for buildings with electrical, plumbing, networking, and room-connectivity (i.e., door) subsystems. *Room* is the component they share as their infrastructural basis. For instance, *Electrical* connections are between rooms, i.e. between the rooms' sockets and switches, as are *Network* connections, for example, between a server room and other rooms' network ports. Note also that the meta-model has components (*DoorGroup*, *NetworkGroup*, *ElectricalGroup*, and *PlumbingGroup*) that can contain copies each of these subsystems in isolation.

The left side of Fig. 5 shows a simple model based on the building meta-model of Fig. 4. Electrical connections are drawn with dashed lines, while the solid connections represent other types of connections. As can be seen, multiple subsystem types are represented and are overlapping on top of their common *Room* components. For the purposes of building construction, it would be of great utility to be able to separate out these subsystems. For instance, the electrical subsystem (*ElectricalGroup*) in isolation could be given to an electrician for wiring purposes. Similarly, the room-connectivity (*DoorGroup*) subsystem could be presented to an inspector or put through a model checker to make sure the building conforms to certain safety codes.

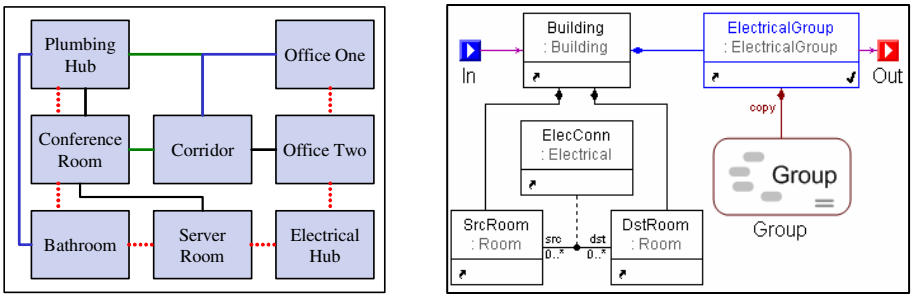

Fig. 5. Building Model (left) and Grouping Rule to Isolate the Electrical Subsystem (right)

The right side of Fig. 5 shows a rule in a GReAT transformation that uses the group operator to separate out the electrical subsystems of the model to its left. The rule creates a new *ElectricalGroup* model that will hold a copy of this subsystem. The group operator contains any *Electrical* connections matched by the rule, along with the rooms connected by these connections. The Boolean expression the group operator uses to group matches together is trivial: it is simply the value "true", indicating that any and all *Electrical* connections and their associated rooms should be included -- this results in a single group that is the entire electrical subsystem. For a more complex setup, the grouping rule could group electrical connections based on the breaker from which they originate, or based on the voltage they carry. Rules for isolating other building subsystems are similar.

Fig. 6 shows the results of applying the GReAT transformation rule in Fig. 5 to the model on its left. The electrical subsystem is successfully separated from the other subsystems. Such a diagram could be useful for an electrician or inspector.

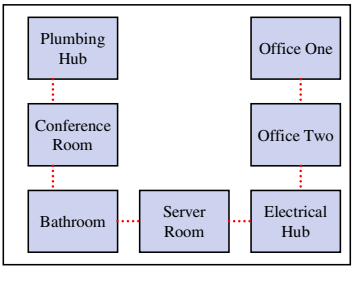

Fig. 6. Building Model's Electrical Subsystem

4 Creating Proxies for Distributed Communication

Models of control systems often consist of dataflow blocks representing mathematical functions that manipulate data obtained from the environment or other input sources to achieve some desired output effect on the controlled system. Control designers initially formulate their controllers with no concern regarding the eventual deployment architecture; however, the implementation is commonly distributed over

separate physical nodes (called 'components') that must pass data between their contained functions over a bus infrastructure in a timely and predictable manner to achieve the desired controlled outputs. To facilitate software abstraction and reusability [13][7], direct dataflow connections between functional blocks deployed on different components are often managed using a proxy on the component hosting the receiver blocks. The bus implementation and the proxy are responsible for marshalling and transferring data between components, thereby allowing the receiver blocks to interact with the proxy through an interface identical to that of the original sender without concern of how to access data from the bus.

Assume we have a system deployed across a set of distributed nodes called *Components*. Each component contains a dataflow graph consisting of *Functions* that pass data to and from each other through *Ports*. A connection between two ports is called a *Dataflow,* and it connects only one sending port to one receiving port. Still, one port can send data to multiple receiving ports through multiple dataflows from the port to each receiver. Dataflows can also connect ports of two functions deployed on separate components.

To implement a modeled distributed system described previously, a *Proxy* for a function should be created in all components that receive input from that function. The sending function should be connected to receiver functions through its proxy instead of direct dataflows to the receivers' ports. The *Group* operator provides a means to perform this operation within GReAT by creating a single proxy of a function on other components that use the function's output within their respective dataflow graphs. The proxy inserted into a component will have an input port and an output port for each output port in the sending function connected to inputs of receiving functions on one component. This implements a mirroring of the interface of the sender's ports that concern the functions on the target component.

Fig. 7 shows the Group rule in GReAT for starting this transformation. It is responsible for creating a single proxy for *Function1* within C*omponent2* if output ports of *Function1* are connected to input ports of *Function2* within *Component2*. The rule takes in the top-level *System* object as its input. The rule first finds all the component objects in the system, and the *Guard* condition code, `Component1.uniqueID() != Component2.uniqueId()`, removes matches where *Component1* and *Component2* are the same component. Now, the rule returns the set of dataflow connections between function ports within these unique components. Once the dataflows, *DataflowFF,* that connect ports of *Function1* and *Function2* are matched, the subgroups consisting of the sending functions' output ports, *F_Output*, must be formed. The grouping criterion of the group operator restricts membership to a subgroup to unique *F_Output* objects in the same function, *Function1*. A single subgroup holds the output ports needed by a single proxy on a component. According to the Group rule execution semantics, objects with the *CreateNew* action (marked with a checkmark) will be created for each subgroup. This creates a single *Proxy* object within *Component2* for each subgroup, and the Group action of *copy* will copy the ports in the subgroup into the proxy. The output packets of the rule will consist of the ports holding the dataflows across separate components and the components where these ports are located. All *Component2* objects found in the match will now contain appropriate proxies, and each proxy will contain the output ports copied from its representative function, *Function1*.

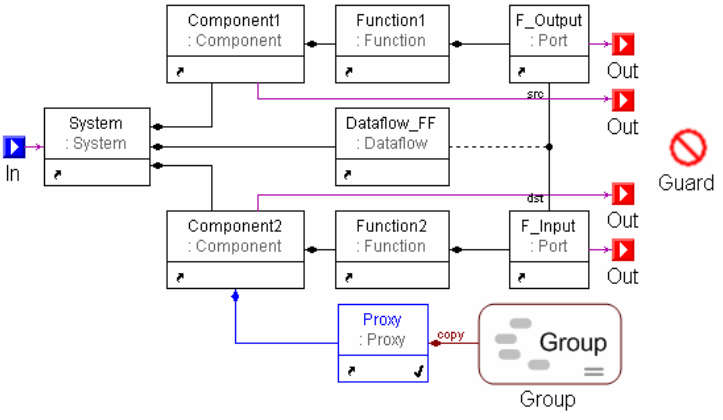

Fig. 7. Group Rule for Creating Proxies with Ports

For each proxy, direct dataflow connections from the original sending function to the receiving functions need to be replaced by the appropriate connections from the proxy to the receiving functions. Also, the same output ports of the sending function must be connected to appropriate input ports of the proxy. The rules to create these dataflows and the input ports of the proxy are shown below in Fig. 8, and they can be executed in parallel.

The left rule of Fig. 8 creates the dataflow connections from a proxy's output ports to the receiver functions' input ports. The output packets from the rule in Fig. 7 contain all of the dataflow connections between *Function1* and *Function2*. The *Guard* condition restricts found matches to those where the output port of the proxy, *P_Output*, has the same name as the output port of *Function1, F_Output*. The resulting matches are the correct ports since the output ports copied into the proxy were the output ports of *Function1* in the previous rule, i.e., they will have the same name. Once matched correctly, the connection from the *F_Output* to the input port of *Function2*, *F_Input*, is replaced by the connection from the output of the proxy, *P_Output*, to *F_Input* through a dataflow, *Dataflow_PF*. The procedural code (contained in *AttributeMapping)* renames the proxy object to be "P" plus the name of its corresponding function, *Function1*, to make it identifiable in the resulting model.

The right rule in Fig. 8 matches the sending function's output ports, *F_Output*, to their copied instances within a proxy on another component. The *Guard* condition restricts found matches to those where the output port of the proxy, *P_Output*, has the same name as the output port of *Function1, F_Output*. For each match, a new input port, *P_Input*, is created within the proxy. This provides an input interface in the proxy that mirrors its output ports. Each output port of the sending function, *F_Output*, is then connected to its corresponding proxy input port. The procedural code (*AttributeMapping)* renames a created input port to be "In_" followed by the name of the output port, *F_Output*, it is now connected to through a dataflow, *Dataflow_FP*.

Fig. 9 shows an example input model that needs proxies included in the components. The boxes labeled C1, C2, and C3 represent components, those labeled F1, F2… F9 represent functions, and arrows within functions represent ports. The arrow indicates the directional flow of data, and directed lines between ports are dataflow

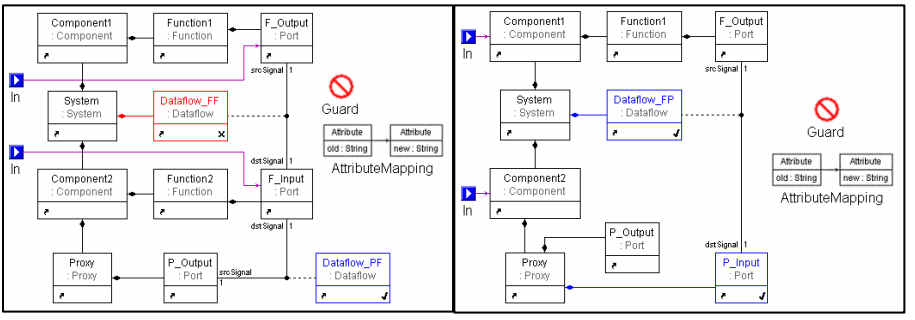

Fig. 8. Rules for Creating Connections to/from a Proxy

connections. The circles represent system inputs and outputs and are of no concern regarding the transformation. The functions F5 and F7 appear twice in components C2 and C3, respectively, as unique instances of the same function block.

Looking at the input model, we see that functions F2 and F3 are the only functions with connections across different components; therefore, proxies must be created for these functions.

Fig. 10 shows the resulting output model of the system after the rules above are applied. During the execution of the first rule, a total of three subgroups would be created for the entire system model. Each subgroup has a corresponding proxy, PF2 and PF3 in this model, where the number identifier for each proxy matches the number identifier for its corresponding function, e.g., PF3 → F3.

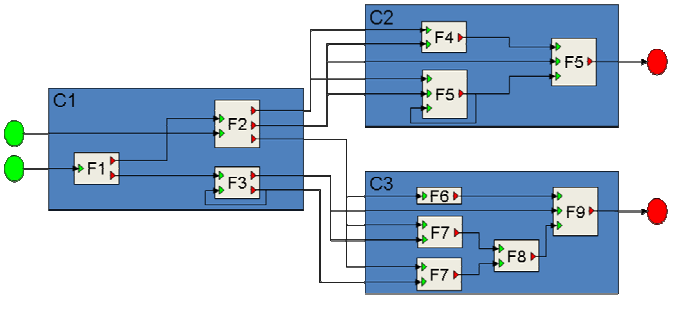

Fig. 9. Sample Input Model

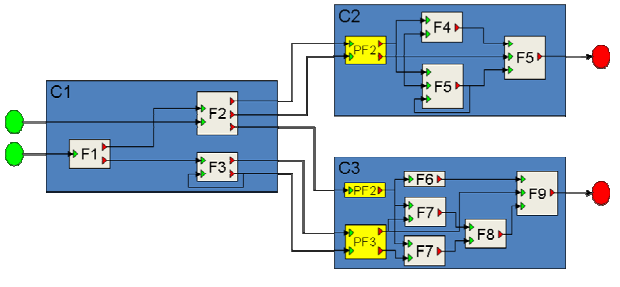

Fig. 10. Sample Output Model

Notice that the proxies' port interfaces are not the same on different components, e.g. PF2 in C2 has a two port interface whereas PF2 in C3 has a one port interface. This occurs since the ports copied into a proxy are the ones explicitly needed by functions on the same component and not the entire set of output ports of the sending function.

5 Shared Variables in a Dataflow Model

A dataflow model consists of *Blocks*, which are connected through *Ports*. Connections between these ports are called *Lines*, which represent flow of data between the blocks through their ports. While the ports may be classified as *input* and *output* ports, the flow of data may be from one input port to another (or one output port to another), as in the case of a hierarchical block, where a top level input port may be passing information along to the lower level blocks.

A single port may be connected to multiple ports through multiple lines. In generating code from such models, we would like to use a single shared variable to represent all such lines. We would also like to make a temporary 'cross-link' association between each line and its shared variable, for specific purposes necessary later in the code generation. The *Group* operator offers a convenient way to achieve this in GReAT. We will see how a Group rule can be used to identify groups of such lines and create a new variable for each such group, and generate the cross-links.

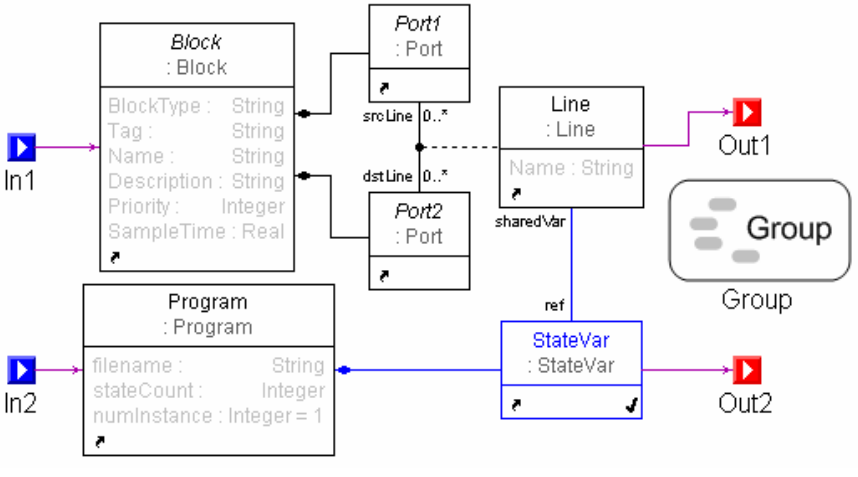

Fig. 11. Group Rule for Creating a Shared Variable

Fig. 11 shows the Group rule in GReAT for creating a single shared variable for a group of lines. For each block, we take the ports contained in that block and group the lines originating from that port. Note that the port may be connected to other ports which are not contained in the same block. *Port1*, *Port2* and *Line* are added to the Group object, and the grouping criterion is set as: `the_Port1 == other_Port1`. This results in multiple lines originating from a single *Port* object in a *Block* being

grouped together. A new variable is then created inside *Program*. Since this is a Group rule, the *CreateNew* action fires once for each subgroup that has been created. This means that a single variable will be created for each set of lines that have been grouped together.

After creating the new variable, we associate it with its *Line*, using a cross-link. This is indicated by the line with the role names *ref* and *sharedVar*. Since the cross-link is a *simple association*, it will be created for each match, between the *Line* in that match, and the *StateVar* created for the group that this match is placed in.

Fig. 12 shows a section of a dataflow diagram. The port *Res* has three *Lines* and the port Con has two Lines coming from it. Let the *Line* connected to port *Pi* be called *Li*. Then, *L1, L2, L3* will be in one subgroup, and *L4* and *L5* will be in another subgroup.

The group rule creates a new shared variable *Var1* for the subgroup {*L1*, *L2*, *L3*}, and another shared variable *Var2* for the subgroup {*L4*, *L5*}. The output packets generated from the group rule are {*L1*, *Var1*}, {*L2*, *Var1*}, {*L3*, *Var1*}, {*L4*, *Var2*} and {*L5*, *Var2*}.

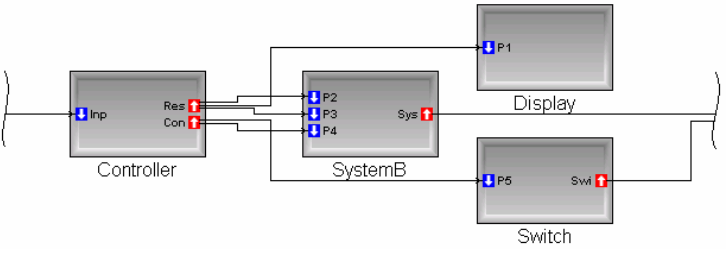

Fig. 12. Sample Dataflow Diagram

6 Ordered Binary Decision Diagram Reduction

Ordered Binary Decision Diagrams (OBDDs) [4] are used for representing and evaluating Boolean functions. Arbitrary OBDDs can often be reduced, using algorithms such as those described in [9], to a more compact representation. This example describes how the group operator simplifies the specification of a transformation that performs this reduction algorithm.

An OBDD can be thought of as a rooted tree consisting of nodes that have variable assignments. A node whose variable is 0 or 1 is called a terminal node, and is called a non-terminal node otherwise. Each non-terminal node contains two outgoing connections: one “low” connection and one “high” connection. If a node’s variable is assigned a value of 0, then the low connection tells which node to evaluate next, and if a node’s variable is assigned a value of 1, then the high connection tells which node to evaluate next. The value of the function for a particular assignment of values to variables is given when a terminal node is reached.

The reduction algorithm in [9] begins by assigning an integer label to each of the nodes in the diagram in the following manner. The first 0-node that is encountered receives the first label (for instance, #0). All of the other terminal 0-nodes have the same value, so they also received the same label. In the same way, all of the terminal

1-nodes receive the next label (#1). Next, we define two terms: given a non-terminal node n, lo(n) is defined to be the node pointed to by the low connection from n (drawn here using dashed lines), and hi(n) is defined to be the node pointed to by the high connection from n (drawn here using solid lines).

The rest of the algorithm proceeds in a bottom-up manner as follows. To assign a label to each of nodes at level i, we assume that we have already assigned a label to all of the nodes at all levels j such that $j > i$. That is, we assume that all of the nodes on levels below the current level have been labeled. A node n at level i receives its label in one of the following three ways:

1. If the label of the node lo(n) is equal to the label of the node hi(n), n also receives this same label
2. If there is another node m such that n and m have the same variable x_i, and the labels of lo(n) and lo(m) are equal and the labels of hi(n) and hi(m) are equal, n receives the same label as m.
3. Otherwise, n receives the next unused integer as its label.

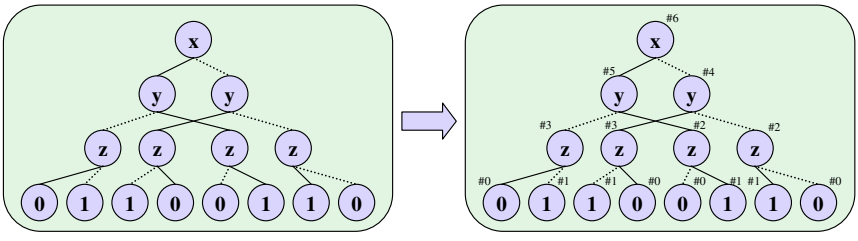

Fig. 13. Unlabeled OBDD (left) and Labelled OBDD (right)

Fig. 13 shows an example OBDD both before and after the labeling algorithm has been applied. In the rest of this example, we assume that the input to our transformation rules is the labeled OBDD on the right of Fig. 13.

After the labeling of the OBDD, the next step of the reduction algorithm of [9], and the transformation we will describe below, removes redundant nodes based on their labels. That is, for each group of nodes with the same label, it creates one new node with that same label and creates the connections between these "reduced" nodes appropriately.

There are two re-writing rules we must write to perform this transformation:

1. Determine which nodes are equivalent based on their labels, and for each set of equivalent nodes, create a new node.
2. Create the low and high connections between the newly created nodes based on the low and high connections that exist between the old nodes.

Fig. 14 shows a rule that performs the first step of our transformation. The incoming context for the rule (the objects bound to the input ports) are the diagrams in which the nodes are found. The diagram named *OldDiagram* is the non-reduced OBDD, and the diagram named *Diagram* will be the reduced OBDD. This rule first finds all of the nodes in *OldDiagram*. A group operator is present, and contains as its

only members the nodes found in *OldDiagram*. The subgroups are formed by iterating over each match (which each consist of a single node found in *OldDiagram*) and evaluating the user-specified grouping criteria against matches already in subgroups; in our case, two matches should be inserted into the same subgroup if the values of their labels are equal.

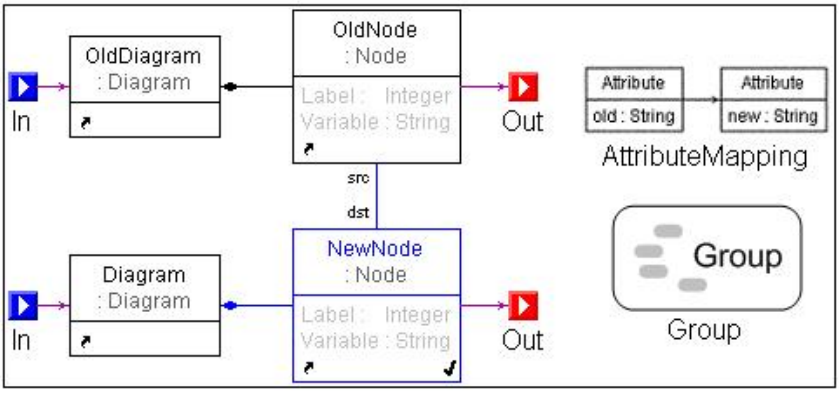

Fig. 14. Rule to Create Nodes

The grouping criteria code that will accomplish this is: `the_OldNode.Label() == other_OldNode.Label()`. After subgroup formation, new objects are created on a per subgroup basis, and new associations are created on a per match basis. Thus, our rule creates one new node in our reduced OBDD (*Diagram*) for each group of nodes that have the same label in the unreduced OBDD, and also creates a temporary association between the newly created nodes and the node in the unreduced OBDD to which the new node corresponds; this temporary association will be matched in the next rule to connect the new states together. Finally, the procedural code (*AttributeMapping*) takes care of setting the values of the attributes (the label and variable values) of the newly created nodes.

The next rule in the sequence, shown in Fig. 15, is responsible for connecting the nodes of the reduced OBDD. The incoming context consists of two elements: a node from the unreduced OBDD (labeled *OldNode*) and the corresponding node in the reduced OBDD (labeled *NewNode*). The rule finds all of the connections in the unreduced OBDD such that *OldNode* is the destination of the connection; it then finds the node in the reduced OBDD that corresponds to this "source" node in the unreduced OBDD by matching the association created in the previous rule (labeled with the rolenames *src* and *dst* in both places). Remember that *NewNode* is already the node in the reduced OBDD corresponding to *OldNode* because they are passed from the previous rule together. The *AttributeMapping* block takes care of setting the connection to the proper type, low or high, with the following code: `NewConnection.Type() = OldConnection.Type()`.

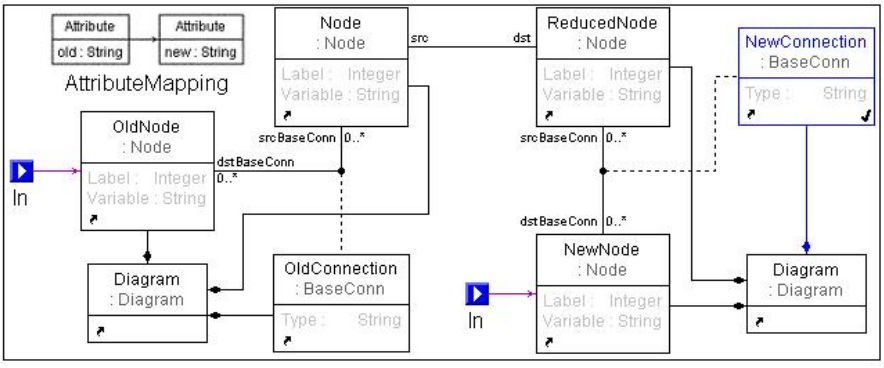

Fig. 15. Rule to Connect States in the Reduced OBDD

The resulting connected and reduced OBDD is shown in Fig. 16.

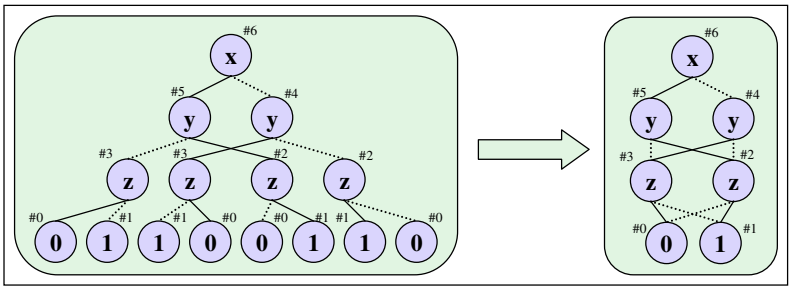

Fig. 16. Connected and (Partially) Reduced OBDD

7 Related Work

Hoffman et al. [5] introduced transformations on frame bounded subgraphs which restrict graph edges from crossing frame boundaries. The copying of such a delimited subgraph permits copying only the nodes and edges contained within a frame. The group operator uses a similar idea when performing actions on grouped objects: only user selected nodes and edges that belong to a group under the membership criterion will have the group action (bind, move, or copy) performed on them. Following the application of the action specified for the group, all edges with an endpoint outside of a subgroup, including those to other formed subgroups in the same rule, will be removed.

Van Gorp et al. [11] implemented the copying of subgraphs in the MoTMoT project [8] using the "copy" and "onCopy" operators. The "copy" and "onCopy" operators provide means to perform deep copies on models and/or copying specified nodes and edges within a rule; however, it is not obvious how a user could implement in MoTMoT transformations presented above using the group operator within the GReAT language. The difficulty for MoTMoT to recreate the same transformations in an equally small number of rules would arise because it does not appear to have the

ability to subdivide the set of all matches in a rule based on a conditional expression and perform actions only on the formed subgroups instead of every match. The group operator extends the normal rule execution semantics in GReAT by allowing the application of copying, or other actions, on a per set/group basis.

As alluded to previously, the ability to handle and manipulate matched objects as sets is a prerequisite to match the capabilities of the group operator. Even though they rely on textual specification of transformations, ATL [2] and VIATRA2 [15] do not appear to be less expressive or powerful than graphical languages such as GReAT. VIATRA2 explicitly uses an Abstract State Machine (ASM) based language and ATL matches many of the ASM constructs; therefore, it is no surprise that both languages provide a data type for handling sets and other mathematical multi-object types, and providing a grouping criterion as a Boolean expression would require no extension to the languages. Also, performing actions on a per group basis would involve using the "foreach" command, common to both languages.

8 Summary and Conclusions

This paper has shown examples for the practical application of a high-level grouping operator in a graph-transformation based model transformation language. The examples provided were derived from practical problems and clearly show the use of the operator to allow more abstract and concise descriptions of complex transformation steps. This simplifies the transformation specification, making it easier to write and maintain.

We have implemented the operator in the GReAT interpretive transformation engine (GRE), and we have a prototype implementation of a code generator that compiles the rules with the group operator into executable code. However, it is the topic of further research how to generate efficient executable code from such rewriting rules. Another research topic is related to the restrictions we have placed on the group operator: these restrictions make the implementation of the group-rules straightforward, but it is not clear how well they stand up in practice. We plan to investigate how these restrictions can be weakened while maintaining the powerful properties of the grouping operator.

Acknowledgements

The research described in this paper has been supported by a grant from NSF/CSR-EHS, titled "Software Composition for Embedded Systems using Graph Transformations", award number CNS-0509098, and by NSF/ITR, titled "Foundations of Hybrid and Embedded Software Systems", award number CCR-0225610.

References

1. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal on Software and System Modeling 5(3), 261–288 (2006)
2. ATL Project. An ECLIPSE GMT Subproject, `http://www.eclipse.org/m2m/atl/`

3. Balasubramanian, D., Karsai, G., Narayanan, A., Shi, F., Thibodeaux, R.: A Subgraph Operator for Graph Transformation Languages. In: GT-VMT 2007 Workshop at ETAPS (2007), http://www.cs.le.ac.uk/events/GTVMT07/
4. Bryant, R.E.: Graph Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
5. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical Graph Transformation. Journal of Computer and System Sciences 64, 249–283 (2002)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Series: Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
7. Farcas, E., Farcas, C., Pree, W., Templ, J.: Transparent distribution of real-time components based on logical execution time. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools For Embedded Systems, LCTES 2005, Chicago, Illinois, USA, June 15-17, 2005, pp. 31–39. ACM Press, New York (2005)
8. Van Gorp, P., Schippers, H., Jannsens, D.: Copying Subgraphs within Model Repositories. In: 5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Vienna, Austria (2006)
9. Huth, M., Ryan, M.: Logic in Computer Science: Modeling and Reasoning about Systems. Cambridge University Press, Cambridge (2000)
10. Personal communications with developers and researchers from industrial labs
11. Schippers, H., Van Gorp, P.: Model Driven, Template Based, Model Transformer (MoTMoT) (2005), http://motmot.sourceforge.net/
12. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRES. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 219–234. Springer, Heidelberg (1995)
13. Silva, A.R., Rosa, F.A., Gonalves, T., Antunes, M.: Distributed Proxy: A Design Pattern for the Incremental Development of Distributed Applications. In: Emmerich, W., Tai, S. (eds.) EDO 2000. LNCS, vol. 1999, pp. 165–181. Springer, Heidelberg (2001)
14. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062. Springer, Heidelberg (2004)
15. VIATRA2 Framework. An ECLIPSE GMT Subproject, http://www.eclsipse.org/gmt