A Metamodel Based Model Transformation Language

Aditya Agrawal
Advisor: Dr. Gabor Karsai
Institute for Software Integrated Systems (ISIS)
Vanderbilt University, Nashville, TN – 37235
Email: aditya.agrawal@vanderbilt.edu
ACM membership number: 0414003

ABSTRACT
The Model Driven Architecture (MDA) could have a greater impact by expanding its scope to Domain Specific MDA (DSMDA). DSMDA is the use of the MDA approach for a given domain. For the DSMDA process, transformers are needed to convert Domain Specific Platform Independent Models (DSPIM–s) to Domain Specific Platform Specific Models (DSPDM–s). Such model transformers are time consuming and error prone to develop and maintain. Hence, a high-level specification language to formally specify and possibly analyze model transformers is desirable. The specification language must also have an execution semantics and framework that can be used to execute the transformations. This research addresses these needs and has produced a language and execution framework that considerably improves the development of model transformers.

Categories & Subject Descriptors: D.2.2 Design Tools and Techniques, F.4.2 [Grammars and Other Rewriting Systems] and D.2.6 [Programming Environments]: graphical environments.


Keywords: Model Driven Architecture, Model Integrated Computing, Graph grammars and transformations.

1. MOTIVATION AND PROBLEM STATEMENT
Model Driven Architecture (MDA) [5], a suite of standards that proposes to use high-level models for software development has drawn focus to the aims of Model Integrated Computing (MIC) [1]. MIC is a methodology that advocates the use of high-level domain-specific models for system specification with support for analysis and automated synthesis of an implementation. To leverage the benefits of MIC in MDA, the MDA scope needs to be expanded to Domain Specific MDA where the focus is on developing the MDA process for specific domains. MIC however, has its own shortcomings such as high development cost and lack of standardization [11].

To tackle these problems, we propose a solution that advocates the use of a framework to support the development and use of Domain Specific Modeling Environments (DSME). Such a framework needs to provide DSMDA developers with high-level languages for the specification of the abstract syntax, concrete syntax, visualization, static semantics, dynamic semantics and mappings to analysis and implementation of a particular DSME. The DSME built can then be used to develop Domain Specific Platform Independent Models (DSPIM). These models represent the behavior and structure of the application to be built with no implementation details. Transformations can then be applied based on the mapping specification to convert DSPIMs to their equivalent Domain Specific Platform Specific Models (DSPSM) [11]. DSME tools such as GME [2] and DOME [6] already provide a major portion of the framework support. They allow developers to specify the abstract syntax, concrete syntax, visualization, and static semantics of the modeling environments/languages. However, developers spend significant effort on writing code that implements the transformation from DSPIMs to DSPSMs.

In order to speed up the development of DSMDAs, a high-level language is needed for the specification of model transformers. An execution framework and an “interpreter” can then be used to execute specifications expressed in the language. Design of such a language is non-trivial as a model transformer can work with arbitrarily different domains and can perform fairly complex computations. From a mathematical viewpoint models in MIC are typed, attributed multi-graphs. Thus, we can use the mathematical concepts of graph transformations [7] to formally specify the intended behaviour of a model-to-model transformer.

There exists a variety of graph transformation techniques described in [7][8][9][10] The prominent among these are node replacement grammars, hyperedge replacement grammars, algebraic approaches and programmed graph replacement systems. These techniques have been developed mostly for the specification and recognition of graph languages, and performing transformations within the same domain: a type-graph), while we need a graph transformer that works on (at least) two different graphs. Moreover, these transformation techniques rarely use a formal language for the specification of structural and semantic constraints on the graphs. In summary, the following features are desired in a model-to-model transformation language:
1. The language should provide the user with a way to specify the different graph domains being used. This helps to ensure that graphs/models of a particular domain do not violate the syntax and static semantics of that domain.

2. There should be support for transformations that create independent graphs/models conforming to different domains than the input models/graphs. In the more general case there can be $n$ input model/domain pairs and $m$ output model/domain pairs.

3. Have uniform support for both transformations within a graph and for rewriting one graph to produce another completely disjoint graph with its own integrity constraints.

4. The language should have efficient implementations for its control flow constructs. The generated implementation for the model transformer should exhibit acceptable performance, and unbounded search should be avoided, if possible.

The new language should be usable and well-suited for transforming graphical models to low-level implementation. It should drastically shorten the time taken to develop a new transformation tool for a graphical language, allowing a large number of domain-specific high-level graphical languages to be developed and used.

2. RESEARCH HYPOTHESIS & METHODS

Based upon the survey of the area my research hypothesis is: “A Metamodel based transformation language using graph rewriting and transformations with support for multiple input and output graphs (corresponding to different domains) with an efficient implementation is suitable for the specification of model transformers. Such a language should help achieve a speed-up (the order of 2 to 10) in the time to develop model transformers and thus help develop a family of Domain Specific Model Driven Architecture tools.”

In order to validate the hypothesis a new model-to-model transformation language is required that leverages the benefits of graph grammars and transformations and provides the required language constructs for the development of model transformers. Language development is based on gathering requirements of model transformations and then researching how these needs can be fulfilled by formal yet simple constructs. Requirements were gathered by looking at various target applications and by creating a list of evaluation problems. From this list, two evaluation problems have been chosen:

1. Generate a non-hierarchical Finite State Machine (FSM) from a Hierarchical Concurrent State Machine (HCSM) representation similar to Statecharts. This problem introduces interesting challenges. To map concurrent state machines to a single machine there is a need for complex operations that include the Cartesian product of the composed state spaces. Evaluation of this particular transformation requires a depth-first bottom-up approach and will test whether the system can allow different traversal schemes.

2. Generate from a given Simulink/Stateflow [16] model the equivalent Hybrid Automata [17]. This is another non-trivial example as the mapping is dependent on subtle interactions between the state machines and the switched network in the input model. It is not even obvious if the problem can be solved in the most general case. The algorithm used to solve this problem converts a restricted Simulink-Stateflow model to its equivalent hybrid system. This algorithm has some complex steps such as state splitting, reachability analysis and special graph walks that make it another interesting problem to try. [15]

The complexity of the example problems gives confidence that if solutions to these problems can be specified in the new language and efficient code can be generated from such a specification then the language will be expressive enough to be used to solve a large number of non-trivial real world problems.

The next step was to develop language constructs that can solve these challenge problems. Development of language constructs includes its syntax, visualization, semantics and algorithms for its execution. These language constructs should then be evaluated on the basis of expressiveness, generality and efficiency. A few candidate constructs will then be implemented using the execution engine to further evaluate them. A significant part of this research effort focused on implementation algorithms and the execution framework for the language.

3. A LANGUAGE FOR GRAPH REWRITING AND TRANSFORMATIONS

The transformation language we have developed to address the needs described in Section 1 is called Graph Rewriting and Transformation language (GReAT).

This language can be divided into 3 distinct parts.

1. Pattern Specification language.
2. Graph transformation language.
3. Control flow language.

Before describing the language, we discuss how this language addresses the first three requirements mentioned Section 1.

3.1 Heterogeneous Graph Transformations

The first requirement of the transformation language was the specification of input and output graph domains. Many approaches have been introduced in the literature to capture graph domains. For instance, schemas are used in PROGRES while AGG uses type graphs. These are proprietary formats for the specification of the graph
domain. We chose UML [3] class diagrams and the Object Constraint Language (OCL) [4] for the specification of domains because it is standardized and it is more expressive than both schema and type graphs.

In model-to-model transformations the input and output graphs are object networks whose “schema” can be represented using UML [3] class diagrams and OCL [4]. Thus, the UML class diagram plays the role of a graph grammar such that it can describe all the “legal” object networks that can be constructed within the domain.

From the UML class diagrams one can generate an object oriented API that can be used to implement the graphs, to traverse the input graph, and to construct the output graph. To satisfy the second requirement, GReAT allows the user to specify any number of domains that can be used for the transformation purposes. For example, Figure 1 shows a UML class diagram that represents the domain of Hierarchical Concurrent State Machines (HCSM) and Figure 2 shows the metamodel of a Finite State Machine (FSM).

The third requirement was to provide both rewriting and transformations in the same language framework. This problem is tackled in GReAT by allowing the user to compose source and target metamodels by defining temporary vertex and edge types that can span across multiple domains and will be used temporarily during the transformation. For example, Figure 3 shows a metamodel that defines edges between HCSM and FSM. 

FiniteState and FiniteTransition are classes from Figure 2. This metamodel defines three types of edges. There is a refersTo edge type that can exist between State and FiniteState, and between Transition and FiniteTransition. Another edge type associatedWith is defined and it can link State objects.

By composing the domains using temporary cross-links we are able to tie the different domains together to make a larger, heterogeneous domain that encompasses all the domains and cross-references. This helps us to have a uniform representation for graph transformations and graph rewriting.

### 3.2 The Pattern Specification Language

The pattern specifications found in graph grammars and transformation languages [7][8][9][10] are not sufficient for our purposes. A more expressive, easy-to-use pattern language has been developed that allows specification of complex graph patterns.

The pattern specification language was developed to extend simple patterns with a notion of cardinality on each pattern vertex and each edge. Precise semantics for such a language was developed along with efficient pattern matching algorithms. For a complete discussion on semantics and expressiveness and matching algorithms of pattern graphs please refer to [12].

### 3.3 Graph transformation language

The heart of GReAT is the graph transformation language. It was inspired by many previous efforts such [7][8][9][10]. It defines the basic transformation entity: a production/rule. A production contains a pattern graph. These pattern objects each conform to a type: class or association from the metamodel. Apart from this, each pattern object has another attribute that specifies the role it plays in the transformation. There are three roles that a pattern object can play:

**bind**: The object is used only to match objects in the graph.
delete: The object is used to match objects, but once the match is computed, the objects are deleted.

new: New objects are created after the match is computed. The execution of a rule involves matching every pattern object marked either bind or delete. If the pattern matcher is successful in finding matches for the pattern, then for each match the pattern objects marked delete are deleted and then the objects marked new are created. Since the pattern matcher returns all matches for the pattern, there can be a case where a host graph object is deleted from a match while the next match still has a binding for it. The delete operation checks for such a situation and if it arises it doesn’t perform the delete and returns failure. Thus only those objects can be deleted that are bound exactly once across all the matches.

Pre-conditions are often required for additional constraints on the transformation application. In GReAT, OCL is used for the pre-condition specification and these constraints are evaluated on the matches before the actions are applied. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing objects. “Attribute mapping” is a specification of such attribute manipulation and is executed after the transformation is applied.

The formal definition of a production is as follows. A production P is a 4-tuple: (pattern graph, pattern role, guard, attribute mapping), where

1. Pattern graph is a graph (defined in Section 3.2).
2. Pattern Role is a mapping for each pattern vertex/edge to an element of role = {bind, delete, new}.
3. Guard is a Boolean expression that operates on the vertex and edge attributes. If the guard is false, then the production will not execute any operations.
4. Attribute mapping is a set of assignment statements that specify values for attributes and can use values of other edge and vertex attributes.

Figure 4 shows an example rule. Each object in the pattern graph refers to a class in the heterogeneous metamodel. The semantic meaning of this reference is that the pattern object should match with a graph object that is an instance of the class represented by the metamodel entity. The default action of the pattern objects is bind. The new action is denoted by a tick mark on the pattern vertex (see the vertex StateNew in figure). Delete is represented using a cross mark (not shown in figure). The In and Out icons in the figure are used for passing graph objects between rules and will be discussed in detail in the next section.

3.4 Controlled Graph Rewriting and Transformation

GReAT has a high-level control flow language built on top of the graph transformation language with the following constructs for improving the efficiency of the transformations: (1) pivoting and (2) sequencing. In this paper these issues will be briefly touched upon. For a complete discussion on the efficiency issues related to graph transformations please refer to [13].

The performance of the pattern matching can be significantly increased if some of the pattern variables are bound to elements of the host graph before the matching algorithm is started (effectively providing a context for the search). The initial matches are provided to a transformation rule with the help of ports that form the input and output interface for each transformation step. Thus a transformation rule is similar to a function, which is applied to the set of bindings received through the input ports and results in a set of bindings over the output ports. For a transformation to be executed graph objects must be supplied to each port in the input interface. In Figure 4 the In and Out icons are input and output ports respectively. Input ports provide the initial match to the pattern matcher while output ports are used to extract graph objects from the rule so that they can be passed along to the next rule. The rules thus operate on packets, which are defined as sets of (port, host graph object) pairs.

Explicit sequencing of rules and a high-level control flow language allows the precise control of transformations and thus helps to manage the complexity of the transformation and allows users to write efficient transformations. The control flow language supports the following features:

- Sequencing: rules can be sequenced to fire one after another.
- Non-Determinism: rules can be executed “in parallel”, where the order of firing of the parallel rules is non-deterministic.
- Hierarchy: Compound rules can contain other compound rules or primitive rules.
- Recursion: A level rule can (directly or indirectly) call itself.
- Test/Case: A branching construct can be used to choose between different control flow paths.

4. KEY CONTRIBUTIONS

In Section 3, GReAT, a graph rewriting and transformation languages has been presented. The key contributions of this research are: (1) A unified paradigm for the specification of
both transformations and rewritings. (2) Development of variable cardinality patterns with precise semantics and efficient matching algorithms. (3) Techniques for transformation optimization – such as pivoting and rule sequencing. (4) A high-level graphical control flow languages for the specification of algorithmic transformations.

5. RESULTS AND CONCLUSIONS

To date, I have developed a language that satisfies the requirements mentioned in Section 1. The GReAT language is Turing complete and has simple programming constructs. A formal specification of the GReAT semantics has been described using the Object-Z notation [14]. The execution engine GReAT-E has also been developed and can execute the transformations expressed in GReAT. The language has been successfully used to implement the both the challenge problems. Table 1 shows a comparison of specifications in GReAT vs. (estimated) hand code. The primitive rules are rules that contain graph transformations while compound rules are higher-level control flow constructs. The comparison gives an approximate measure of how the transformations relate to hand code. However, better experiments are required to provide concrete results.

Table 1: Comparison of GReAT specification VS hand code

<table>
<thead>
<tr>
<th>Problems</th>
<th>GReAT</th>
<th>Hand code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark and sweep algorithm on Finite State Machine (FSM)</td>
<td>7/2</td>
<td>~2</td>
</tr>
<tr>
<td>Hierarchical Data Flow (HDF) to Flat Data Flow (FDF)</td>
<td>11/3</td>
<td>~3</td>
</tr>
<tr>
<td>Hierarchical Concurrent State Machine (HCSM) to Finite State Machine (FSM)</td>
<td>21/5</td>
<td>~8</td>
</tr>
<tr>
<td>Simulink Stateflow to C code</td>
<td>70/50</td>
<td>~25</td>
</tr>
<tr>
<td>Matlab Simulink/ Stateflow to Hybrid System [15]</td>
<td>154/43</td>
<td>~50</td>
</tr>
</tbody>
</table>

The first public release of GReAT was in November 2003 and since then there have been two other releases. It is being used by students and researchers from other universities and from the industry. To date, GReAT has been downloaded by over 100 researchers from more than 10 different countries. Researchers at Daimler Chrysler are using GReAT to generate automated test cases to verify correctness of the transformations.

6. ACKNOWLEDGEMENTS

The DARPA/IXO MOBIES program, Air Force Research Laboratory under agreement number F30602-00-1-0580 and NSF ITR on “Foundations of Hybrid and Embedded Software Systems” programs have supported, in part, the activities described in this paper.

7. REFERENCES


