Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the
Third International Workshop on Graph Based Tools
(GraBaTs 2006)

The Graph Rewriting and Transformation Language: GReAT

Daniel Balasubramanian, Anantha Narayanan, Chris vanBuskirk, and Gabor Karsai

8 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

The Graph Rewriting and Transformation Language: GReAT

Daniel Balasubramanian[1], **Anantha Narayanan**[1], **Chris vanBuskirk**[1], **and Gabor Karsai**[1]

[1]Institute for Software-Integrated Systems, Vanderbilt University, Nashville, TN 37235, USA

Abstract: In this paper, we describe the language and features of our graph transformation tool, GReAT. We begin with a brief introduction and motivation, followed by an overview of the actual language, the modeling framework, and the tools that were written to support transformations. Finally, we compare GReAT to other similar tools, discuss additional functionality we are currently implementing, and describe some of our experiences with the tool thus far.

Keywords: Model Transformation, Modeling Tools

1 Introduction

Model transformations are a central part of the model based approach to software engineering. The construction of model transformations using textual languages is time consuming, costly, and rarely has a formal foundation. Models can be considered as labeled multigraphs. Thus, model transformations can be accomplished using the techniques of graph transformations [EER96]. Since graph transformations are grounded in mathematical concepts, we can use them to formally specify the intended behavior of model transformations. The *Graph Rewriting and Transformation (GReAT)* language is a graphical language for the specification of graph transformations between domain-specific modeling languages (DSMLs). The rest of this paper describes the *GReAT* language and toolset, a comprehensive set of tools which can be used to specify, debug and execute efficient graph transformations between DSMLs.

2 The GReAT Language

The GReAT language consists of three sub-languages: the pattern specification language, the transformation rule language, and the sequencing or control flow language. Additionally, the input and the output languages of a transformation are defined in terms of meta-models.

The meta-information of the DSMLs on either side (i.e. input or output) of the transformation is specified using UML class diagrams. Transformations also often require the ability to maintain temporary information that may correspond to both paradigms; for instance, a transformation from a hierarchal state-machine language to one without hierarchy benefits from the ability to create temporary links between hierarchal states in the first DSML and flat states in the second DSML. GReAT allows the users to create additional, heterogeneous class diagrams that describe the cross-domain and temporary associations. Transformations in GReAT are specified over the set of meta-models and the heterogeneous class diagrams.

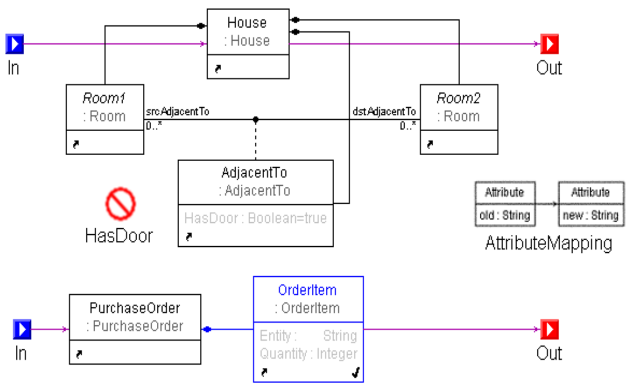


Figure 1: An example GReAT rule

2.1 Pattern Specification

GReAT uses the UML classes defined in the input meta-models to specify patterns that will be matched in the host graph. A pattern graph consists of nodes and edges, and each node and edge must have a corresponding class and association element in the class diagram of the input meta-model. For instance, the user might draw a simple pattern that selects all elements of type 'A' in a container class that are associated with elements of type 'B' in the same container, as three classes (container, A, B), two containment relations, and one association. In the event that a pattern requires that a given element *not* be present in the matching subgraph, the user can set the pattern cardinality on the containment association of this element to zero, effectively providing a *negative application condition*.

2.2 Transformation Rule

The basic transformation entity in GReAT is a production rule. A rule in GReAT has the following elements:

- *Pattern:* a graph with pattern vertices and edges,
- *Actions:* a mapping of pattern vertices and edges to the set of actions $\{Bind, CreateNew, Delete\}$,
- *Input interface:* a set of distinct input ports that can receive graph objects from previous rules,
- *Output interface:* a set of distinct output ports that will transfer graph objects to the next rule,
- *Guard:* an expression that evaluates true or false and determines whether a rule actions should be executed for the matched subgraph,
- *Attribute mapping:* code that is executed for each valid match to generate the values of edge and vertex attributes,

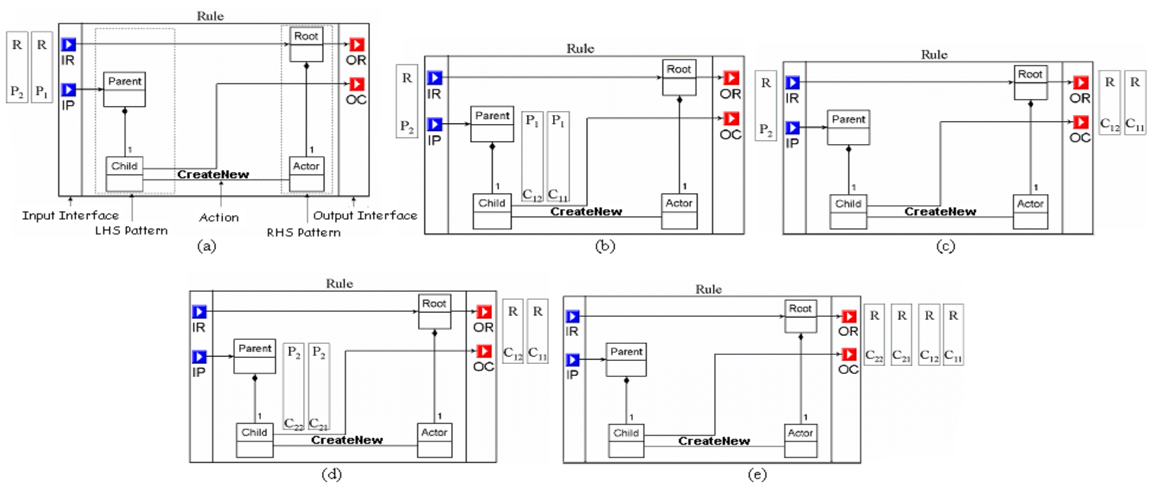


Figure 2: Rule execution in GReAT

- *Match condition:* a flag determining whether *all matches* are executed, or any one, non-deterministically chosen match is executed.

Figure 1 shows a simple transformation rule in GReAT, with a simple pattern. The objects, “House” and “PurchaseOrder” are *bound* to the input ports, which means that they are received as input from a rule that has already been executed. These bound objects form the initial context for the rule for subsequent pattern searches, which means that we search for the objects, “Room1”, “Room2”, and “AdjacentTo” inside this “House” object. Once these elements are found, we check the guard condition, “HasDoor”, which contains procedural (C++) code that can access the attributes of objects. If the guard condition is true, then we create a new “OrderItem” object inside the bound object “PurchaseOrder”, and finally we execute the procedural code contained in the “AttributeMapping” block.

It is important to note that in GReAT both the LHS and RHS of a transformation rule are specified together in a composite pattern. The objects marked as “Bound” can be considered as the LHS of the rule, and the objects marked “CreateNew” or “Delete” are the RHS. This is in contrast to some other popular approaches in the field, in which the LHS and RHS are specified separately.

The execution of a production rule assumes that the input interface is bound to some nodes in the graph. In the case of the first, top-level rule, the user selects which elements from the input and output graphs are passed in. These bindings for the input (and later for the output) ports form a *packet*. When a rule fires, the binding is established, and then a pattern matching algorithm finds a matching subgraph within the connected neighborhood of the bound nodes. Frequently many such matches are generated, each match forming a consistent binding for all edges and nodes in the pattern graph of the rule. These matches will generate packets via the output ports for downstream rules as described below.

Figure 2 gives an overview of the steps of a single rule execution. Two input packets arrive in (a). The rule execution begins with the first packet. The rule first binds all incoming objects from

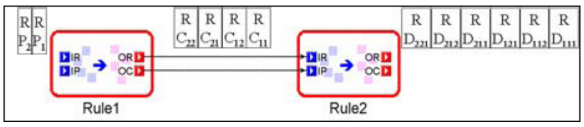


Figure 3: Sequence of rules

the first packet (containing *Root* R and *Parent* P_1). Suppose that in the input model, P_1 contains two children, C_{11} and C_{12}. The intermediate packets produced for these two matches are shown in (b). At this point, a new *Actor* is created for each match. If a guard had been present, only those matches for which the guard condition evaluated to *true* would have been retained. The rule binds the *Root* and the *Child* objects to the output ports. Part (c) shows the output packets generated by this binding of graph objects to output ports. These packets will be sent to the next rule after all the input packets to this rule have been processed. Parts (d) and (e) show the rest of the execution of this rule for the remaining input packet, with part (e) showing the output packets created by this rule.

2.3 Control Flow

We have seen how a single rule executes in GReAT. GReAT allows the construction of larger transformations using the following control flow concepts:

1. *Sequencing* - Rules can be coupled with other rules to execute sequentially. Figure 3 shows two rules in sequence. Rule 1 executes first, consuming all its input packets. Rule 2 fires after Rule 1 has completed producing all its output packets.
2. *Non-determinism* - When rules are not connected sequentially, they can be connected in parallel, and the order of rule execution will be non-deterministic.
3. *Hierarchy* - Rules can be composed hierarchically using *Blocks*. Blocks can contain primitive rules or other blocks. There are two types of blocks in GReAT: *Block* and *ForBlock*. The only difference in their semantics is that in the *ForBlock*, each input packet passes through all the contained rules before the next input packet is passed, while in a *Block* all packets are consumed by the first rule, then the generated packets are passed to the next rule, and so on.
4. *Recursion* - The output of rules can be connected to inputs of blocks higher in the containment hierarchy, which results in the output packets being sent back as input packets to the preceding rules (in effect, recursively activating the rules for a subgraph).
5. *Conditional execution* - GReAT offers special blocks called Test/Case blocks for conditional execution. *Test* blocks can have multiple *Case* blocks, and multiple outputs. The output packets are placed on the output ports determined by the *Case* that matches successfully.

3 Tools

3.1 Modeling Framework

GReAT is not a standalone tool; rather, it is used in conjunction with the Generic Modeling Environment (GME) [Led01]. However, once a transformation has been developed, a standalone executable can be executed outside of GME. The typical modeling and transformation process proceeds as follows.

1. A GReAT transformation project is created in GME, the UML meta-models for the input and the target of the transformation are attached, and the heterogeneous class diagrams created.
2. The transformation rules are specified by drawing patterns of UML classes from elements of the meta-models, and the rules are sequenced to form a transformation program.
3. The transformation is executed. Depending on the required level of performance and maturity of the transformation, this can be done using one of three supplied tools: the GR-Engine, the GR-Debugger, or the code produced by the GR-Code Generator.

3.2 Execution Engines

The GR-Engine provides the fastest way of testing transformation prototypes. The transformation rules are interpreted by the GR-Engine, and thus the level of performance is not as high as with compiled code. The GR-Engine is often used early in the development of a transformation so that results can be tested quickly.

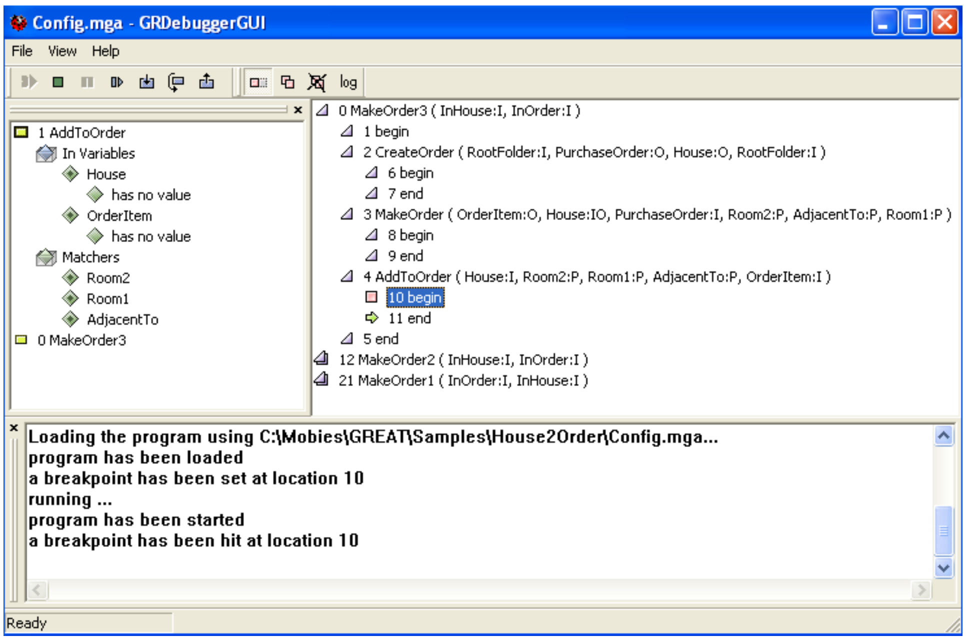
Just as the need for debuggers arises with complex programs written in traditional procedural programming languages, it often becomes very helpful to have the ability to debug graph transformations as well. The GR-Debugger is built on top of the GR-Engine, and offers the typical features found in traditional debuggers, e.g., breakpoints, stepping in/over rules, etc. A graphical window displays a list of the transformation rules, and the user can set breakpoints at any rule and step through the transformation rule by rule, allowing them to see the results of a particular rule or see what elements are being matched at any given time. Figure 4 shows the GR-Debugger.

While both the GR-Engine and GR-Debugger rely on an interpreter to execute the transformation rules, the Code Generator produces the corresponding transformation code in C++ using a model data structure library called UDM [MBL+03]. Our empirical studies have shown that in most cases, this results in a 10x-100x improvement in transformation execution time [VAS04].

4 Comparison

Other graph transformation tools include VIATRA, AGG, Fujaba, and PROGRES. [MGVKar] provides a good comparison and feature summary of the first three, along with GReAT, in the framework of the taxonomy developed in [MCG05].

Fujaba (From UML to Java And Back Again) is geared toward the creation of Java code from UML models in such a way that the UML models can be reverse engineered from the generated Java code by reversing the transformation. Fujaba uses a technique known as *story-driven modeling*, in which an activity diagram specifies the order in which the graph transformation rules



```
1 AddToOrder
    In Variables
        House
            has no value
        OrderItem
            has no value
    Matchers
        Room2
        Room1
        AdjacentTo
0 MakeOrder3
```

```
0 MakeOrder3 ( InHouse:I, InOrder:I )
    1 begin
    2 CreateOrder ( RootFolder:I, PurchaseOrder:O, House:O, RootFolder:I )
        6 begin
        7 end
    3 MakeOrder ( OrderItem:O, House:IO, PurchaseOrder:I, Room2:P, AdjacentTo:P, Room1:P )
        8 begin
        9 end
    4 AddToOrder ( House:I, Room2:P, Room1:P, AdjacentTo:P, OrderItem:I )
        10 begin
        11 end
    5 end
12 MakeOrder2 ( InHouse:I, InOrder:I )
21 MakeOrder1 ( InOrder:I, InHouse:I )
```

```
Loading the program using C:\Mobies\GREAT\Samples\House2Order\Config.mga...
program has been loaded
a breakpoint has been set at location 10
running ...
program has been started
a breakpoint has been hit at location 10
```

Figure 4: GReAT Debugger

should be applied. GReAT can also be used to generate non-trivial code from models [NKS+05]. However, transformations in GReAT are unidirectional, and a reverse transformation must be manually written.

AGG (Attributed Graph Grammar System) is a general purpose graph transformation tool used in high level Java applications. It uses an algebraic approach to graph transformations. In AGG a transformation can be performed only within a single domain, while GReAT allows heterogeneous transformations. AGG also relies on a *graph grammar* approach rather than programmed graph transformations. Using this method, AGG starts with an initial input graph, and then applies all applicable graph productions in parallel. Thus, this approach may not be viable if the user requires explicit control over the rule sequencing.

PROGRES (PROgrammed Graph REwriting Systems) is a another general purpose transformation tool. It is generally used for prototyping process modeling and reengineering tools. PROGRES combines UML-like class diagrams and graph re-write rules which operate on these diagrams. Once a transformation is specified, it can be translated into C and Tcl/Tk code for rapid prototyping. GReAT transformations can be translated into C++ code, but this is mainly done to increase the performance of the transformation.

VIATRA (VIsual Automated model TRAnsformation) is primarily considered a model verification tool that relies on graph transformation techniques, especially in the area of dependability analysis, although it can also be used areas of model-driven development, such as code generation. Transformations in VIATRA are specified in a way similar to GReAT. Graph patterns are defined from a collection of model elements, and then assembled into complex transformations by using abstract state machine rules. Novel features of VIATRA include generic graph transformation rules, in which type parameters can be passed into a rule, making a transformations easy to reuse between domains. Currently, GReAT supports only a limited form of type-parameterization of rules through the use of UML inheritance. For instance, instead of specifying a specific child class in a transformation, one could match the base class, and then filter the specific results using guard conditions, which are written in C++.

Recent additions to GReAT, such as the ability to sort the results of pattern matches, and the distinguished merging of matches (both described in [VNS+05]), are not directly supported in any of the aforementioned tools, although one could possibly implement them with a sequence of rules. We believe that adding these features gives support for quickly executing common tasks, and greatly improves the usability of our tool.

5 Conclusions and Future Work

We have tested the usability and scalability of GReAT with several non-trivial transformations, including a transformation that generates C code from Simulink models [NKS+05], and a transformation that generates real-time E-Code from a Giotto model [Sze05]. In both cases, we were able to express the transformation using the existing features of GReAT, although we also found that additional features would make both transformations easier to define. Thus, we implemented some of these additional features, such as support for sorting the results of pattern matches and support for global objects [VNS+05].

Currently, we are working on implementing novel capabilities, such as the ability to move a (containment) hierarchy of objects contained by one element to be contained by another element. However, this feature requires a precise definition of the semantics of moving an object's connections and children objects, and thus does not have a straightforward solution. The challenge is to decide on an implementation that will make the feature as powerful as possible while still giving the user a high degree of freedom.

In general, we have found that GReAT is a highly usable tool for quickly specifying model transformations. In particular, we believe that our explicit rule sequencing saves the user from dealing with confluence considerations, and our ability to operate on any number of input/output models during a transformation can save the user from having to define multiple transformations [AKL03]. We are continuing our research to determine what constructs and operations should be directly supported by the language so that it is sufficiently expressive, yet still simple enough to be learned and used efficiently.

Bibliography

[AKL03] A. Agrawal, G. Karsai, A. Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Pp. 8–15. ACM Press, New York, NY, USA, 2003.
doi:http://doi.acm.org/10.1145/949344.949347

[EER96] G. Engels, H. Ehrig, G. Rozenberg(eds.). Special Issue on Graph Transformation Systems. In *Fundamenta Informaticae, Vol. 26, No. 3/4, No. 1/2*. 1996.

[Led01] A. Ledeczi. Composing Domain-Specific Design Environments. In *Computer*. 2001.

[MBL+03] E. Magyari, A. Bakay, A. Lang, T. Paka, A. Vizhanyo, A. Agrawal, G. Karsai. UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003*. 2003.

[MCG05] T. Mens, K. Czarnecki, P. V. Gorp. A Taxonomy of Model Transformations. In Bezivin and Heckel (eds.), *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings 04101. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/11> [date of citation: 2005-01-01].

[MGVKar] T. Mens, P. V. Gorp, D. Varro, G. Karsai. *Applying a Model Transformation Taxonomy to Graph Transformation Technology*. (To Appear).

[NKS+05] S. Neema, Z. Kalmar, F. Shi, A. Vizhanyo, G. Karsai. A visually-specified code generator for Simulink/Stateflow. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2005.

[Sze05] T. Szemethy. Case Study: Model Transformations for Time-Triggered Systems. In *International Workshop on Graph and Model Transformations (GRaMoT)*. 2005.

[VAS04] A. Vizhanyo, A. Agrawal, F. Shi. Towards Generation of High-performance Transformations. In *Generative Programming and Component Engineering, Vancouver, Canada*. 2004.

[VNS+05] A. Vizhanyo, S. Neema, F. Shi, D. Balasubramanian, G. Karsai. Improving the Usability of a Graph Transformation Language. In *International Workshop on Graph and Model Transformations (GRaMoT)*. 2005.