Graph Transformations on Domain-Specific Models

ADITYA AGRAWAL, GABOR KARSAI, FENG SHI

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

Email: {aditya.agrawal, gabor.karsai, feng.shi}@vanderbilt.edu

ABSTRACT. In model driven development, model transformations play a crucial role. This paper introduces a new, UML-based approach for the specification and implementation of model transformations. The technique is based on graph transformations, where the transformations are a set of explicitly sequenced elementary rewriting operations. UML class diagrams are used to represent the graph grammars for the input and output graphs. The paper's main contributions are the visual language designed for the representation of transformation programs and the graph transformation execution engine which implements the semantics of the language.

Keywords. *Model transformation, UML, graph transformation, graph rewriting, Model Driven Architecture.*

1. Introduction

The model driven development of systems [4] necessitates the transformation of models into other models (e.g. analysis models) and artifacts (e.g. executable code) relevant in the system development process. Writing complex transformations is not easy, and tools are needed. Graph grammars and graph transformations (GGT) have been recognized as a powerful technique for specifying complex transformations. They can be used in various situations in a software development process [13][14][15][16]. Many tasks in software development have be formulated using this approach, including weaving of aspect-oriented programs [24], application of design patterns [15], and the transformation of platform-independent models into platform specific models [6]. A special class of transformations arises in Model Integrated Computing (MIC) [1]. MIC is an approach in which a domain-specific modeling language and generator tools are developed and then the domain-specific language is used for creating and evolving the system (or a product-line of systems) through modeling and generation. During the last decade, MIC has gained acceptance through

various fielded systems [25][26], and it is recognized in both academia and industry today. In the MIC approach, a crucial point is the generation, where design time models are transformed into executable models and analysis models. Executable models are used to configure a run-time platform (e.g. a component framework), while analysis models are used to verify the system using simulation and various other verification techniques. The development of the model transformation tools is the cornerstone of MIC: the model transformation tools (also called model interpreters) establish a bridge between the domain-specific models and their execution-time and analysis-time equivalents.

In this paper we propose to use GGT techniques to provide an infrastructure for model transformations. We will use the MIC software process as the context, in which we present our results, but they easily generalize to universal model transformations like the ones advocated in OMG's Model Driven Architecture [4]. Section 2 briefly introduces Model Integrated Computing (MIC), and reviews graph grammars and transformations. Section 3 describes Graph Rewriting and Transformation (GReAT) a language that allows transformations from one domain to another using heterogeneous metamodels. GReAT has a rich pattern specification sub-language, a graph transformation sub-language and a high-level control flow sub-language and has been designed to addressed the specific needs of the model transformation problem. Section 4 provides details of the execution engine that implements GReAT. Section 5 shows an example model transformation using GReAT along with some results. Section 6 discusses the conclusions and proposals for future research.

2. Background and Related Work

2.1. Model Integrated Computing (MIC)

MIC is a software and system development approach that advocates the use of domain-specific models to represent relevant aspects of a system. The models capture system design and are used to synthesize executable systems, perform analysis or "drive" simulations. The advantage of this methodology is that it expedites the design process, supports evolution, eases system maintenance and reduces costs [1].

The MIC development cycle (see Figure 1) starts with the formal specification of a new application domain. The specification proceeds by identifying the domain concepts, their attributes, and relationships among them through a process called metamodeling [1]. Metamodeling is enacted through the creation of metamodels that define the abstract syntax, static semantics and visualization rules of the domain. The visualization rules determine how domain models are to be visualized and manipulated in a visual modeling environment. Once the domain has been defined, the specification; i.e. the metamodel of the domain is used to generate a Domain-specific Design Environment (DSDE) through the step called "Meta-Level Translation". The DSDE can then be used to create domain-specific designs/models; for example, a particular state machine is a domain-specific design that conforms to the rules specified in the metamodel of the state machine domain. However, to do something useful with these models such as to synthesize executable code, perform analysis or drive simulators, we have to convert the models into another format like executable code, input language of analysis tools, or configuration files for simulators. This mapping of the models to another useful form is called model transformation and is performed by model transformers [1]. Model transformers (also called "model interpreters") are programs that convert models in a given domain into models of another domain. For instance, a source model can be in the form of a synchronous dataflow network of signal processing operations, while the target model can be in the form of Petri-nets, suitable for predicting the performance of the network. Note that the result of the transformation can be considered as another model that conforms to a different metamodel: the metamodel of the target [1].

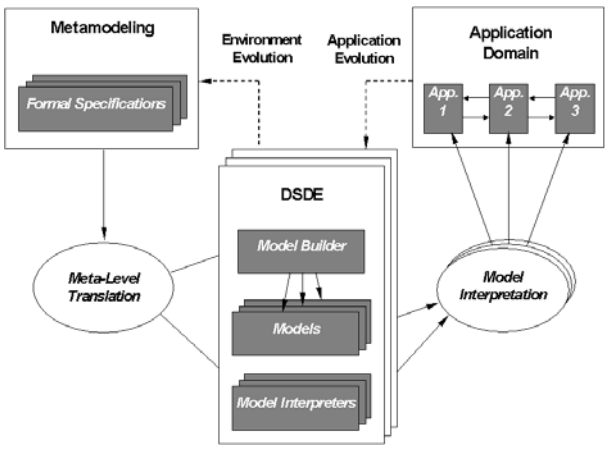


Figure 1 The MIC Development Cycle [2]

MIC promotes a metamodel-based approach to system construction, which has gained acceptance in recent years. The flagship research products following this approach are: Atom^3 [28], DOME [29], Moses [30], and GME [2]. Each implementation has a metamodeling layer that allows the specification of a domain-specific modeling languages and a modeling layer that allows the creation and modification of domain models.

The Generic Modeling Environment (GME) is the main component of the latest generation of MIC technologies developed at the Institute for Software Integrated Systems (ISIS), Vanderbilt University since the late 1980s. GME provides a framework for creating domain-specific modeling environments [2]. An important distinguishing property of the metamodeling environment of GME is that it is based on UML class diagrams [3]; an industry standard, which are used to describe domain-specific modeling languages and their corresponding modeling environment by capturing the syntax, semantics and visualization rules of the target domain. A tool called the meta-interpreter interprets the metamodels and generates a configuration file for GME. This configuration file acts as a meta-program for the (generic) GME editing engine, so that it makes GME behave like a specialized modeling environment supporting the target domain. Note that the core of GME is used both as the metamodeling environment and the target environment; the metamodeling language is just another domain-specific language that the common editing engine supports.

GME has both a metamodeling environment and metamodel transformer that generates a new modelling environment from the metamodels. However, until now there were no generic tools to automatically generate domain-specific model transformers. Each model transformer was written by hand and this was the most time consuming and error-prone phase of the MIC approach. There was a need to develop methods and tools to automate and speed up the process of creating model transformers.

The MIC approach described above is gaining a lot of attention recently with the advent of the Model Driven Architecture (MDA) by Object Management Group (OMG) [4]. MIC can be considered as a particular manifestation of MDA, which is tailored towards system construction via domain-specific modeling languages [6].

2.2. Graph Grammars and Transformations

To enhance the development of model transformers we need a way to precisely specify the operation of those transformers on categories of models, and then generate the model transformer code from the specification. However, this task is non-trivial as a model transformer can be required to work with two arbitrarily different domains and perform fairly complex computations. Hence, the specification language needs to be powerful enough to cover diverse needs and yet be simple and usable.

Note that the metamodels, which are UML class diagrams, define the abstract syntax of the visual modeling language. In fact, GME allows the creation and manipulation of only such object structures that are compliant with those UML class diagrams. The objects edited in GME are called models, and the metamodels determine how model objects are composed, what attributes they have, what static semantic constraints are imposed on them, etc.

From a mathematical viewpoint models in MIC are graphs, to be more precise: vertex and edge labelled multi-graphs, where the labels are denoting the corresponding entities (i.e. types) in the metamodel. It is plausible to formulate the model transformation problem as a graph transformation problem. We can then use the mathematical concepts of graph transformations to formally specify the intended behaviour of a model transformer.

A variety of graph transformation techniques are described in[7][8][9][10][11][12][19]. These techniques include node replacement grammars, hyperedge replacement grammars, algebraic approaches, and programmed graph replacement systems. Graph grammar techniques such as node replacement grammars, hyper edge replacement grammars, and algebraic approaches such as the ones used in AGG do not provide sufficiently rich mechanisms for controlling the application of transformation rules. PROGRES has a rich set of control mechanisms, however, they perform transformations within the same domain. Domains specify the structural integrity constraints that the graphs must conform to; in PROGRES these constraints are represented using schemas [7], while in AGG these are represented using type graphs [27].

In MIC, the domain is represented by a metamodel, and the model transformations typically transform graphs that conform to one metamodel to models that conform to a completely different metamodel. For example, a model transformer may be

required to convert models/graphs belonging to the "state machine" domain to models/graphs conforming to the "flow chart" domain. The graph transformation system must provide support for these transformations across heterogeneous domains. There is yet another problem: maintaining references between the different models/graphs. During the transformations it is usually required to link graph objects belonging to different domains.

To illustrate the point let us consider a very simple transformation that needs to transform models conforming to one domain to another. For sake of simplicity we consider that the source domain has only one type on vertices `V1` and only one type of edges `E1` and that the target domain has again only one type of vertices `V2` and only one type of edges `E2`. The transformation's aim is to create a vertex and edge in the target set for each vertex and edge in the source set:

$$\forall e1 \ni E1 \Rightarrow \exists_1 e2 \ni E2, \forall v2 \ni V1 \Rightarrow \exists_1 v2 \ni V2$$

(where $\exists_1$ means "precisely one"). A simple algorithm could first create a target vertex for each source vertex and then create the edges. To create a target edge `e2` that corresponds to source edge `e1` we need to find the vertices in the target that correspond to the two source vertices `e1` is incident with. This information needs to be saved in the first phase of the transformation for use in the second phase, and can be considered as maintaining reference between two graphs. There are other examples where the referencing is not that easy, for example, in a transformation that determines the cross product of two sets of vertices to generate a new set of vertices. In this case each pair of source vertices should reference a single target vertex. A method is required to specify and use this information.

The existing GGT approaches are powerful but not well suited for the specification and implementation of model transformers as described. Hence, a new approach that targets the specific needs of model-to-model transformation is required. The new approach should have the following features:

- 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 the domain.
- 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.

- Cross-references between graph domains should be supported through well-formed, preferably graphical language constructs.
- 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.
- All the previous points aim at increasing programmer productivity in writing model transformers, thus the language should be usable by software engineers with average experience. This is the primary goal.

The new language should be usable and suited for addressing the needs of 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.

Many papers in recent times have shown how graph transformation techniques can be used for (1) specification of program transformations [13], (2) defining the semantics of a hierarchical state machines [14], (3) supporting design patterns [15] and (4) tool integration [16]. The new language should be able to implement the ideas presented in these papers.

3. A Language for Graph Rewriting and Transformations

The transformation language we have developed to address the needs discussed above is called Graph Rewriting and Transformation language (GReAT).

This language can be divided into 3 distinct parts.

- Pattern Specification language.
- Graph transformation language.
- Control flow language.

Before describing the language, we discuss how this language addresses the first three requirements mentioned in Section 2.2.

3.1. Heterogeneous Graph Transformations

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 approaches are specific to the particular systems, while standards like UML are widely used in the software community today, and we have chosen to follow the UML route. It was also a pragmatic decision, as UML was used in our tools already.

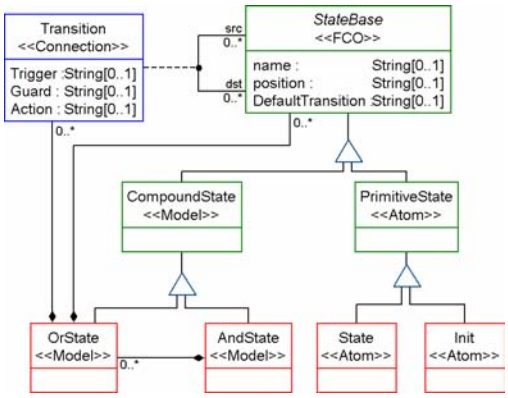


Figure 2 Metamodel of Hierarchical Concurrent State machine using UML class diagrams

In model-to-model transformations the input and output graphs are object networks whose "schema" can be represented using UML class diagrams and expressions in the Object Constraint Language (OCL) [21]. UML provides a rich language to specify structural constraints while OCL can be used to specify non-structural, semantic constraints. Thus, a UML class diagram can play the role of a graph grammar in that it can describe all the "legal" object networks that can be constructed with the domain. Finally, UML can be used to generate an object oriented API that can be used to traverse the input graph and to generate the output graph. GReAT allows the user to specify any number of domains that can be used for the transformation purposes. Figure 3 shows a UML class diagram that represents the domain of Hierarchical Concurrent State Machines (HCSM) and Figure 3 shows the metamodel of a Finite State Machine (FSM).

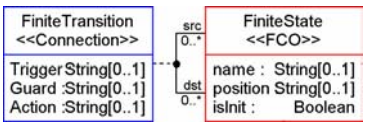


Figure 3 Metamodel of a simple finite state machine

As discussed above, one problem that we need to address is how to maintain links between objects across multiple domains, such that these links appear as first-class elements (i.e., edges) in the graph transformation process. This problem is tackled in GReAT by using an additional domain to represent all the cross-domain links. Apart from using UML to specify all the different domains that will be used for the transformation, UML is also used to specify a temporary domain that contains the information of all the types of cross-links the transformation needs to know about. For example, Figure 1 shows a metamodel that defines associations/edges between HCSM and FSM. The *State* and *Transition* are classes from Figure 2 while the *FiniteState* and *FiniteTransition* are classes from Figure 3. 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 exist between *State* objects.

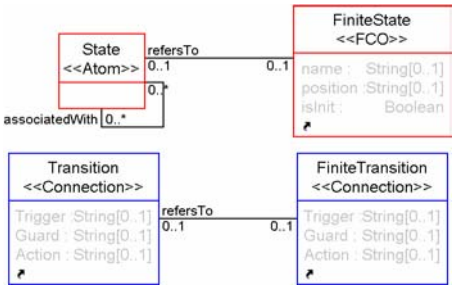


Figure 4 A metamodel that introduces cross-links

Cross-links can be defined not only between different domains but can also be used to extend a domain to provide some extra functionality required by the transformation. By using yet another domain to specify the 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 also helps us to have the same representation for cross-links as for any other edges.

3.2. Definitions

Before describing GReAT, some initial definitions are presented in this section. Graphs used in the GReAT language are typed and attributed multi-graphs and are defined below.

3.2.1. Vertex

A vertex V is a pair: (class, attrs), where class is a UML class, and attrs is a map that maps each defined attribute of the class into a value.

3.2.2. Edge

An edge E is a 3-tuple (etype, src, dst), where etype is the association the edge belongs to. In UML, simple associations are distinguished by their endpoint classes. This information can be considered as an "edge type". The association classes in UML can also be distinguished using two edges: one from the source class to the association class and another one from the association class to the destination class. Src and dst are the vertices that the edge is incident upon. The class of these vertices must be identical to the endpoint classes of etype.

3.2.3. Graph

A graph G is pair (GV, GE), Where GV is a set of vertices in the graph and GE is the set of edges and $\forall e = (etype, src, dst) \in GE, src \in GV \wedge dst \in GV$.

3.2.4. Match

A match M is a pair (MVB, MEB), where MVB is a set of vertex bindings and MEB is a set of edge bindings. Vertex binding is defined as a pair (PV, HV), where PV is a pattern vertex and HV is a host graph vertex. Similarly, edge binding is a pair (PE, HE), where PE is a pattern edge and HE is a host edge. The match must satisfy the following property.

$\forall EB \in MEB$, where
$EB = (pe, he), pe = (pet, psrc, pdst), he = (het, hsrc, hdst)$
$\exists VBS \in MVB \wedge VBD \in MVB$
$\because VBS = (psrc, hsrc) \wedge VBD = (pdst, hdst)$

The match doesn't have a restriction that specifies that each pattern object must have a binding. This is intentional, as the match is also used to specify partial matching of pattern graphs.

3.3. The Pattern Specification Language

A full graph transformation language is built upon a graph pattern specification language and pattern matching. Graph patterns allow selecting portions of the

input (host) graph, and thus specify the scope of individual transformation steps. The specification techniques found in graph grammars and transformation languages [7][8][9][10][17][18][19][20] were not sufficient for our purposes, as they did not follow UML concepts. This paper introduces an expressive yet easy to use pattern specification language, which is closely related to UML class diagrams.

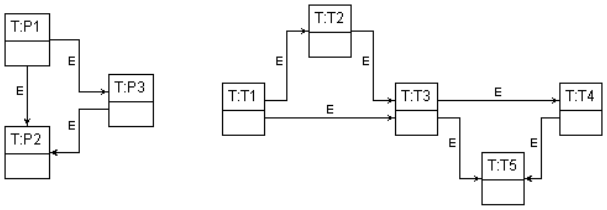
Recall that the goal of the pattern language is to specify patterns over graphs (of objects and links), where the vertices and edges belong to specific classes and associations. In the language we will rely on the assumption that a UML class diagram is available for the objects. The UML class diagram can be considered as the "graph grammar," which specifies all legal constructs formed over the objects that are instances of classes introduced in the class diagram.

3.3.1. Simple Patterns

A simple pattern is one in which the pattern represents the exact sub graph. For example, if we were looking for a clique of size three in a graph, we would draw up the clique as the pattern specification. These patterns can be alternatively called single cardinality patterns, as each vertex drawn in the pattern specification needs to match exactly one vertex in the host graph.

These patterns are straightforward to specify; however, ensuring determinism on such graphs is not. In this case determinism means that given a graph and pattern the match returned should be the same from one execution of the pattern matcher to another and from one matching algorithm to another. Pattern matching in graphs is non-deterministic and different matching algorithms may yield different results.

Consider the example in Figure 5(a). The figure describes a pattern that has three vertices P1, P2 and P3, each of type T. The pattern can match with the host graph shown in Figure 5(b) to return two valid matches, {(P1,T1), (P2,T3), (P3,T2)} and {(P1,T3), (P2,T5), (P3,T4)}. For sake of brevity matches are considered as a set of vertex bindings, edge bindings have been ignored as they can be inferred from the vertex bindings. We see that the result of the matching depends upon the starting point of the search and the exact implementation of the algorithm.



(a) Pattern (b) Host graph

Figure 5 Non-determinism in matching a simple pattern

The solution for this problem is to return the set of all the valid matches for a given pattern. The set of matches will always be the same for a given pattern and host graph.

Returning all the matches however, has a time complexity of $O(C_h^{C_p})$, where C_h is the number of host vertices and C_p is the number pattern vertices. To make the pattern matching usable we need to optimize it. One approach is to start the pattern matcher with an initial context. By context we mean that the pattern matcher is started with an initial partial match. For example, in Figure 5 the pattern matcher could be started with a binding {(T1,P1)} thus, the context for the matching is the host vertex T1 and the matcher will return only one match {(P1,T1), (P2,T3), (P3,T2)}. The initial binding reduces the search complexity in two ways, (1) the exponential is reduced to only the unmatched pattern vertices and (2) only host graph elements within a distance d from the bound vertex are used for the search, where d is the longest pattern path from the bound pattern vertex.

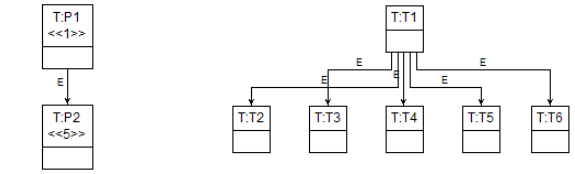
An algorithm for matching such kinds of patterns is given in Appendix 1. The algorithm takes as input the pattern, host graph and a partial match and returns a set of matches. The partial match must have at least one vertex of the pattern bound to the host graph. It uses a recursive approach to solving the matching problem and returns a set of matches.

There are cases where we would like to use the pattern matcher on the entire graph and not restrict it to any context. This can be achieved by running the pattern-matching algorithm for each host vertex.

3.3.2. Fixed Cardinality Patterns

Suppose we need to specify a string pattern that starts with an ‘s’ and is followed by 5 ‘o’-s. Obviously we could enumerate the ‘o’s and write “sooooo”. However,

this is not a scalable solution and thus a representation format is required to specify such strings in a concise and scalable manner. For strings we could write it as “s5o” and use the semantic meaning that o needs to be enumerated 5 times assuming that ‘5’ is not part of the alphabet of this particular language.



(a) Pattern (b) The graph it will match

Figure 6 Pattern specification with cardinality

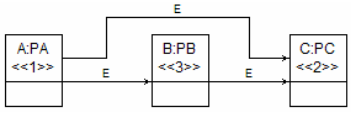
The same argument holds for graphs, and a similar technique can be used. The pattern vertex definition can be changed to a pair (class, cardinality), where cardinality is an integer. Vertex binding can also be redefined as a pair (PV, HVS), where PV is a pattern vertex and HVS is a set of host vertices. For example, Figure 6(a) shows a pattern with cardinality on vertices. The pattern vertex cardinality is specified in angular brackets and a pattern vertex must match n host graph vertices where n is its cardinality. In this case the match is {(P1,T1), (P2,{T2, T3, T4, T5, T6})}.

The fixed cardinality pattern and matching also have non-determinism. Even in this case the issue can be dealt with by returning all the possible matches. If all the possible matches are returned there is a problem of returning a large number of matches. For example in Figure 6, if the host graph contained another vertex T7 adjacent to T1 then the number of matches returned would be ${}^{6}C_{5}$ (all combinations of 5 vertices out of 6). Thus 6 matches will be returned and each having only one vertex different from the other.

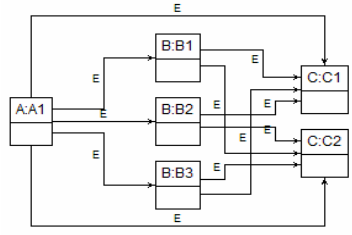
A more immediate concern is how this notion of cardinality truly extends to graphs. In strings, we have the advantage of a strict ordering from left to right, while graphs don’t. By just extending the example in Figure 6 with another pattern vertex we see that the specification is ambiguous.

In Figure 7(a) we show a pattern having three vertices. There are different semantics that can be associated with the pattern. One possible semantic is to consider each pattern vertex pv to have a set of matches equalling the cardinality of the vertex. Then an edge between two pattern vertices pv1 & pv2, implies that in a match each v1, v2 pair are adjacent, where v1 is bound to pv1 and v2 is

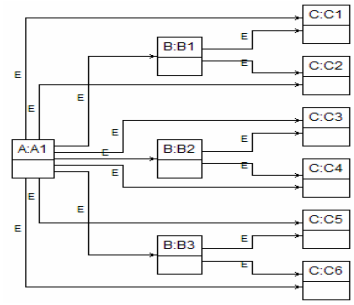
bound to pv2. This semantic when applied to the pattern in Figure 7(a) gives the graph in Figure 7(b).



(a) Pattern with three vertices



(b) Set semantics



(c) Tree semantics

Figure 7 Pattern with different semantic meanings

The algorithm to search the host graph for a set of matches according to the above-mentioned semantics is given in Appendix 2. The algorithm is a direct extension of the algorithm discussed in 3.3.1.

The set semantics will always return a match of the structure shown in Figure 7(b), and it doesn't depend upon the factors like the starting point of the search and how the search is conducted. However, with the set semantics it is not obvious how to represent a pattern to match the graph shown in Figure 7(c).

Another possible semantics could be the tree semantics: If a pattern vertex pv1 with cardinality c1 is adjacent to pattern vertex pv2 with cardinality c2, then the semantics is, each vertex bound to v1 will be adjacent to c2 vertices bound to v2. Let b1 = (pv1,V1) and b2 = (pv2,V2) be the bindings for pv1 and pv2 respectively. Then

$$\forall v_1 \in V1 \overset{c2}{\underset{n=1}{\exists}} v_{2n} \in V2, \wedge e(v_1, v_{2n})$$ **Relation 1**

This semantics when applied to the pattern gives Figure 7(c). The tree semantic is weak in the sense that it will yield different results for different traversals of the pattern vertices and edges. For the traversal sequence pa, pb, pc we get a the graph shown if Figure 7(c) while for the traversal sequence pa, pc, pb we will get a different graph as shown in Figure 8. Another problem with the tree semantics is that graphs like the one shown in Figure 7(b) cannot be expressed in a concise manner.

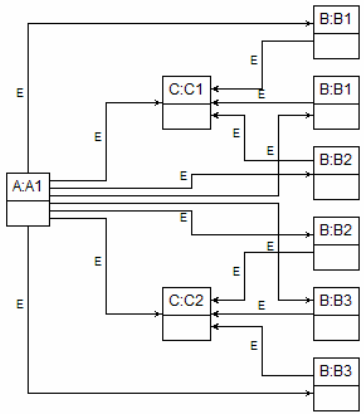


Figure 8 Conflicting match for the tree semantics

Both semantics discussed so far are incomplete in the sense that certain pattern matches cannot be expressed with it. Choosing either one compromises the expressiveness of the language. Furthermore, the tree semantics also brings in a different form of non-determinism because different traversal sequences yield different results.

Fortunately, there is a pragmatic solution that solves all the problems: to use a more expressive, extended set notation.

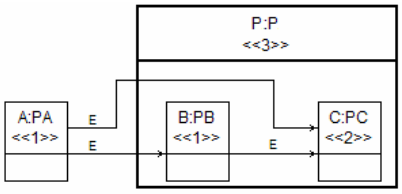
3.3.3. Extending the Set Semantics

For example, if we want to specify a string “sxyxyxy”, we see that “xy” is repeated 3 times. Extending the notation used before we would express it as “s3(xy)”. Using parenthesis we were able to represent the fact that the “xy” sequence should occur 3 times. A similar notion can be used in graphs as well. That is, to use the notion of grouping vertices of a pattern to form a sub pattern and then a larger pattern can be constructed using these sub patterns as vertices. If

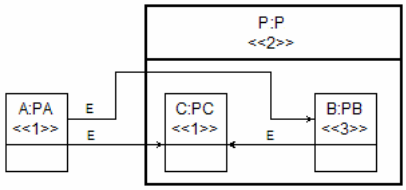
a group consists of a sub graph and has the cardinality n then the n sub graph need to be found. Another important point here is that while in strings the ordering of each element of the group is implicit in graphs we have to specify the connectivity and thus edges can be specified across groups.

To illustrate the point Figure 9(a) shows the pattern that would express the graph in Figure 7(c) and Figure 9(b) shows the graph the expresses the graph in Figure 8. With respect to the pattern P in Figure 9(a) there will be exactly one vertex PB that will connect to exactly 2 vertices of type PC. The larger pattern will consist of the 3 sub patterns of the type described by P. The resulting graph that will be matched is shown in Figure 7(c).

The above exercise illustrated two points. First, the set semantics along with the grouping notion can express all the graphs that the tree semantics can express and the second point is that the semantics are still precise and map to exactly one graph.



(a) Pattern for Figure 7(c)



(b) Pattern for Figure 8

Figure 9 Hierarchical patterns using set semantics

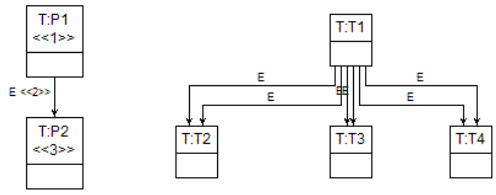
At this point it is apparent that we can express a variety of graphs in an intuitive, concise and precise way. However, a large number of graphs are missing from the Grouped Set Semantics (GSS) that we described above: these graphs are those having more than one edge for the same pair of vertices.

3.3.4. Cardinality For Edges

Adding cardinality to pattern edges helps us express additional graph patterns in a compact manner. Another example is called for and is shown in Figure 10. The figure shows a pattern with cardinality on the edge. The semantic meaning is an extension of Relation 1. Let b1=(V1,pv1) and b2 defined as

$$\forall v1 \in V1, v2 \in V2, \exists_{n=1}^{C} e_n(v1, v2) \qquad \textbf{Relation 2}$$

The extension is that instead of having one edge between each pair of vertices there can be C edges where C is the cardinality of the pattern edge.



(a) Pattern (b) Matching Host graph

Figure 10 Pattern with cardinality on edge.

3.3.5. Variable Cardinality

Sometimes, the sub graph to be matched is not fixed but part of a family of graphs. To show an analogy, suppose we want to match a string starting with 's' followed by 1 or more 'b's. Therefore, the pattern specification represents a family of strings. This can be expressed with the help of regular expressions, such as "s(b)+". In the general case the number of 'b's can be bound by two numbers, the lower and upper bound. To extend the example let us consider that 5 to 10 'b's could follow the 's'. By extending the regular expression notation slightly, we can come up with a notation "s(5..10)(b)".

Using a similar method for graphs, we can allow the notation of cardinality to be variable of the form (x..y), where the lower bound is x and the upper bound is y. Hence a particular pattern vertex should match at least x host graph vertices and not more that y host graph vertices. The upper bound can however be *, representing no limit. This approach can also be used to specify optional components in a pattern by having the cardinality of optional components as (0..1).

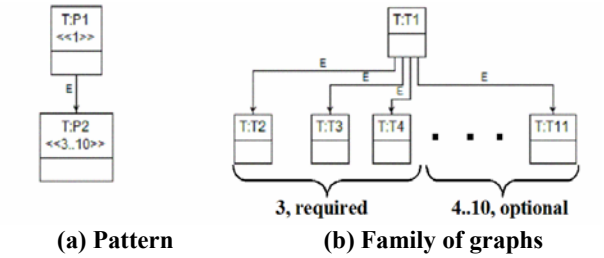


Figure 11 Variable cardinality pattern and family of graphs

In Figure 11 we see a variable cardinality example. The pattern in Figure 11(a) specifies that 3..10 P2s can be connected to a P1, thus the family of graphs represented is given in Figure 11(b). The required portion must be present while the optional part may or may not be present. We have finally extended the specification language to express a truly large set of graphs.

However, there are a few problems with variable cardinality. Let us consider the pattern in Figure 11(a) and let us say that we have a graph having T2..T11 connected to T1 in the host graph. Should the pattern-matching algorithm return only one match namely the entire host graph or all possible sub graphs with cardinality 3, 4 till cardinality 10. The way we answer this question is that if more than one match occurs; then both the matches will be returned if and only if neither match is a proper sub set of the other. Thus the matches returned would each be maximal and consistent with respect to the pattern.

$$\forall m1, m2 \in M, m1 \not\subset m2 \wedge m2 \not\subset m1 \qquad \textbf{Relation 3}$$

Relation 3 states that from the returned set of matches there should not be any two matches such that one is the subset of the other.

This construction yields a precise and consistent language, which can be used to specify complex patterns in a concise manner.

3.3.6. Pattern Graph and Match Definition

After the discussion on the specification of patterns we can now define pattern vertices, edges and graphs.

A pattern vertex PV is a pair: (class, cardinality), where class is a UML class defined in the heterogeneous metamodel and cardinality is a pair (lower bound, upper bound). A pattern edge PE is a 4-tuple (etype, src, dst, cardinality), where etype is the association the edge belongs to. Src and dst are the pattern vertices that the edge is incident upon. The class of these vertices must be identical to the

endpoint classes of etype. A pattern graph PG is pair (GPV, GPE), Where GPV is a set of vertices in the graph and GPE is the set of edges and

$\forall pe = (etype, src, dst, c) \in GPE, src \in GPV \wedge dst \in GPV$.

The definition of a match can also be suitably revised to a pair (MVB, MEB), where MVB is a set of vertex bindings and MEB is a set of edge bindings. Vertex binding is defined as a pair (PV, HV), where PV is a pattern vertex and HV is a set of host graph vertices. Similarly edge binding is a pair (PE, HE), where PE is a pattern edge and HE is a set of host graph edges. The match must satisfy the following properties.

$\forall EB \in MEB$, where $EB = (pe, HE), pe = (pet, psrc, pdst, cardinality)$,
$\forall he \in HE, he = (het, hsrc, hdst), \exists VBS \in MVB \wedge VBD \in MVB$
$\because VBS = (psrc, hsrc) \wedge VBD = (pdst, hdst)$

and

$\forall EB \in MEB$, where
$EB = (pe, HE), pe = (pet, psrc, pdst, (lower, upper))$
$lower \leq C_{HE} \leq upper$
and
$\forall VB \in MVB$, where
$VB = (pv, HV), pv = (pclass, (lower, upper))$
$lower \leq C_{HV} \leq upper$

3.4. Graph Rewriting/Transformation Language

The graph transformation language is inspired by many previous efforts such [9][10][11][19][20]. 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 different roles that a pattern object can play. They are:

*bind***:** The object is used 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***:** After the match is computed, new objects are created.

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.

Sometimes the patterns by themselves are not enough to specify the exact graph parts to match and we need other, non-structural constraints on the pattern. For example, "an integer attribute of a particular vertex should be within a range." These constraints are described using Object Constraint Language (OCL) [21], as it is a widely used standard and is directly related to UML the metamodeling language of GME. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing objects, this done via "attribute mapping".

The formal definition of a production is as follows. A production P is a triple (pattern graph, guard, attribute mapping), where

- *Pattern graph* is a graph (defined in Section 3.3.6).
- *Pattern Role* is a mapping for each pattern vertex/edge to an element of role = {bind, delete, new}.
- *Guard* is a boolean-valued expression that operates on the vertex and edge attributes. If the guard is false, then the production will not execute any operations.
- *Attribute mapping* is a set of assignment statements that specify values for attributes and can use values of other edge and vertex attributes.

Figure 12 describes the algorithm executing a production (a "rule"). The algorithm calls the pattern matcher described in Appendix 1 and 2. A "Packet" provides the initial binding required by the pattern matcher and the "Effector" function performs deletion and creation of objects, described later in the paper.

```
Function Name  : ExecuteRule
Inputs         :   1. Rule rule (rule to execute)
                   2. List of Packets inputs
Outputs        :   1. List of Packets outputs
outputs = ExecuteRule(rule, inputs)
{   List of Packets matches
    List of Packets outputs
    for each input in inputs
    {   matches = PatternMatcher(rule, input)
        for each match in matches
        {   if match doesn't satisfy guard
                matches.Remove(match)
        }
        for each match in matches
```

```
        {   Effector(rule, match)
            outputs.Add(match)
        }
    }
    return outputs
}
```

Figure 12 Algorithm for rule execution

3.4.1. Language Realization

The goal of the language is (1) to transform models that (a) belong to one meta-model into models that belong to another meta-model or (b) to transform models within one meta-model, and (2) to maintain the consistency of the models with respect to their meta-models. Hence, it is important that the language only allows the user to draw patterns that conform to the meta-models.

To maintain consistency and provide usability in GReAT, the following use case is defined. The use case is supported through the services of the modeling environment (GME).

- The user first imports the input and output metamodels of the models to transform in the form of libraries.
- Next, the user specifies another metamodel that defines all the temporary vertices and edges that he/she will need for the transformation.
- After attaching and specifying these metamodels the user can then draw productions/rules that specify patterns.

Figure 13 shows an example rule. The rule contains a pattern graph, a Guard and an AttributeMapping. Each object in the pattern graph refers to a class in the heterogeneous metamodel. The semantic meaning of the 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.

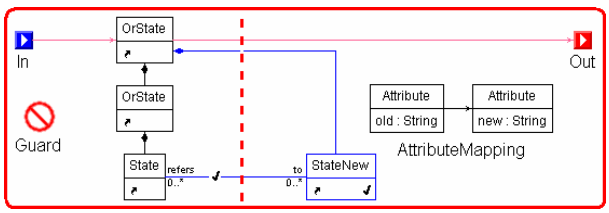


Figure 13 An example rule with patterns, guards and attribute mapping

GReAT relies on UML metamodels for defining patterns. Furthermore, the patterns are also specified in (a superset of the) UML syntax and since the modeler uses UML for metamodeling, as it was more intuitive to describe the rules also in UML. By making the user reference each pattern object we can enforce the consistency of the patterns and thus the consistency of the transformations.

3.5. The Language for Controlled Graph Rewriting and Transformation

In section 3.3.1 concerns about the efficiency of the pattern matching algorithm were discussed. 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 13 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.

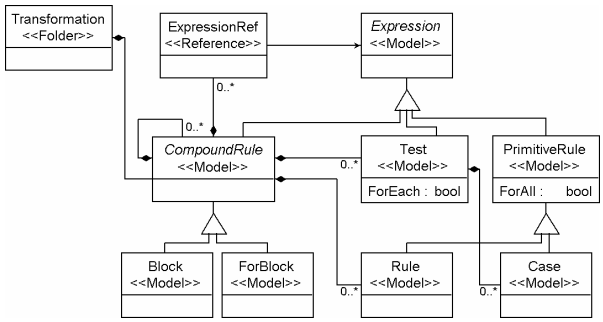


Figure 14: UML class diagram for the abstract syntax classes of GreAT: The core transformation classes

The next concern is the application order of rewriting productions. Classical graph grammars apply any production that is feasible. This, very powerful technique is

good for generating and matching languages but model-to-model transformations often can and need to follow an algorithm that requires a more strict control over the execution sequence of rules, with the additional benefit of making the implementation more efficient.

In order to better manage complexity in transformation programs it is important to have higher-level constructs, like hierarchical rules and control structures in the graph rewriting language. For this reason, we support (1) the nesting of rules and (2) control structures. We show these capabilities here using the classes that form the abstract syntax tree of the language. The common abstract base class for the language is *Expression*, as shown in Figure 14, and all other constructs like *Rules* and *Blocks* are derived from it. The derivation implies a shared base semantics: all these classes represent some kind of graph transformations.

Figure 15 shows input-output interfaces (*Ports*) of the *Expressions* (*In* and *Out*), as well as the sequencing (*Sequence*), the pattern class objects (*PatternClass*) and their connection to the ports (*Binding*). The interface of the expressions allows the outputs of one expression to be the input of another expression, in a dataflow-like manner. This is used to sequence expression execution.

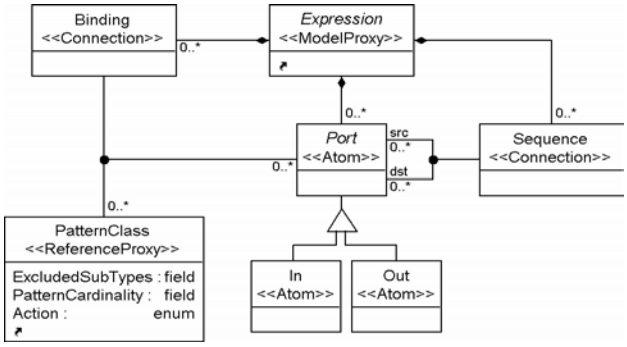


Figure 15: UML class diagram for the abstract syntax classes of GreAT: The interface

A *CompoundRule* can contain other compound rules, *Tests* and *PrimitiveRules*. The primitive rules of the language are to express primitive transformations. A *Test* is a special expression and is used to change the control flow during execution. Figure 16 describes a high-level algorithm that shows all the rules to be the same from outside were each one has a different implementation but the same interface.

```
Function Name  : Execute
Inputs         :   1. List of Packets inputs
```

```
                    2. Expression expression
Outputs         :   1. List of Packets outputs
outputs = Execute(expression, inputs)
{   if(expression is a for block)
        return ExecuteForBlock(expression, inputs)
    if(expression is a block)
        return ExecuteBlock(expression, inputs)
    if(expression is a test)
        return ExecuteTest(expression, inputs)
    if(expression is a rule)
        return ExecuteRule(expression, inputs)
}
```

Figure 16 The expression execution algorithm

The control flow language has the following basic control flow concepts.

- Sequencing – rules can be sequenced to fire one after another
- Non-Determinism – rules can be specified to be executed "in parallel", where the order of firing of the parallel rules is non deterministic.
- Hierarchy – CompoundRules can contain other CompoundRules or Expressions
- Recursion – A high level rule can call itself.
- Test/Case – A conditional branching construct that can be used to choose between different control flow paths.

Note that the approach followed here can be considered as a highly specialized version of the transformation unit concepts introduced in [31]. The hierarchical rules can be viewed as graph transformation modules, but in GReAT the control condition is restricted. Also, GreAT does not address the issue of transactions, as all rule execution is assumed single-threaded.

3.5.1. Sequencing of Rules

If the output interface of a rule is associated with the input interface of another rule, they will execute sequentially. Figure 17 shows the flow of packets through the rules. The packets are shown as a vertical set of letters where each letter refers to host graph object. The packet objects map to the ports of a rule in the vertical layout. Thus the top graph object is bound to the top port and so on. Figure 17(a) shows the initial condition where there are two input packets on the input interface of Rule 1. Rule 1 will fire to first to consume all its input packets and produce a number of output packets as shown in Figure 17(b). Then rule 2 will fire to consume all its input packets to produce a number of output packets (shown in Figure 17(c)).

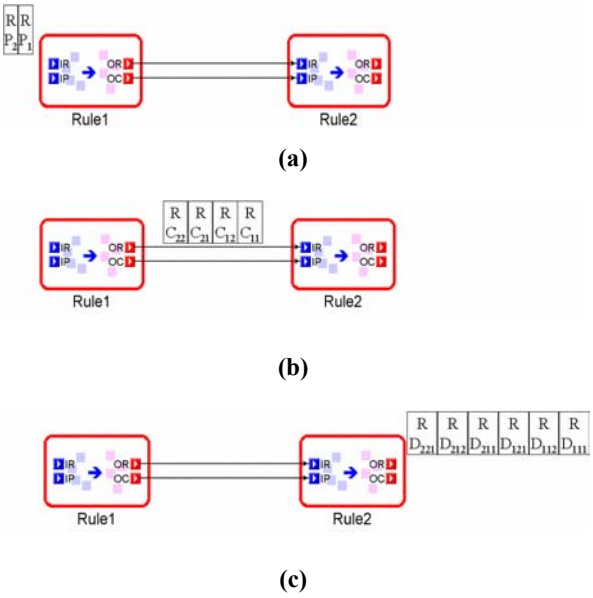


Figure 17 Firing of a sequence of 2 rules

3.5.2. Hierarchical Rules

There are two kinds of hierarchical "container" rules: (1) Block, and (2) ForBlock. Both *Block* and *ForBlock* have the same semantics with respect to rules connected to them. Thus, if in Figure 17 the rules 1 and 2 were hierarchical, then they would have had the same effects as described above. All the semantic differences are internal to the hierarchical rules.

```
Function Name  : ExecuteBlock
Inputs         :   1. List of Packets inputs
                   2. Expression block
Outputs        :   1. List of Packets outputs
outputs = ExecuteBlock(block, inputs)
{   List of Packets outputs
    Stack of Rules ready_rules
    foreach next_rule of block.next_rules()
    {   if(next_rule equals  block)
        {   outputs.Add(inputs )
        }
        else
        {   ready_rule.Push(next_rule,inputs)              }
    }
    while( ready_rules.NotEmpty())
    {   current, arguments = ready_rules.Pop()
        return_arguments = Execute(current,
                                   arguments)
        For Each next_rule of current.next_rules()
        {   if(next_rule equals block)
            {   outputs.add(inputs )
            }
            else
            {   ready_rule.Push(next_rule,inputs)
            }
        }
    }
    return outputs
}
```

Figure 18 Block execution algorithm

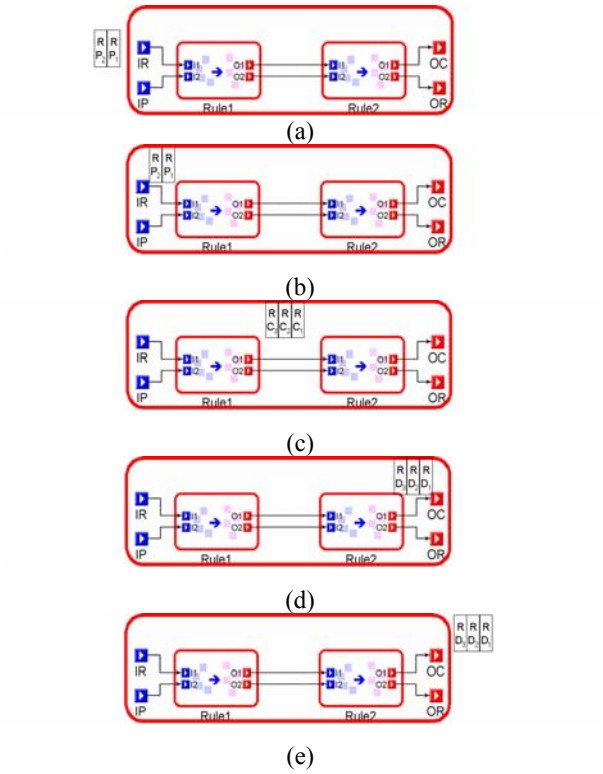
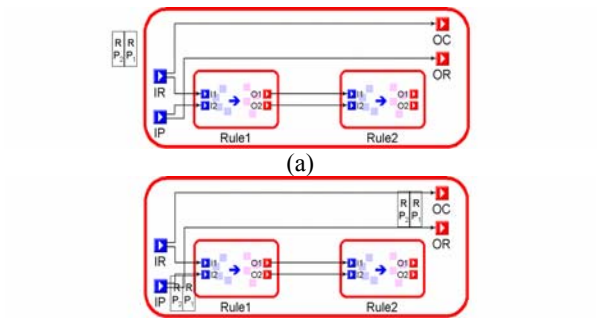
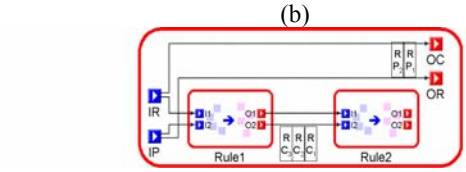


Figure 19 Rule execution of a Block

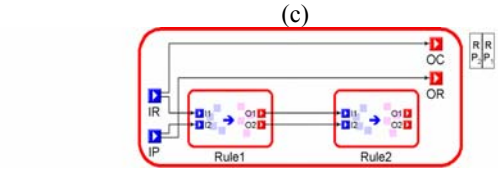
The *Block* has the following semantics: it will push all its incoming packets through to the first internal rule (i.e. it is same as the regular rule semantics). The input interface of the block can be attached to the input interface of any internal block or to the output interface of the block. In other words the block can send output packets from any internal rule or pass its input packets as output. However, the output interface of a block must be attached to exactly one interface and it cannot be attached to two different interfaces. Figure 19 illustrates the execution of rules within a block. Figure 20 illustrates the case when the output interface of a block is connected to the input interface of the same block.



(b)



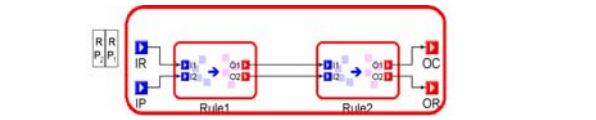
(c)



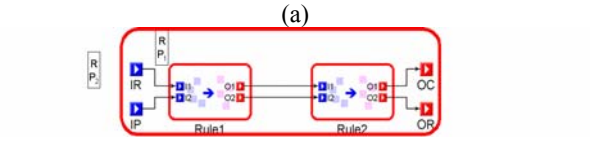
(d)

Figure 20 Sequence of execution within a *Block*

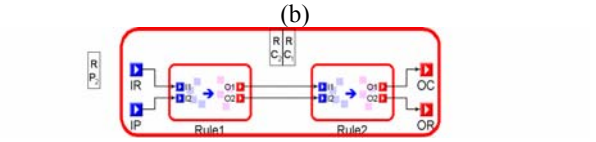
The *ForBlock* has a different semantics for execution within the block. If we have n incoming packets in a *ForBlock* then the first packet will be pushed through all its internal rules to produce output packets and then the next packet will be taken. The semantics are illustrated with the help of an example in Figure 21.



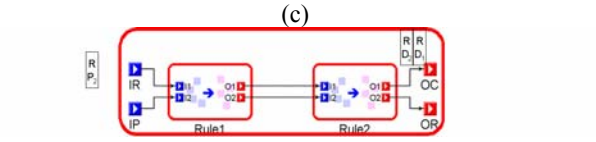
(a)



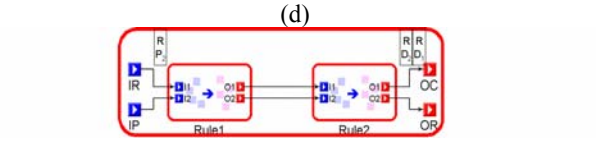
(b)



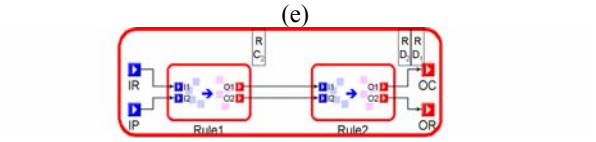
(c)



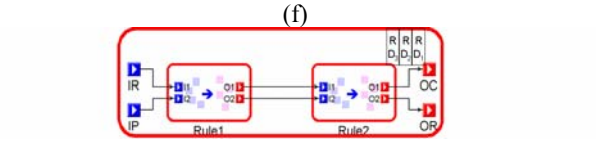
(d)



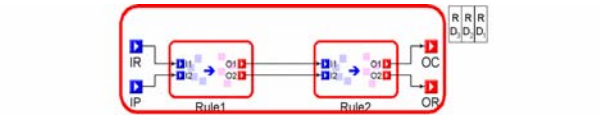
(e)



(f)



(g)



(h)

Figure 21 Rule execution sequence of a *ForBlock*

Similar to the block the input interface of the *ForBlock* can also be associated with the input interface of any internal rule or the output interface of itself.

```
Function Name   : ExecuteForBlock
Inputs          :   1. List of Packects inputs
                    2. Expression forblock
Outputs         :   1. List of Packects outputs
outputs = ExecuteForBlock(forblock, inputs)
{   List of Packects outputs
    foreach input in inputs
    {   returns = ExecuteBlock(forblock, input)
        outputs.Add(returns)
    }
    return outputs
}
```

Figure 22 For block execution algorithm

3.5.3. Branching using test case

There are many scenarios where the transformation to be applied is conditional and a "branching" construct is required. GReAT supports a branching construct called *Test/Case*.

The external semantics of a *Test/Case* is similar to any other rule. When fired or executed it consumes all its input packets to produce some output packets. In Figure 23 a test is shown that has two cases. The *Test* has one input interface and two output interfaces ({OR1, OP1} and {OR2, OP2}). When the test is fired each incoming packet is tested and placed on the corresponding output interface.

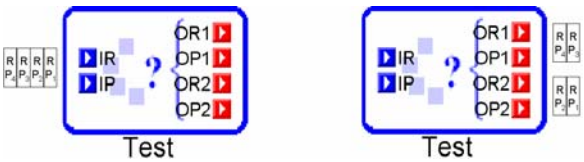


Figure 23 Execution of a *Test/Case* construct

The test must contain at least one *Case*, and a case is a rule with no output pattern and no actions. It contains a pattern (containing *bind* objects only), a guard condition and an input/output interface. If the pattern matches and the guard

evaluates to true, then the case succeeds and the input packet given to the case is passed along, otherwise the case fails.

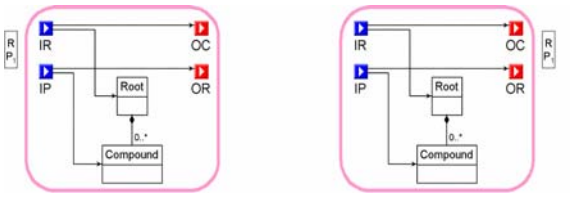


Figure 24: Execution of a single *Case*

Figure 24 shows a case with a successful execution. The input packet has a valid match and so the packet is allowed to go forward.

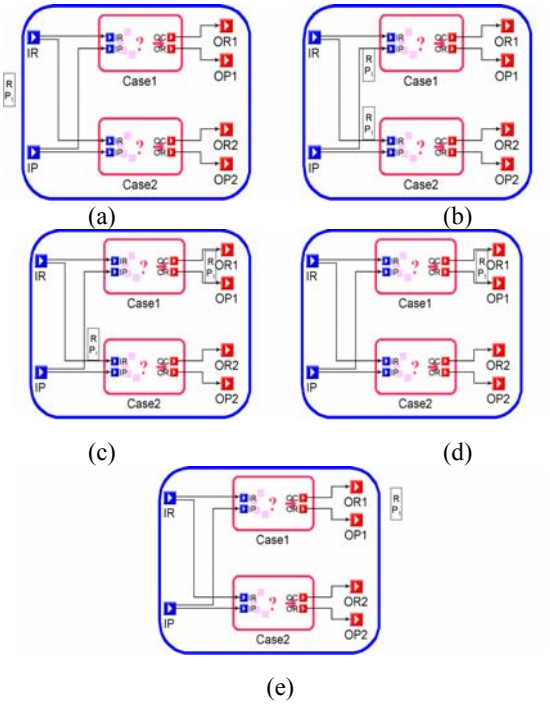


Figure 25 Inside the execution of a *Test*

When a test has many cases then each input packet is checked with each case to see which cases are satisfied for the particular packet and the packet is placed in the output interface of each satisfied case. This behavior is similar to a set of "if" statements without the "else" part. Since the default semantics are that an input packet will be tested with all the cases and more than one case may succeed, there is a requirement for an exclusive style of branching so that only one case succeeds. A variant of this behavior is achieved using a special attribute of a *Case* called the "cut". When *Case* has its "cut" behavior enabled, if the case succeeds

on a given input, the input will not be tried with the subsequent cases. If each case in a test has the "Cut" enabled, then the test will behave like an if-elseif-else programming construct. To implement the "cut" an explicit ordering of the cases is required. The order of testing cases is derived from the physical placement of the case within the test, in the graphical model. The cases are evaluated from top to bottom. If there is a tie in the y co-ordinate then the x co-ordinate is used from left to right.

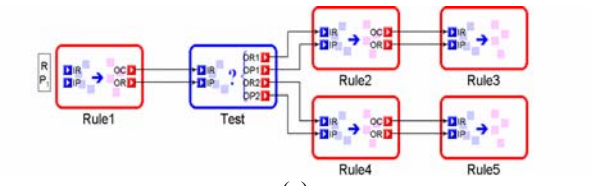
In Figure 25 the execution of a test is shown. An input packet is replicated for each case. Then the input packet is tried with the first case, it succeeds and is copied to the output of the case. Since the "cut" is not enabled in the first case the packet is tried with the second case, this time it fails and the packet is removed. Finally, after all input packets have been consumed the output interfaces have the respective packets.

```
Function Name  : ExecuteTest
Inputs         :   1. List of Packects inputs
                   2. Expression test
Outputs        :   1. List of Packects outputs
outputs = ExecuteTest(test, inputs)
{   List of Packects outputs
    List of Cases cases =
                  test.cases_in_sequence()
    for each input in inputs {
        for each case in cases {
            returns = ExecuteCase(case, input)
            outputs.Add(returns)
            if(case has a cut and return exist)
                break
        }
    }
    return outputs
}
```

Figure 26 Test execution algorithm

3.5.4. Non-deterministic Execution

When a rule is connected to more than one follow-up rule, or when there is a test with more than successful cases, then the execution becomes non-deterministic. The execution engine chooses a path non-deterministically, and the path that is chosen is executed completely before the next path is chosen.



(a)

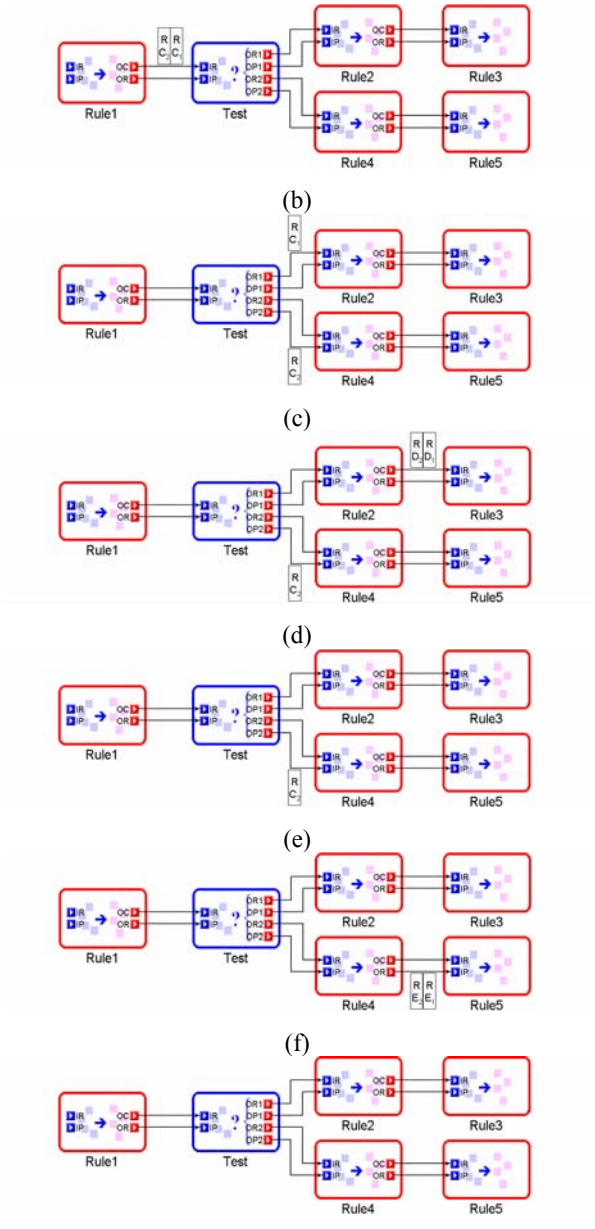


Figure 27 A non-deterministic execution sequence

Figure 27 shows a non-deterministic execution sequence. Here the non-deterministic execution is caused due to a test/case but it could also have been a rule connected to more than one other rule. After the branch there are packets at both the output interfaces of the test. Thus both rule 2 and rule 4 are ready to fire, and rule 2 is chosen non-deterministically and fired, followed by the execution of following rules. This ends at rule 3. Then rule 4 and 5 are fired.

3.5.5. Termination

At one point, the transformation must terminate. A rule sequence is terminated either when a rule has no output interface or when a rule having an output interface does not produce any output packets.
If the firing of a rule produces zero output packets then the rules following it will not be executed. Hence in Figure 27, if rule 4 produced zero output packets then rule 5 would not have been fired.

4. The Implementation

The language described above was *defined* with the help of a metamodel: a UML class diagram, which was then compiled into a GME metaprogram —resulting in a visual modeling environment that allows creating and editing transformation programs. The three sub-languages were defined as three separate, but related class diagrams, thus yielding a modular design for the language. The metamodel composition capabilities of GME [2] allowed this. Such a modular design also enables changing and evolving the sub-languages independently.
The language was *implemented* using an interpreter, but work also has started on a code generator that compiles the transformation rules into executable code. The interpreter is supplied with the transformation rules and the starting input packets (typically the root model of the dominant hierarchy).

4.1.1. The UDM Package

The technology used for the implementation of GReAT is Universal Data Model (UDM) package [22]. UDM is a reflective, meta-programmable package that is supported by a development process and a set of tools to generate C++ accessible interfaces from UML class diagrams of data structures. The generated APIs can utilize a variety of data storage implementations (called “backends”) for models (such as XML, GME model databases, ODBC databases, etc.). The data storage implementation is transparent to the user and the same API can be used to access and store data in any (supported) format.
UDM provides a convenient programmatic access and can be used to build generators or translators for different data structures described in UML class diagrams. Note that the programmer has two different interfaces: one of them is a

domain-specific one, which is generated based on the UML class diagrams, and another, generic one, which allows manipulating objects using symbolic names (class names, attribute names, association role names, etc.). The typical process of using the UDM is as follows:

- A UML class diagram (metamodel) is created in either of the two supported modeling tools (Microsoft Visio or GME). The UML class diagram is then converted into an XML representation with the help of a UDM translator tool.
- The XML file is then used to generate a C++ API (consisting of a source and a header file) specific to the particular class diagram, as well as an XML DTD (to be used in the XML backend). The generated C++ files are then compiled and linked with the generic and backend-specific UDM libraries.
- Networks of objects compliant with the specified UML class diagram can be created either using the (generated) diagram-specific API, or using the generic (symbolic) API.

The tool chain of the UDM process is described below. Figure 28 shows a simplified UDM based development scenario. Note that UDM includes a reflection package, as the meta-models (obtained from the UML class diagram) are explicitly included in the form of initialized data structures.

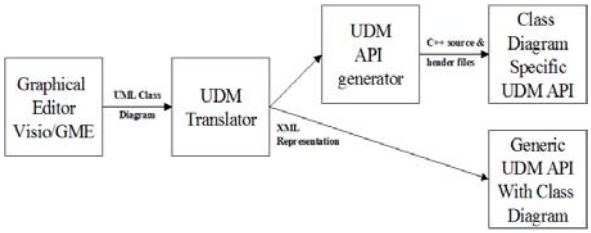


Figure 28 Tool chain for generation of UDM API

The GReAT interpreter is an experimental testbed developed for testing the transformation language and to validate that the language is powerful enough to express common transformation problems. The interpreter takes the input graph, applies the transformations to it, and generates the output graph. Inputs to the GReAT interpreter include: (1) the UML class diagrams for the input and output graphs (i.e. the meta-models), (2) the transformation specification, and (3) the input graph: the input models. The GReAT interpreter traverses the rules according to the sequencing and produces an output graph based upon the actions in the rules.

The architecture of the run time system is shown in Figure 29. The interpreter accesses the input and output graph with the help of a generic UDM API that allows the traversal of input and output graph. The rewrite rules are stored in their own language format and can be accessed using the language specific UDM API. The GReAT is composed of two major components, (1) Sequencer, (2) Rule Executor (RE). The Rule Executor is further broken down into (1) Pattern Matcher (PM) and (2) Effector (or 'output generator'). The Sequencer determines the order of execution for the rules using the '*Execute'* function described above and it calls the *ExecuteRule* for each rule. The rule executor internally calls the PM with the pattern of the rule. The matches found by the PM are used by the Effector to manipulate the output graph by performing the actions specified in the rules.

The Pattern Matcher finds the subgraph(s) in the input graph that are isomorphic to the pattern specification. When a pattern vertex/edge matches a vertex/edge in the input graph, the pattern vertex/edge will be bound to that vertex/edge. The matcher starts with an initial binding supplied to it by the Sequencer. Then it incrementally extends the bindings till there are no unbound edges/vertices in the pattern. At each step it first checks every unbound edge that has both its vertices bound and tries to bind these. After it succeeds to bind all such edges it then finds an edge with one vertex bound and then binds the edge and its unbound vertex. This process is repeated till all the vertices and edges are bound. The recursive algorithm for the matches is shown in Appendices 1 and 2.

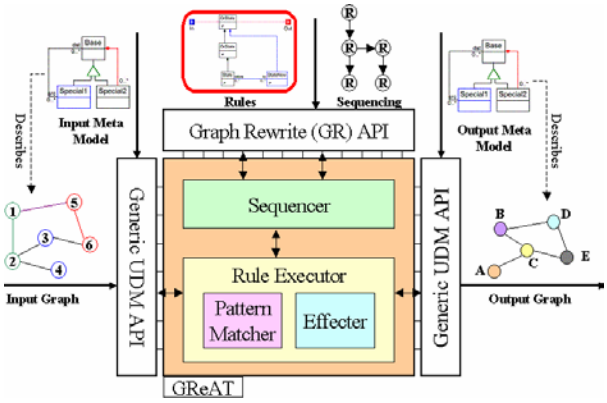


Figure 29 The GReAT interpreter

5. Examples and Results

To test GReAT and to measure its usability we chose some challenge problems that would accurately reflect the needs of the model-to-model transformation application area. The challenge problems chosen were as follows.

- Generate a non-hierarchical Finite State Machine (FSM) from a Hierarchical Concurrent State Machine (HCSM). This problem introduces interesting challenges. To map concurrent state machines to a single machine there is a need for complex operations that include computing a Cartesian product over the state spaces of the concurrent machines. The evaluation of this particular transformation requires a depth-first, bottom up approach and will also test whether the system can allow different traversal schemes.
- The next example is to generate an equivalent Hybrid Automata Network model from a given Matlab Simulink-Stateflow model. This is another non-trivial example as the mapping is not a straightforward one-to-one mapping. 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 an equivalent hybrid automata network model that produces the same dynamic behavior. This algorithm has some complex steps such as state splitting, reachability analysis and special graph walks that make it another interesting problem to try.
- The final example is to build a pre-processor for domain-specific extensions to a language. For example, to develop a pre-processor that converts Aspect-Java code to its equivalent Java code. This example is chosen because it is not a toy problem and should be able to test that system thoroughly and see if the system can be used to solve real world problems.

The diversity of the example problems chosen above gives confidence that if the new language can actually solve all the above-mentioned problems using easy to use concepts and if the system can generate efficient implementations from the specification, then it should be able to solve a large number of non-trivial real world problems.

Out of the challenge problems described above the first three have been solved along with other simple example problems using the GReAT language and interpreter. Flattening the state machine example is implemented using a recursive depth-first bottom up algorithm that first calls flattening on its children

before flattening itself. The reachability analysis problem uses the mark and sweep algorithm [23].

The goal of using a graph transformation based specification language is to increase programmer's productivity. Table 1 shows some preliminary results by comparing the size of and time taken to develop GReAT specifications for model transformation problems to estimated equivalent lines of procedural code. The primitive rules are rules that contain graph transformation specification, while compound rules are higher-level control flow constructs. These preliminary tests have shown that each primitive rule corresponds to approximately 30 lines of hand code. The corresponding hand code is fairly complex and not very straightforward to write. This makes us believe that the language can actually provide increase in productivity. However, better tests need to be designed and performed using more experiments to provide more precise results.

Table 1 Comparison of GReAT implementation vs code

Problem	**GReAT**		**Hand code**
	Primitive/Compound Rules	Time (man-hours)	Est. LOC
Mark and sweep algorithm on Finite State Machine (FSM)	7/2	~2	100
Hierarchical Data Flow (HDF) to Flat Data Flow (FDF)	11/3	~3	200
Hierarchical Concurrent State Machine (HCSM) to Finite State Machine (FSM)	21/5	~8	500
Matlab Simulink/ Stateflow to Hybrid System	66/43	~20	3000

5.1. HCSM to FSM example

A model transformer that converts Hierarchical Concurrent State Machine (HCSM) models to Finite State Machine (FSM) is described in this section. This transformer uses the HCSM metamodel (Figure 2), and the FSM metamodel (Figure 3). A metamodel was also introduced to define the (temporary) cross-links

(Figure 4). The transformation algorithm used can be divided into two parts. The first part of the algorithm flattens the HCSM graph within the HCSM domain, and in the second part an isomorphic copy of the flattened HCSM is created in FSM domain.

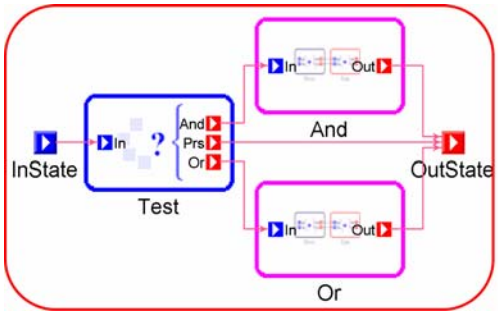


Figure 30 The top-level rule

The flattening algorithm is depth-first/bottom-up. This is achieved using a recursive block *Top-level* (shown in Figure 30). The *top-level* gets input from the input port *InState.* The input can be of type *or-state*, *and-state* or *simple-state*. The first expression inside *top-level* is a test/case called *Test* that branches according to the type of input. If the input is an *and-state* it is passed to the block called *And* that flattens the *and-state*. If the input is an *or-state* it is passed to the block called *Or* that deals with the flattening the *or-state,* and if the input is a *simple-state* it is passed directly to the output port *OutState* without any processing.

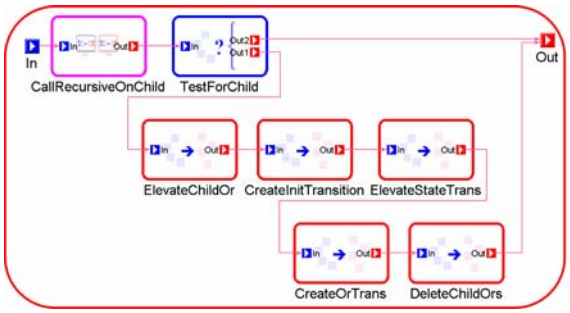


Figure 31 Inside the OR rule

Figure 31 shown the rules inside the *Or* block of Figure 30. These internal rules are used to flatten an *or-state*. The first rule in the rule chain is *CallRecursiveOnChildren*, a block that finds all the contained states of the or-state being processed and then called the top-level rule (Figure 30) for each of them. The next expression *TestForChild* will only execute after the recursive calls have been executed and thus at this point the *or-state* being flattened will only contain flat *or-states* (*and-state* when flattened will also produce an equivalent flat *or-*

state) and primitive states. *TestForChild* is a test/case and it tests to see if the or-state contains any *or-state* type children. If not then the or-state is already flat and is passed to the output port. If the or-state contains other or-states then it is passed to *ElevateChildOr* rule (Figure 32).

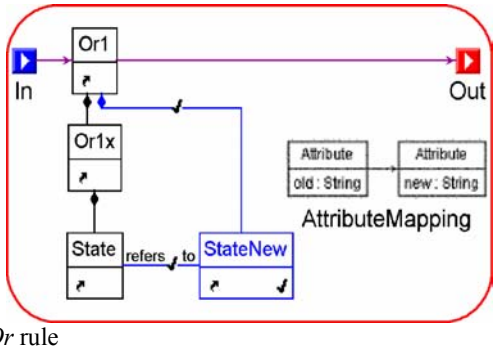


Figure 32 *ElevateChildOr* rule

Figure 32 shows *ElevateChildOr* rule. In the rule the or-state being flattened is *Or1* and for each contained *Or1x* child or-state having a child *State* a new *StateNew* is created as the child of *Or1*. The next rule in sequence is *CreateInitTransition*. This rule is used to create equivalent transitions for the init transition within Or1. *ElevateTrans* is the next rule and it creates transitions for each transition contained in Or1x. *CreateOrTrans* The next rule is used to create equivalent transitions for each transition that is incident upon Or1x. The last rule in the sequence *DeleteChildOrs* is used to delete Or1x. At this stage the Or1 state is a flat or state.

Flattening an and-state is more complex and requires a few more rules. For the sake of brevity we have not described the flattening of the *and-states.*

6. Conclusion and Future Work

This paper has shown a technique for model transformations based on graph transformations. Model transformations have some specific requirements such as (1) multiple graph domains may be involved in the transformation, (2) there is a need for the specification and use of links that cross domains, and (3) support for sequencing the transformation rules are required. Due to these requirements previous approaches could not be directly used.

Graph Rewriting and Transformations (GReAT): a new, graphical language that addressed these requirements was introduced. GReAT is based on the use of UML class diagrams (and OCL) for representing the domains of the

transformations (and structural integrity constraints over those domains). Transformations over multiple domains are supported, and cross-links among domains are defined at the metamodeling level.

The transformation language itself is divided into three sub languages: (1) Pattern Specification language, (2) Graph Rewriting/Transformation language and (3) the language for Controlled Graph Rewriting and Transformation. The Pattern Specification language introduced a concise way to represent fairly complex graphs, and various pattern matching algorithms were also developed. The Graph Rewriting/Transformation language is used to define graph transformation steps. Pattern graphs are embellished with actions like *new*, *bind*, and *delete* to express actions within a transformation. Pre-conditions for the transformations are captured in the form of a guard, and attribute mappings are used to modify the values of attributes. The language for Controlled Graph Rewriting and Transformation defines high-level, hierarchical control structures for rule sequencing, modularization, and branching. We have shown the syntax and semantics of the graph transformation language, and its implementation. A number of small to medium size model transformation problems have been solved using this language.

There are a number of open questions that we would like to address in our ongoing research. Although the current language is powerful enough for writing complex transformation programs, we need to verify it on more complex examples. Currently, the execution engine is not efficient, and we are working on a code generator that will generate executable code from the transformation specifications expressed in GReAT. It is expected, that the efficiency of the language implementation will be greatly improved. Experiments need to be performed that measure efficacy of the GReAT implementation with respect to hand code. The primary goal of this research is to decrease the development and maintenance time of model transformers. To verify the increase in productivity, experiments need to be carefully designed and performed using many subjects. We plan to address these issues in further research.

7. Acknowledgements

The DARPA/IXO MOBIES program and USAF/AFRL under contract F30602-00-1-0580, and the NSF ITR on "Foundations of Hybrid and Embedded Software Systems" have supported in part, the activities described in this paper.

8. References

[1] J. Sztipanovits, and G. Karsai, “Model-Integrated Computing”, Computer, Apr. 1997, pp. 110-112

[2] A. Ledeczi, et al., “Composing Domain-Specific Design Environments”, Computer, Nov. 2001, pp. 44-51.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1998.

[4] “The Model Driven Architecture”, http://www.omg.org/mda/, OMG, Needham, MA, 2002.

[5] “Request For Proposal: MOF 2.0 Query/Views/Transformations”, OMG Document: ad/2002-04-10, 2002, OMG, Needham, MA.

[6] Agrawal A., Levendovszky T., Sprinkle J., Shi F., Karsai G., “Generative Programming via Graph Transformations in the Model Driven Architecture”, Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA , Nov. 5, 2002, Seattle, WA.

[7] Grzegorz Rozenberg, “Handbook of Graph Grammars and Computing by Graph Transformation”, World Scientific Publishing Co. Pte. Ltd., 1997.

[8] Blostein D., Schürr A., ”Computing with Graphs and Graph Rewriting”, Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.

[9] H. Gottler, “Attributed graph grammars for graphics”, H. Ehrig, M. Nagl, and G. Rosenberg, editors, Graph Grammars and their Application lo Computer Science, LNCS 153, pages 130-142, Springer-Verlag, 1982.

[10] H. Göttler, "Diagram Editors = Graphs + Attributes + Graph Grammars," International Journal of Man-Machine Studies, Vol 37, No 4, Oct. 1992, pp. 481-502.

[11] J. Loyall and S. Kaplan, "Visual Concurrent Programming with Delta-Grammars," Journal of Visual Languages and Computing, Vol 3, 1992, pp. 107-133.

[12] D. Blostein, H. Fahmy, and A. Grbavec, “Practical Use of Graph Rewriting”, 5th Workshop on Graph Grammars and Their Application To Computer Science, Lecture Notes in Computer Science, Heidelberg, 1995.

[13] U. Assmann, “How to Uniformly specify Program Analysis and Transformation”, Proceedings of the 6 International Conference on Compiler Construction (CC) '96, LNCS 1060, Springer, 1996.

[14] A. Maggiolo-Schettini, A. Peron, “A Graph Rewriting Framework for Statecharts Semantics”, Proc.\ 5th Int.\ Workshop on Graph Grammars and their Application to Computer Science, 1996.

[15] A. Radermacher, ``Support for Design Patterns through Graph Transformation Tools", Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, Sep. 1999.

[16] A. Bredenfeld, R. Camposano, “Tool integration and construction using generated graph-based design representations”, Proceedings of the 32nd ACM/IEEE conference on Design automation conference, p.94-99, June 12-16, 1995, San Francisco, CA.

[17] H. Fahmy, B. Blostein, “A Graph Grammar for Recognition of Music Notation”, Machine Vision and Applications, Vol. 6, No. 2 (1993), 83-99.

[18] G. Engels, H. Ehrig, G. Rozenberg (eds.), “Special Issue on Graph Transformation Systems”, Fundamenta Informaticae, Vol. 26, No. 3/4 (1996), No. 1/2, IOS Press (1995).

[19] G.Schmidt, R. Berghammer (eds.), “Proc. Int. Workshop on Graph-Theoritic Concepts in Computer Science”, (WG ’91), LNCS 570, Springer Verlag (1991).

[20] H.Ehrig, M. Pfender, H. J. Schneider, “Graph-grammars: an algebraic approach”, Proceedings IEEE Conference on Automata and Switching Theory, pages 167-180 (1973).

[21] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.

[22] A. Bakay, “The UDM Framework,” http://www.isis.vanderbilt.edu/Projects/mobies/.

[23] J. McCarthy “Recursive functions of symbolic expressions and their computation by machine – I”, Communications of the ACM, 3(1), 184-195, 1960.

[24] Uwe Assmann, “Aspect Weaving by Graph Rewriting”, Generative Component-based Software Engineering (GCSE), p. 24-36, Oct 1999.

[25] G. Karsai, S. Padalkar, H. Franke, J. Sztipanovits, ”A Practical Method For Creating Plant Diagnositics Applications”, Integrated Computer-Aided Engineering, 3, 4, pp. 291-304, 1996.

[26] E. Long, A. Misra, J. Sztipanovits, “Increasing Productivity at Saturn”, IEEE Computer Magazine, August 1998.

[27] AGG, http://tfs.cs.tu-berlin.de/agg/.

[28] J. D. Lara , H. Vangheluwe, “Using AToM^3 as a Meta-CASE Tool”, Proceedings of the 4th International Conference on Enterprise Information Systems ICEIS'2002 , 642-649, Ciudad Real, Spain, April 2002.

[29] “Dome Guide”, Honeywell, Inc. Morris Township, N.J, 1999.

[30] Kim Mason, “Moses Formalism Creation – Tutorial”, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092, Switzerland, February 9, 2000.

[31] H. Kreowski, S. Kuske: “Graph Transformation Units and Modules,” in H. Ehrig, G. Engels, H. Kreowski, G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, pages 607-638. World Scientific, Singapore, 1999.

9. Appendices

9.1. Appendix 1 – Pattern matching algorithm using simple patterns

```
Function Name: PatternMatcher
Inputs              :      1. Pattern Graph pattern
                                  2. Match p_match (a partial Match)
Outputs             :      1. List of Matches matches

matches = PatternMatcher (pattern,  p_match)
{
      foreach pattern edge that has both Src and Dst vertices having
valid binding
      {     if(corresponding graph edge doesn't exists between graph
vertices)
            {     return an empty match list
                  Bind pattern and host graph edge and add binding to
p_match
                  Delete the pattern edge from the pattern
            }
      }
      Edge edge = get pattern edge with one vertex bound to host graph
      if(edge exists)
      {     vertices = vertices of the host graph adjacent to the bound
vertex
            make a copy of pater in new_pattern
            Delete edge from new_pattern
            foreach vertex v in vertices)
            {     new_match = p_match + new binding(unbound pattern
vertex, vertex)
                  ret_match = PatternMatcher(new_pattern, graph,
new_match)
                  Add ret_match to matches
            }
            Return matches
      }

      If(all patern edges are bound)
      {     Add p_match to matches
            Return matches
      }
      else
            Return empty list
}
```

9.2. Appendix 2 – Pattern matching algorithm with fixed cardinality

```
Function Name: PatternMatcher

Inputs:      1. Pattern Graph pattern
       2. Match p_match (a partial Match)
Outputs:  1. List of Packects matches

matches = PatternMatcher (pattern, p_match)
{            new_pattern = copy of Pattern.
foreach pattern edge with both Src and Dst vertices bound
{      if(corresponding edge doesn't exists between host graph vertices)
              return false.
       Add edge binding to p_match
       Delete edge from new_pattern.
}

Edge edge = pattern edge with one vertex bound to host graph
if(edge exists)
{      Delete edge from new_pattern.
       foreach vertex v in bound vertices of edge
       {      peer_vertices[v] = vertices adjacent to vetrex bound to v
       }
       Intersect all the peer_vertices to form new list peer
       If(cardinality of peer Ci >= Cd cardinality of corresponding
pattern vertex)
       {      For(Each combination of Cd from Ci)
              {      peer_c is the unique combination
                     new_match = p_match + new binding(pattern vertex,
peer_c)
                     ret_match = PatternMatcher(new_pattern, new_match)
                                  Add ret_matches to Matches
              }
              Return matches.
       }
}

If(all patern matches are bound)
{      Add p_match to matches.
       return matches.
}
else
       return enpty list.
}
```