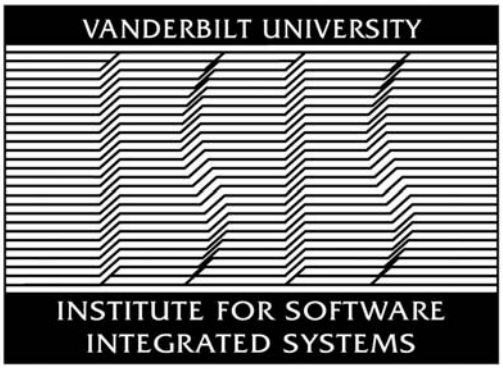


Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235

TECHNICAL REPORT

TR #: ISIS-03-401
Title: Interpreter Writing Using Graph Transformations
Authors: Aditya Agrawal, Gabor Karsai and Feng Shi

Interpreter Writing Using Graph Transformations

Aditya Agrawal
aditya.agrawal@vanderbilt.edu

Gabor Karsai
gabor@vuse.vanderbilt.edu

Feng Shi
feng.shi@vanderbilt.edu

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA
+1(615) 343-7567

ABSTRACT

This paper introduces a UML-based approach for specifying model transformations. The technique is based on graph transformations, where UML class diagrams are used to represent the graph grammars of the input and the output of the transformations, and the transformations are represented as explicitly sequenced elementary rewriting operations. The paper discusses 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

Graph grammars and graph transformations (GGT) have been recognized as a powerful technique for specifying complex transformations that can be used in various situations in a software development process [13][14][15][16]. Many tasks in software development can 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 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 execution models and analysis models. Execution 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 analytical equivalents. In a larger context, model transformations are essential in many systems and development practices, not only MIC. Here, we will use MIC as the software development process, but the same motivation applies to OMG's Model-Driven Architecture as well.

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.

Section 2 briefly introduces Model Integrated Computing (MIC), and reviews graph grammars and transformations. Section 3 describes the solution to the model transformation problem. Section 4 provides details of the implementation, while Section 5 shows a few selected applications and 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 capturing the design are then used to synthesize executable systems, perform analysis or drive simulations. The advantage of this methodology is that it speeds up the design process, facilitates evolution, helps in system maintenance and reduces the cost of the development cycle [1].

The MIC development cycle (see Figure 1) starts with the formal specification of a new application domain. The specification proceeds by identifying the concepts, their attributes, and relationships among them through a process called metamodeling. 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 of the domain is used to generate a Domain Specific Design Environment (DSDE). 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 synthesize executable code, perform analysis or drive simulators, we have to convert the models into another format like executable code, input format of some analysis tool or configuration files for simulators. This mapping of models to a more useful form is called model interpretation and is performed by model interpreters. Model interpreters are programs that convert models of a given domain into another format. For mapping each domain to output format a unique model interpreter is required. The output can be considered as another model that conforms to a different metamodel and thus these model interpreters can be considered to be mappings between models [1].

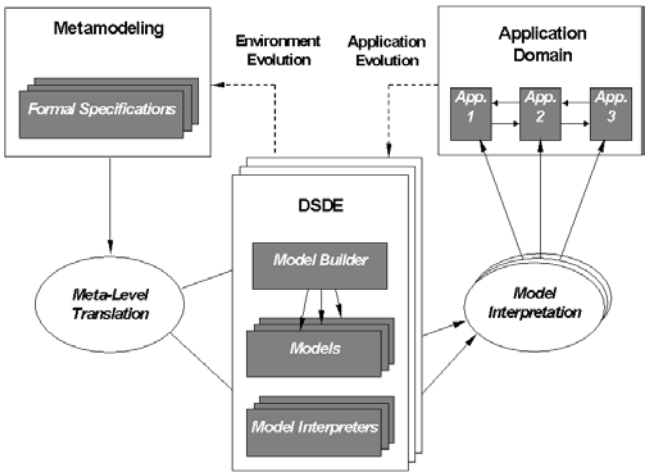


Figure 1 The MIC Development Cycle [2]

The premier MIC implementation is built around a metaprogrammable toolkit called Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems (ISIS), Vanderbilt University. It provides an environment for creating domain-specific modeling environments [2]. The metamodeling environment of GME is based on UML class diagrams [3]. It is used to describe a domain specific modeling language and a corresponding environment by capturing the syntax, semantics and visualization rules of the target environment. 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. Thus the core of GME is used both as the metamodeling environment and the target environment.

GME has both a metamodeling environment and metamodel interpreter that generates a new modelling environment from the metamodels. However there are no generic tools or methods to automatically generate domain specific model interpreters. Each model interpreter is written by hand and this is the most time consuming and error prone phase of the MIC approach. There is a need to develop methods and tools to automate and speed up the process of creating model interpreters.

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]. The MDA is a particular application of the MIC approach where the domain language will be UML 2.0. However, a more general approach to the MDA problem will be to achieve domain specific model driven software development. [6]

2.2 Graph Grammars and Transformations

On analysing the problem of how to speed up the development of model interpreters we perceive the need for a way to specify the operation of model interpreter. The specification can then be used to generate the model interpreter code. However, this task is non-trivial as a model interpreter 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 a visual modeling language. In fact, GME creates and manipulates 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 semantics are imposed on them, etc.

From a mathematical viewpoint one can recognize that 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. Thus, the model transformation problem can be converted into a graph transformation problem. We can then use the mathematical concepts of graph transformations to formally specify the intended behaviour of a model interpreter.

There are a variety of graph transformation techniques described in [7][8][9][10][11][12][19]. The prominent among these are node replacement grammars, hyper edge replacement grammars, algebraic approaches and programmed graph replacement systems. The next few paragraphs will discuss each approach and show why they cannot be used directly to solve the model interpretation problem [7].

Node replacement grammars are a class of graph grammars that are based primarily upon the replacement of nodes in a graph. The basic production of every node replacement

grammar has a LHS subgraph (called mother graph) that produces an RHS subgraph (called daughter graph). Usually the LHS subgraph consists of only one node making this class of grammars context free. The productions can be applied whenever there is a mother node in the host graph and if there are two productions that can be applied then the order of application is non-deterministic. This can cause different production application sequences to yield different resulting graphs [7].

A property called confluence is defined as follows: a graph transformation system is confluent if and only if the production application sequence does not affect the final result of the transformation. In order to satisfy this property there are many restricted node replacement grammars that satisfy confluence [7].

Node replacement grammars are suitable for defining and identifying graphical language but are not suitable for defining transformation algorithms. The primary reason is that these languages are context free and have no production sequencing, and hence difficult to represent algorithms in them.

Hyperedge replacement grammars deal with the productions that replace hyper edges by subgraphs. Each production has a hyperedge on the LHS, which is replaced by a subgraph on the RHS. Hyperedge replacement by definition is confluent, associative and parallelizable. But its shortcomings are similar to the node replacement grammars. These too are context free and do not provide sequencing and conditional application of productions [7].

The next approach to graph grammars is the algebraic approach, developed at the University of Berlin. The approach is based on a generalization of Chomsky grammars from strings to graphs. The main goal was to generalize the string concatenation to a gluing construction of graphs. The approach is algebraic because graphs are considered as special kinds of algebra and the gluing is defined by algebraic constructions called pushouts. The pushout approach has been taken from a more general field of category theory and has been applied to the more specific field of algebraic theory of graph grammars. There are two basic algebraic approaches (a) Double PushOut (DPO) and (b) Single PushOut (SPO). Significant research has been done on pushouts and how productions can be parallelized. The algebraic approach is more powerful and has concepts for sequencing and parallelizing the rules [7][20].

However, the sequencing of rules is limited only to sequential and parallel execution of the rules. It lacks high-level sequencing constructs such as conditional branching of productions, loping and recursion. The lack of high-level sequencing means that the user cannot represent and/or choose between depth-first search or breadth-first search.

The last approach to be discussed is that of programmed replacement systems, which are the most practical of all the approaches discussed so far. The leading research result is the PROgrammed GRaph REplacement System (PROGRES)[8][19]. The major breakthrough of PROGRES is that they concentrate equally on productions and sequencing of the productions. Thus the system has a graph replacement language that defines the productions and also programming constructs that define the order of application of the productions. The PROGRES system consists of two parts - the first is a logic based structure replacement system that describes graph transformation productions of the language and the second is a collection of programming constructs such as recursion, non-deterministic application of productions, conditions and loops. Apart from these PROGRES can also specify static integrity constraints on the graphs. This is done with a language called schemas that define the graph domain. However, PROGRESS is also not suitable for specifying model-interpreters because: (1) schemas are powerful but not as powerful or as widely used as UML class diagrams to specify integrity constraints, (2) PROGRESS deals mainly with transformations on a single graph and do not produce a new graph that conforms to a different schema/metamodel, and (3) PROGRESS is mainly a programming language with graphical productions and thus not at the level of abstraction desired for specifying model-interpreters [7][8].

Apart from these mathematical approaches there is another dimension to graph transformation systems. Namely, how the productions are specified. Is the specification syntax and semantics easy to use and readable? Some of the prominent notations are the Y [9], X [10], and Delta [11] notation. However, there are few diagrammatic and graphical notations for the specification of the control flow of these productions [12].

One can recognize the existing GGT approaches are not well suited for specifying and implementing model interpreters. Hence, a new approach targeted for model-to-model transformation is required. The new approach should have the following features:

1. As UML is a widely used and accepted standard for specification of classes and objects. It should use UML for specification of static structure (i.e. that data model) and integrity constraints.
2. There should be support for transformations that create an entirely different graph based upon a given graph. The two graphs may have different static structure and integrity constraints.
3. The new approach should be expressive enough to specify model interpreters that convert models of high-level graphical languages to low-level implementations, with no or minimal textual coding.

4. The new language should have efficient implementations of its programming constructs. The implementation should have comparable efficiency to its equivalent hand written code.
5. The new language should be "user friendly" and increase programmer productivity.

The new language should be usable and suited for addressing the needs of mapping graphical languages to their low-level implementation. It should drastically shorten the time taken to develop a new 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 machine [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, or GreAT for short.

This language can be divided into 3 distinct parts.

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

Before we discuss the language we should spend some time to define the basic concepts.

3.1 Graph Definition

The graphs used in the GreAT language are typed and attributed multi-graphs and are defined below.

3.1.1 Vertex

A vertex is 3-tuple (name, type, attributes), where $name \in Name$ and $type \in Type$. Name is a set of all names in the system, Type is a set of all types in the system and attributes is set of attribute that are defined as (name, type, value), where $value \in Value$ and Value is the set of all values in the system. The functions defined on vertices are:

(1) Name: V→String,
$\forall v \in V, v.Name() = \{name \mid v = (name, type, attributes)\}$

(2) Type: V→String,
$\forall v \in V, v.Type() = \{type \mid v = (name, type, attributes)\}$.

3.1.2 Edge

An edge is a 4-tuple (name, type, src, dst), where both src and dst are elements of V, the set of all vertices. The functions on edges are

(1) Name: E→String,
$\forall e \in E, e.Name() = \{name \mid e = (name, type, src, dst)\}$,

(2) Type: E→String,
$\forall e \in E, e.Type() = \{type \mid e = (name, type, src, dst)\}$,

(3) Src: E→V,
$\forall e \in E, e.Src() = \{src \mid e = (name, type, src, dst)\}$

(4) Dst: E→V,
$\forall e \in E, e.Dst() = \{dst \mid e = (name, type, src, dst)\}$

3.1.3 Graph

A graph is an ordered pair (GV, GE), Where $GV \subseteq V$, $GE \subseteq E$ and $\forall e \in GE, Src(e) \in GV \wedge Dst(e) \in GV$.

3.2 The Pattern Specification Language

The heart of a graph transformation language is the pattern specification language and pattern matching. The pattern specification found in graph grammars and transformation languages [7][8][9][10][17][18][19][20] are not sufficient for our purposes, as they do not follow UML concepts. This paper introduces an expressive yet easy to use pattern specification language, which is tightly coupled to the UML class diagrams. String matching will be used to draw analogies.

Recall that the goal of the pattern language is to specify patterns over graphs (of objects), where the objects belong to specific classes. 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 (network) constructs formed over the objects that are instances of classes introduced in the class diagram.

3.2.1 Simple Patterns

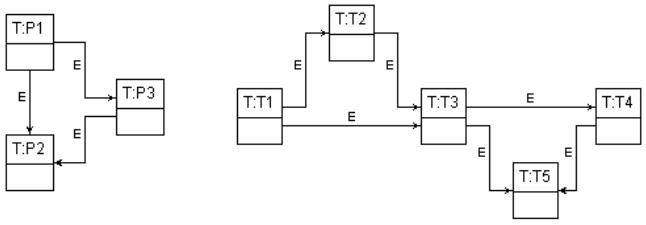
A simple pattern in string matching is the exact string that is being searched for in a larger structure. For example, the string "success" is a simple pattern to be matched in a document. This class of patterns are represented as the specific sub-graph in graph matching. For example, if we were looking 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.

Thus, we can define pattern vertices and pattern edges to be the same as vertices and edges respectively. In order to find and return matches from the matcher we have to define matches. A match is a pair (MVB, MEB), where $MVB \subseteq VB$ and $MEB \subseteq EB$. VB and EB are the set of all possible vertex and edge bindings. A vertex binding is defined as a pair (v, pv), where $v \in V$ and $pv \in PV$ and PV is the set of all pattern vertices. An edge binging is also a

pair (e, pe), where $e \in E$ and $pe \in PE$ and PE is the set of all pattern edges.

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. In string matching, the same string can occur many times and can overlap. For example consider the string "success" in a document containing the sentence "A great successsuccess". It is not obvious which of the two overlapping instances of success should be returned. If an ordering is imposed, we can say that the first occurrence of success should be used. However, in graphs there is no obvious ordering of vertices and edges.

Consider the example in Figure 2(a). The figure describes a pattern that has three vertices $P1, P2 \& P3 \in V$ and for each each P, Type(P) = T. The pattern can match with the host graph shown in Figure 2(b) to return two valid results {(T1,P1), (T3,P2), (T2,P3)} or {(T3,P1), (T5,P2), (T4,P3)}. For sake of simplicity 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 staring point of the search and the exact implementation of the algorithm.



(a) Pattern **(b) Host graph**

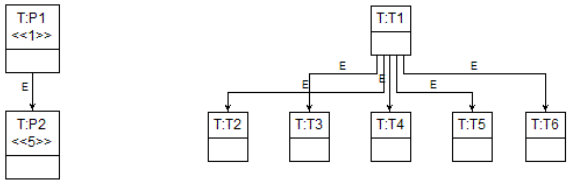
Figure 2 Non-determinism in matching a simple pattern

The solution for this problem is to return a 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.

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.

3.2.2 Fixed Cardinality Patterns

Consider an example from the domain of textual languages. A string needs to be matched such that it starts with an 's' and is followed by 5 'o's. To specify such a pattern string 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 set of this particular language.



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

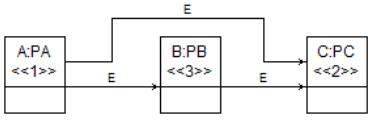
Figure 3 Pattern specification with cardinality

The same argument holds for graphs, and a similar technique can be used. The pattern vertex definition can be extended to a triple (name, type, cardinality), where cardinality is an integer and vertex binding can be defined as a pair (vs, pv), where $vs \subseteq V$. For example, Figure 3(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 {(T1,P1), ({T2, T3, T4, T5, T6},P2)}.

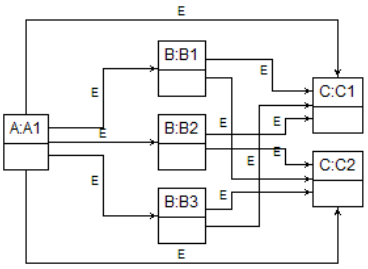
The fixed cardinality pattern and matching also have non-determinism. Even in this case the issue can be dealt with by returning all the possible matches. If all the possible matches are returned there is a problem of returning a large number of matches. For example in Figure 3, if the host graph contained 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 text, we have the advantage of a strict ordering from left to right, while graphs don't. By just extending the example in Figure 3 with another pattern vertex we see that the specification is ambiguous.

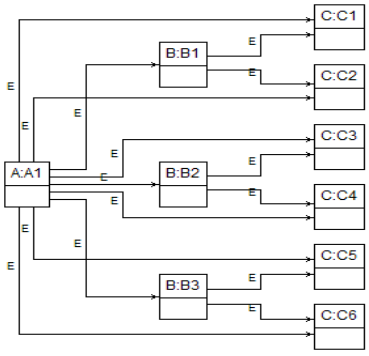
In Figure 4 (a) we see a pattern having three vertices. There are different semantics that can be associated with the pattern. One possible semantic is to consider each pattern vertex pv to have a set of matches equalling the cardinality of the vertex. Then an edge between two pattern vertices pv1 & 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 4 (a) gives the graph in Figure 4 (b).



(a) Pattern with three vertices



(b) Set semantics



(c) Tree semantics

Figure 4 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 3.2.1.

The set semantics will always return a match of the structure shown in Figure 4 (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 4 (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 = (V1, pv1) and b2 = (V2, pv2) be the bindings for pv1 and pv2 respectively. Then

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

This semantics when applied to the pattern gives Figure 4 (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 4 (c) while for the traversal sequence pa, pb, pc we will get a different graph as shown in Figure 5. Another problem with the tree semantics is that graphs like the one shown in Figure 4 (b) cannot be expressed in a concise manner.

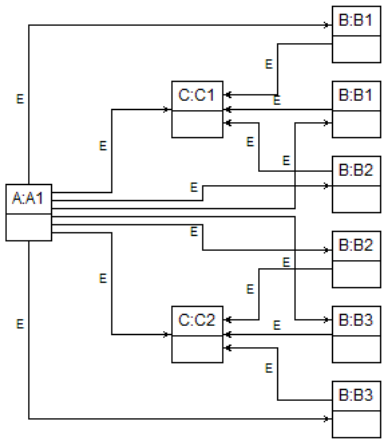


Figure 5 Conflicting match for the tree semantics

Both the semantics discussed so far are incomplete in the sense that certain graphs cannot be expressed with it. Choosing either compromise 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.

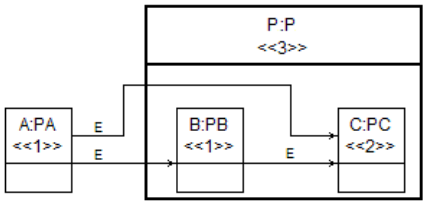
Fortunately, there is a good solution that solves all the problems. The solution is to use an extended set notation that is more expressive.

3.2.3 Extending the Set Semantics

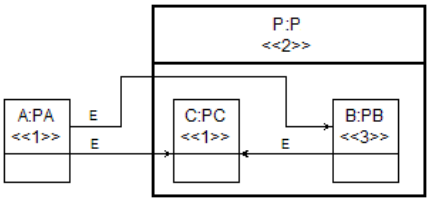
For example we want to match the string "sxyxyxy", we see that "xy" is repeated 3 times. Extending the notation used before we would express it as "s3(xy)". Using parenthesis we were able to represent the fact that the "xy" sequence should occur 3 times. A similar notion can be used in graphs as well. 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 6 (a) shows the pattern that would express the graph in Figure 4 (c) and Figure 6 (b) shows the graph the expresses the graph in Figure 5. With respect to the pattern P in Figure 6 (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 4 (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 4 (c)



(b) Pattern for Figure 5

Figure 6 Hierarchical patterns using set semantics

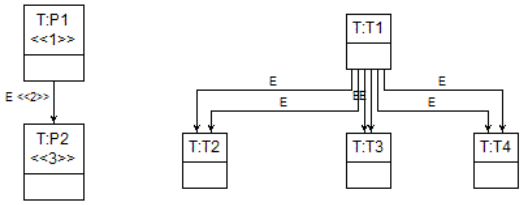
At this point we see that we can express a large variety of graphs in an intuitive, concise and precise way. However, a large number of graphs are missing from the Grouped Set Semantics (GSS) that we described above. This class of graphs are those having more than one edge for the same pair of vertices.

3.2.4 Cardinality For Edges

Adding cardinality to pattern edges helps us express a larger number of graph patterns in a compact manner. Another example is called for and is shown in Figure 7. 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

$$\forall v1 \in V1, v2 \in V2, \overset{C}{\underset{n=1}{\exists}} e_n(v1, v2) \quad \textbf{… Relation 2}$$

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.



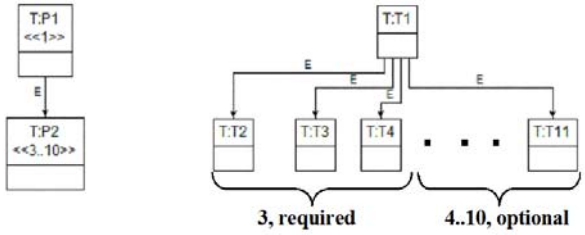
(a) Pattern **(b) Matching Host graph**

Figure 7 Pattern with cardinality on edge.

3.2.5 Variable Cardinality

Sometimes, the sub graph to be matched is not fixed but part of a family of graphs. For example, again from the string matching world, 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 in terms of regular expressions as “s(b)+”. In the general case the number of ‘b’s can be bound by two number, 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).



(a) Pattern **(b) Family of graphs**

Figure 8 Variable cardinality pattern and family of graphs

In Figure 8 we see a variable cardinality example. The pattern in Figure 8 (a) specifies that 3..10 P2s can be connected to a P1, thus the family of graphs represented is given in Figure 8 (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 8 (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.

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

Relation 3 states that from the set of matches that will be returned 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 Graph Rewriting Transformation Language

Pattern specification is required and a very important part of any graph transformation language. There are also some other concerns such as specification of static structural constraint in graphs and to ensure that these are maintained through the transformations [8]. This problem has been addressed in a number of other approaches such as [17][18].

In model-interpreter structural integrity is a bigger concern because model-to-model transformations usually transform models from one domain to models that conform to another domain. This makes the problem two fold. The first problem is to specify and maintain two different models conforming to two different meta-model (in MIC meta-models are used to specify structural integrity constraints). There is another, bigger problem: maintaining references between the two models. It is important to maintain some sort of reference, link and other intermediate data to store temporary values and to correlate graph objects between the two domains.

To illustrate the point let us consider a very simple transformation that needs to transform models conforming to one meta-model to another. For sake of simplicity we consider that the source model has only one type on vertices V1 and only one type of edges E1 and that the destination 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 for each vertex and edge in the source. The algorithm first creates a vertex for each vertex in the source and then creates the edges. We see that for the second phase of the transformation, that is when we need to map the edges of the source to the destination we need to know which vertex in the destination corresponds to which vertex in the source. This is the problem of maintaining references between the two models. There are other examples where the referencing is not that easy, for example, consider a transformation that takes a cross product of a set of vertices to generate a new set of vertices. Then two vertices in the source actually should reference each destination vertex. Hence we need a method to specify and used models of different domains as well as references and other temporary objects.

Thus we needed a way to keep the models from different domains different and still be able to define temporary vertices and edges that belonged to the transformation and could possibly be incident on or adjacent to the source and/or destination models.

The solution to the problem is to use the source and destination meta-models to specify the temporary vertices and edges. This creates a unified meta-model along with the temporary objects. The advantage of this approach is that we can then treat the source model, destination model and temporary objects as a single graph and then be able to use standard graph grammar and transformation techniques to specify the transformation. The rewriting language then uses patterns described above, where each pattern object's type conforms to the unified metamodel and only transformations that do not violate the metamodel are allowed. At the end of the transformation the temporary objects are removed that the two models again conform exactly to their respective meta-models. The transformation language is inspired by many previous efforts such [9][10][11][19][20]. A production is defined to be a pattern that consists of pattern vertices and edges. These pattern objects each conform to a type from the metamodel. Apart from this each pattern has another attribute that specifies the role it plays in the transformation. There are three different roles that a pattern can play. They are:

1. Bind – used to match objects in the graph.
2. Delete – also used to match objects in the graph but after these objects are matches they are deleted from the graph.
3. New – used to create objects after the pattern is matched

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. 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. An example for such a constraint is: "an attribute of a particular vertex should be within limits." 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 object, this need the "attribute mapping". The formal definition of a production is as follows. A production p is a triple (pattern graph, guard, attribute mapping), where

1. Pattern graph is a pair (Pv, Pe), where $Pv \subseteq PV$ the set of all pattern vertices and $Pe \subseteq PE$ the set of all pattern edges.
2. Guard is a set of expressions that operate on the vertex and edge attributes and evaluate to either true of false. If the guard is false, then the production will not execute any operations.
3. Attribute mapping is a set of assignment statements that specify values for attributes and can use values of other edge and vertex attributes.

In Figure 9 describes an algorithm that implements the rule. The algorithm calls the pattern matcher described in

Appendix 1 and 2. The "Effector" function performs deletion and creation of objects and is 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 9 Algorithm for rule execution

3.3.1 Language Realization

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

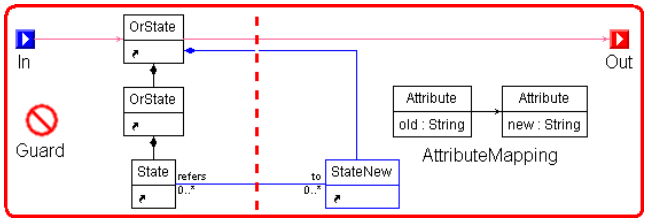


Figure 10 An example rule with patterns, guards and attribute mapping

To maintain consistency and provide usability in GreAT, the following usage method is defined:

1. The user first attaches input and output metamodels of the models to transform in the form of libraries.
2. Then the users specify another metamodel that defines all the temporary vertices and edges that he/she will need for the transformation.
3. After attaching and specifying these metamodels the user can then draw productions/rules that specify patterns. Each object in the pattern refers to a particular metamodel entity. 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.

Thus, GReAT uses UML metamodels as the basic entities for defining patterns. Furthermore, the patterns are also specified in UML syntax and since the modeler uses UML for metamodeling, it is 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.

3.4 Controlled Graph Rewriting and Transformation

In order to increase efficiency and effectiveness of the transformation language it is essential to have efficient implementations for the productions. The pattern matcher being the most time consuming operation needs to be made as effective as possible. In order to make the search algorithm less time consuming the pattern is not searched in the entire graph but is searched within a context. The context is specified by an initial set of bindings for some pattern vertices and edges. This helps to greatly reduce the time complexity of the search. This initial set of bindings is established by using Port objects in the rewriting rules that form the interface of the rewriting rule.

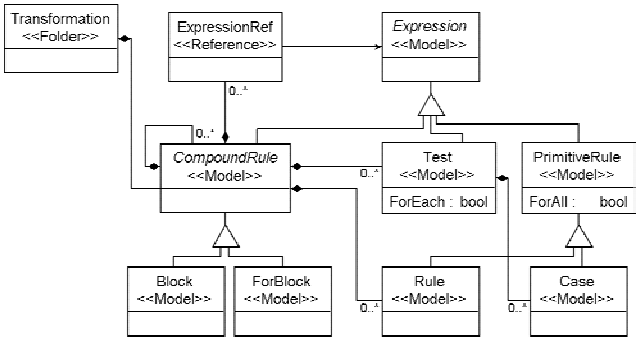


Figure 11 UML class diagram of the expression hierarchy

The next concern is the application order of rewriting productions. Classical graph grammars apply any production that is feasible. This technique is good for generating and matching languages but model-to-model transformations need to follow an algorithm that requires a more strict control over the execution sequence of rules. Furthermore, by specifying a rule execution sequence the implementation can be made more efficient.

In order to provide manageability and mitigation of complexity it is important to have higher-level constructs, like hierarchy of rules in the graph rewriting language. For this reason, we allow nesting of rules and control structures. This latter feature allows modularization and abstraction through the encapsulation of algorithms in blocks. The common base abstraction for the language is "Expression", as shown Figure 11, and all other constructs like Rules and Blocks are derived from it. The derivation implies a shared base semantics: these constructs represent graph transformations.

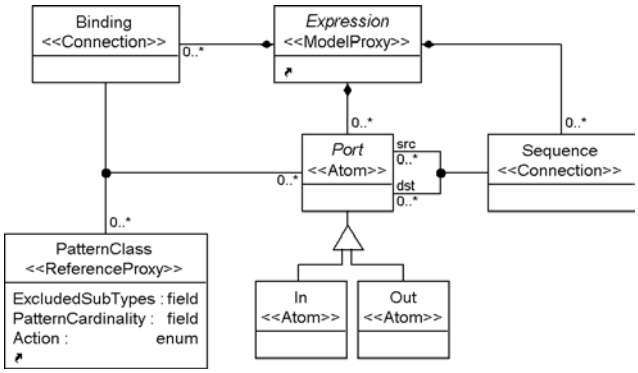


Figure 12: UML class diagram of the expression interface

Figure 11 and Figure 12 show the expression hierarchy in the controlled graph rewriting language, and the input-output interfaces available on the expressions.

Each expression has the same interface and it allows the outputs of one expression to be the input of another expression, in a dataflow-like manner. This is used to sequence expression and allow the expression to be used as black boxes.

A compound rule can contain other compound rules, tests and primitive rules. The primitive rules of the language are to express primitive transformations. A test is a special expression and is used to choose different paths for control flow. Figure 13 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 13 The expression execution algorithm

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

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

3.4.1 Sequencing of Rules

If a rule is coupled to another rule they will execute sequentially. Thus, in Figure 14 rule 1 will fire first to consume all its tokens and produce a number of output tokens. Then rule 2 will fire to consume all its input tokens to produce a number of output tokens.

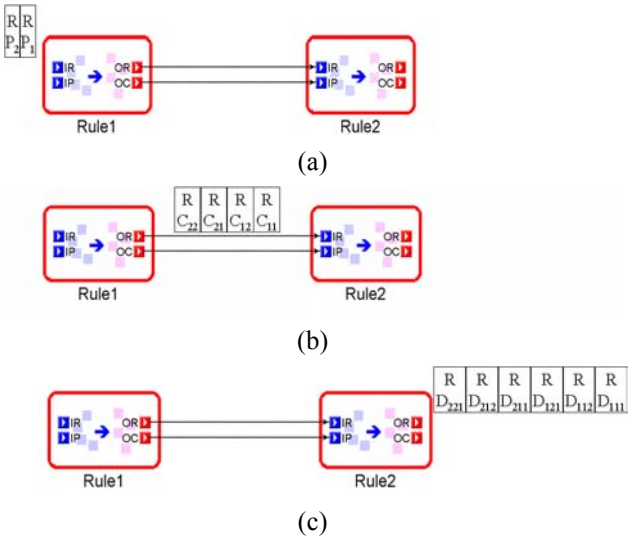


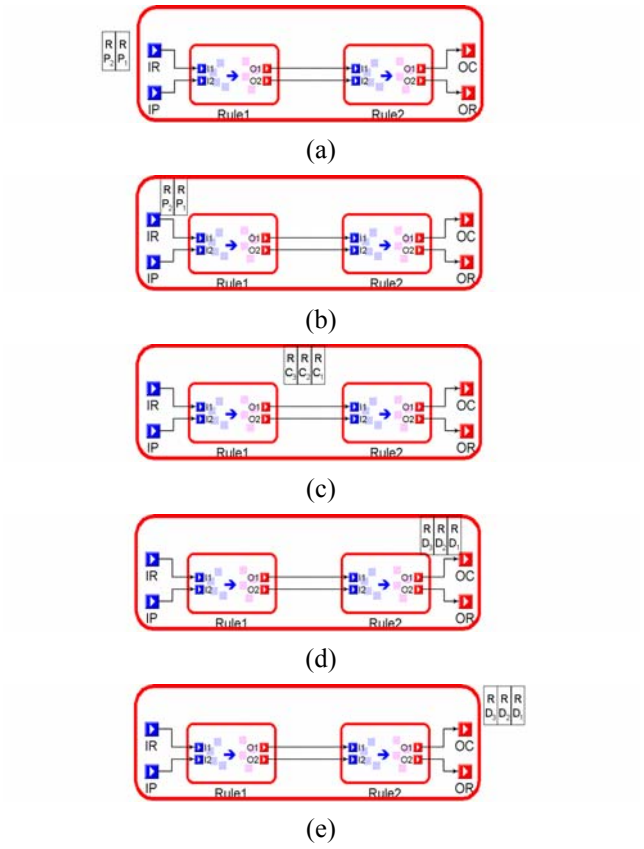
Figure 14 Firing of a sequence of 2 rules

3.4.2 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
    For Each 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 15 Block execution algorithm

There are two types of hierarchical container rules: (1) Block, and (2) For Block. Both *Block* and *For Block* have the same semantics with respect to rules connected to and from it. Thus if in Figure 14 the rules 1 and 2 were hierarchical even then they would have has the same action as described there. Only the semantics within a hierarchical rule differs.

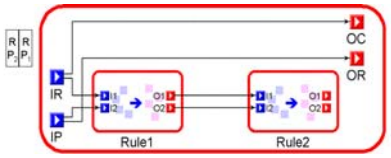


(a) (b) (c) (d) (e)

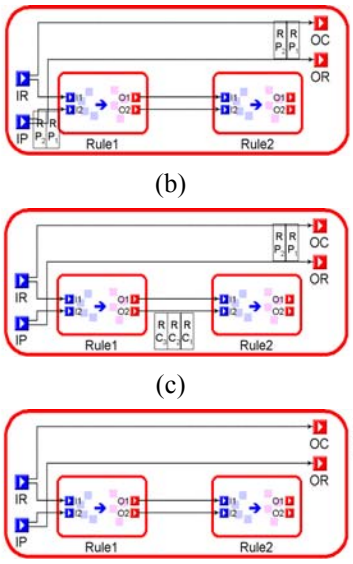
Figure 16 Rule execution of a Block

A block is a container that encapsulates a number of rules. 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 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 16 illustrates the execution of rules within a block.

Figure 17 illustrates the case when the output interface of a block is connected to the input interface of the same block.



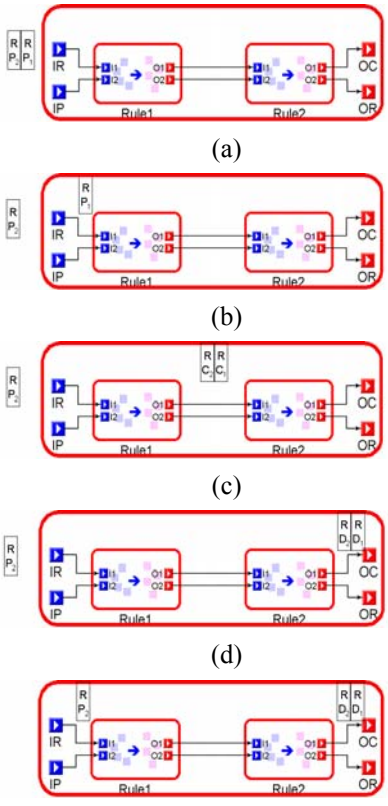
(a)



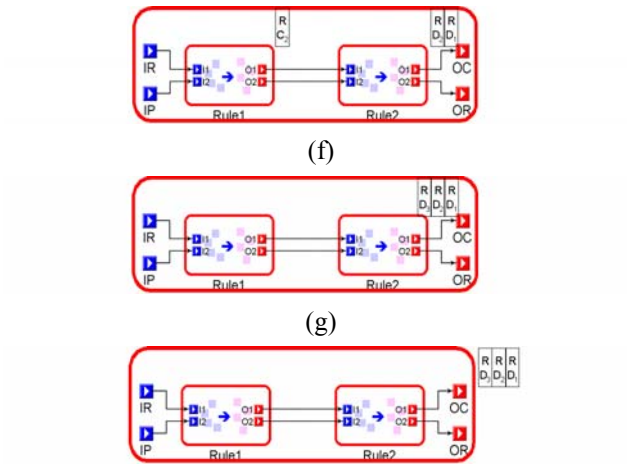
(b) (c) (d)

Figure 17 Sequence of execution within a block

The "For Block" has different semantics for execution within the block. If we have n incoming tokens in a "For Bock" then the first packet will be pushed through thru 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 18.



(a) (b) (c) (d) (e)



(f)

(g)

(h)

Figure 18 Rule execution sequence of a "for block"

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

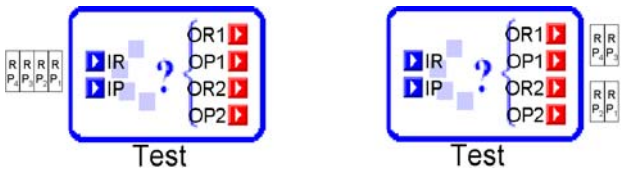
```
Function Name   : ExecuteForBlock
Inputs          :   1. List of Packects inputs
                    2. Expression forblock
Outputs         :   1. List of Packects outputs
outputs = ExecuteForBlock(forblock, inputs)
{   List of Packects outputs
    for each input in inputs
    {   returns = ExecuteBlock(forblock,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

Figure 19 For block execution algorithm

3.4.3 Branching using test case

There are many scenarios where the transformation to be applied is conditional and a "branching" construct is required. We support a branching construct called test/case.

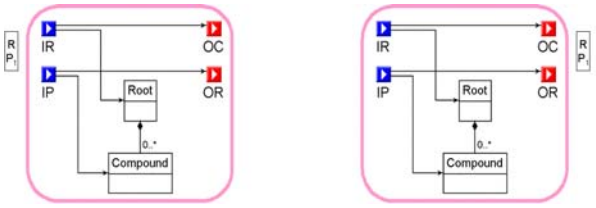
The external semantics of a test/case is similar to any other rule. When fired or executed it consumes all its input packets to produce some output packets. In Figure 20 a test is shown that has two cases. The Test has one input interface and two output interfaces ({OR1, OP1} and {OR2, OP2}). When the test is fired each incoming packet is tested and placed in the corresponding output interface.



(a) **(b)**

Figure 20 Execution of a test case construct

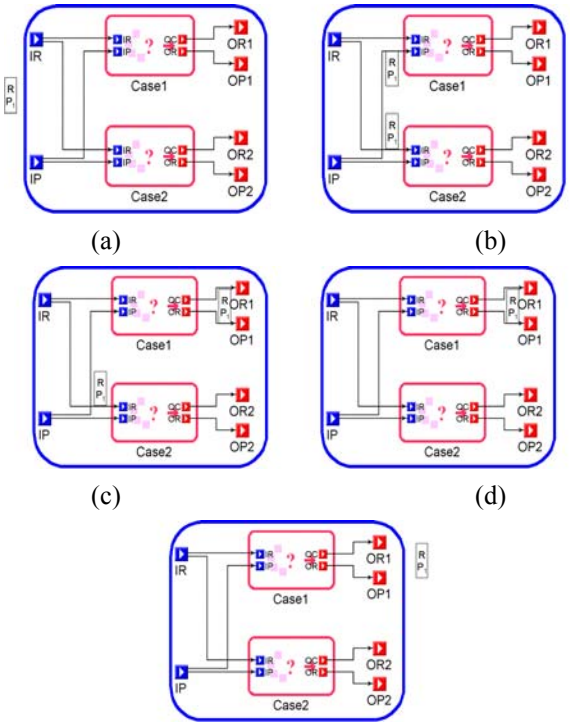
The test must contain at least one case. Each case is a rule with no output pattern and no actions. It contains an LHS pattern a guard condition and an input and output interface. If the LHS pattern has a match then the case succeeds and the input packet to the case is passed along. If the pattern has no matches then the test fails. Also, if the match doesn't satisfy the guard condition, the case fails.



(a) **(b)**

Figure 21: Execution of a case

Figure 21 shows a case with a successful execution. The input packet has a valid match and so the packet it allowed to go forward.



(a) (b)

(c) (d)

(e)

Figure 22 Execution of a test condition

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. 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. The case also has another attribute called the cut. When enabled, it means

that if the case succeeds for a given packet then the packet should not be tested with the remaining cases.

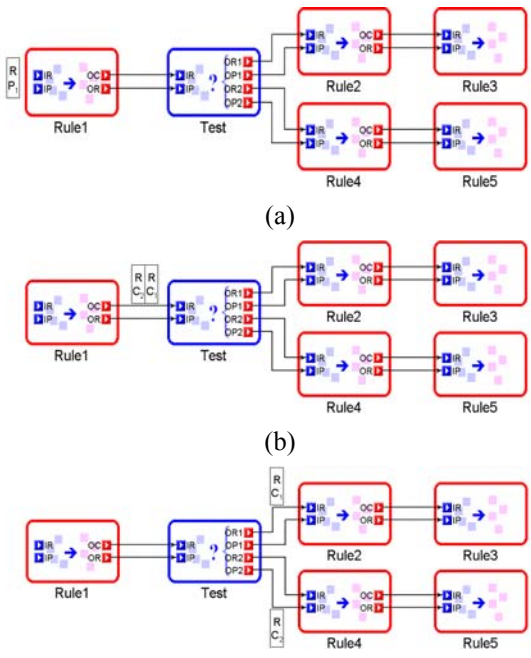
In Figure 22 the execution of a test is shown. An input packet is replicated for each case. Then the input packet is tried with the first case, it succeeds and is copied to the output of the case. Then the packet is tried with the second case, this time it fails and the packet is removed. Finally after 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 returrns exist)
                break
        }
    }
    return outputs
}
```

Figure 23 Test execution algorithm

3.4.4 Non-deterministic Execution

When a rule is connected to more than one following rule or when there is a test condition with more than one path then it is called non-deterministic execution. The non-deterministic execution semantics is defined such that any of the different paths can be chosen for execution first. Once a path is chosen it is executed completely before the next path is chosen.



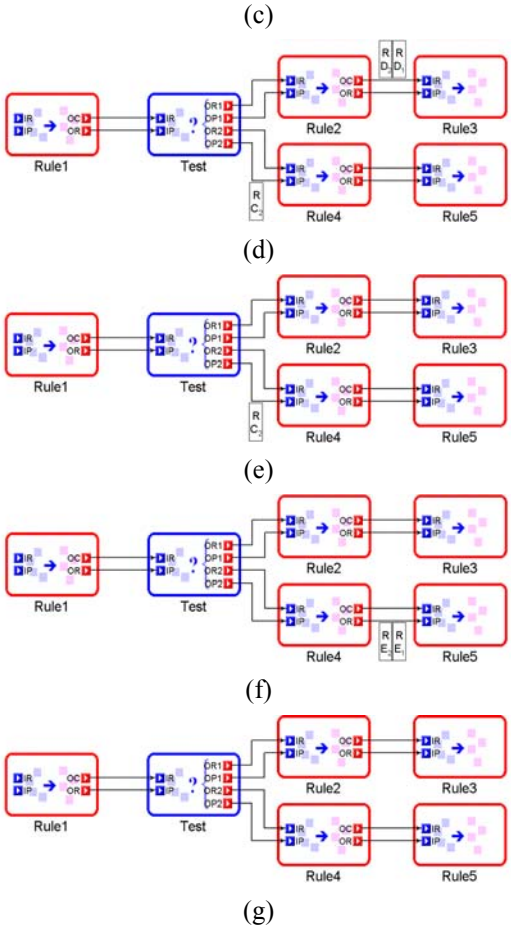


Figure 24 A non-deterministic execution sequence

Figure 24 shows a non-deterministic execution sequence. Here the non-deterministic execution is caused because of a test condition 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, in this case rule 2 is chosen and fired, followed by the execution of following rules. This ends at rule 3. Then rule 4 & 5 are fired.

3.4.5 Termination

Finally termination of a transformation needs to be discussed. A rule sequence is terminated either when a rule has no output interface or when a rule having an output interface does not producing any output packets.

Thus if the firing of a rule produces zero output packets then the rules following it will not be executed. Hence in Figure 24 if rule 4 produced zero output packets then rule 5 would not have been fired. However, the there should be a construct to sequence rules without having to bind the ports.

4. The Implementation

The language is currently implemented using an interpreter. This interpreter is supplied with the transformation rules and the starting input packets.

4.1.1 The UDM Package

The technology used for the implementation of GReAT is Universal Data Model (UDM) [22]. The Universal Data Model (UDM) is a meta-programmable package [22] that includes a development process and a set of supporting tools to generate C++ accessible interfaces from UML class diagrams of data structures. The generated APIs can use a variety of data storage formats such as XML, GME model databases and memory-based objects. The data storage format is transparent to the user and the same API can be used to access and store data to any 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 (Visio or GME). The UML class diagram is then converted into an XML representation with the help of a UDM tool.
- The XML file is then used to generate a C++ API (pair 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 UDM library and one of the implementation specific UDM libraries. The user can easily create, modify and traverse object graphs described by the class diagram.
- Alternatively, the generated XML file can be directly used to create, modify and traverse object graphs corresponding to the particular metamodel using the generic UDM API.

The Tool chain in the UDM process is described below. Figure 25 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 explicit in the form of initialized data structures.

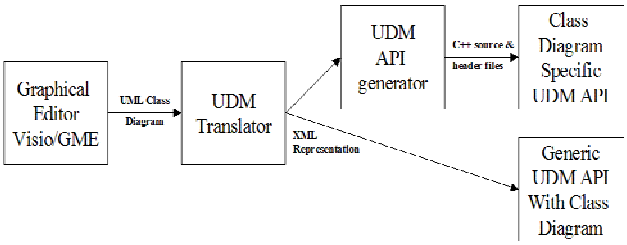


Figure 25 Tool chain for generation of UDM API

The GreAT interpreter is an experimental testbed developed for testing the transformation language and to validate that the language is powerful enough to express most common transformation problems. The interpreter takes the input graph, applies the transformations to it, and generates the output graph. Inputs to the GreAT interpreter are (1) the UML class diagrams for the input and output graphs (also known as meta-models), (2) the transformation specification and (3) the input graph. The GRE traverses the rules according to the sequencing and produces an output graph based upon the actions of the rules.

The architecture of the run time system is shown in Figure 26. 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 their own language format and can be accessed using the language specific UDM API.

The GRE 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 for each rule it calls the ExecuteRule. The rule executor internally calls the PM with the LHS 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 Appendix 1 & 2.

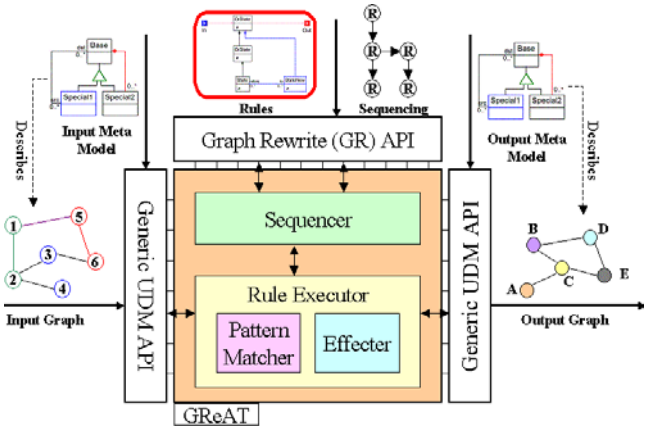


Figure 26 The GReAT interpreter

5. Examples and Results

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

1. 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 Cartesian production of the parallel state space. The evaluation of the this particular transformation requires a depth-first, bottom up approach and will also test whether the system can allow different traversal schemes.
2. Reachability analysis and deletion of unreachable states is the next challenge problem. This problem is chosen to check weather the language can be used to express and implement algorithms seamlessly.
3. The next example is to generate the equivalent Hybrid System from a given Matlab Simulink and 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 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.
4. The final example is to build a pre-processor for domain specific extensions to a language. The problem is to build a pre-processor that will convert 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].

Table 1 shows the examples that have been implemented using GReAT and the lines of code required to hand code them. The ratio between hand code and the number of rules is between 1:10 and 1:30. Thus, we see that GReAT can be used for significant speed up in the development time of model-to-model interpreters.

Table 1 Comparison of GReAT implementation vs code

Problem	GReAT		Hand Code
	Primitive Rules #	Compound Rules #	LOC
Mark and sweep algorithm on Finite State Machine (FSM)	7	2	100
Hierarchical Data Flow (HDF) to Flat Data Flow (FDF)	11	3	200
Hierarchical Concurrent State Machine (HCSM) to Finite State Machine (FSM)	21	5	500
Matlab Simulink/ Stateflow to Hybrid System	25	9	1000

5.1 HCSM to FSM example

The algorithm used to generate an equivalent FSM from a HCSM and how it is defined in GReAT is described in this section. The algorithm is a depth first bottom up algorithm. The top-most rule is a recursive rule that takes as input either an or-state, and-state or a simple state. The rule then tests to find out the type of the input. If the input is an-and state it passes the input to a sub rule that flattens the and-state, if the input is an or-state is then passes it to another sub rule that deals with the flattening the or state and if the input is a simple state then the rule simply returns the state.

For the sake of simplicity and clarity let us consider that the state machine has only or-states and thus only the rule with or-state is described here. The rules for flattening the

or-state first call the top-level rule on all the states contained in the or-state.

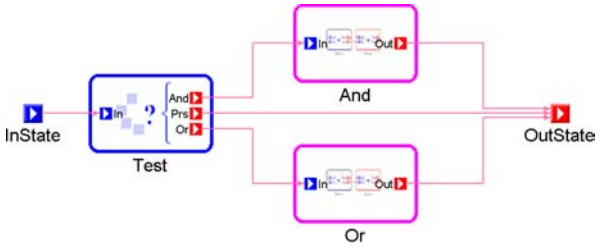


Figure 27 The top-level rule

It then proceeds with flattening all the or-states within it. Thus the algorithm will first act upon an or-state that only contains simple states or other or-states that contain simple states. At this state the next rule is to elevate all the states contained in the child or-states. Let the or-state being flattened be Or1 and let it have child or-states named Or11 to Or1n. Then for each Or1x, where $1 \le x \le n$ and for each child state of Or1x there will be a new state created as the child of Or1. The next rule maps the init transition of Or1 to an or-state to the correct child of the or-state. After this the next rule creates a corresponding transition for each transition that existed within the child or-states Or11 to Or1n. The next rule then creates the transitions that existed between Or1x states within Or1 and creates the corresponding transition between the elevated states. The final rule then deletes the Or11 to Or1n.

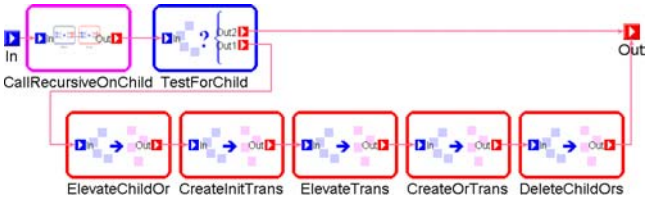


Figure 28 The or-state rule

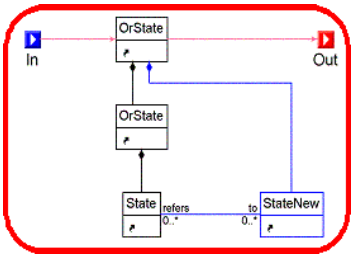


Figure 29 The elevation rule

6. Conclusion and Future Work

This paper has shown a technique for model transformations based on graph transformations. The transformations are represented in the form of explicitly sequenced transformation steps, which use graph patterns and actions like *new*, *bind*, and *delete* to capture an elementary action. The transformation models are tightly coupled to the concepts of UML, and are based on the notion that UML class diagrams define meta-models that are graph grammars for the graph of objects. We have shown the syntax and semantics of the graph transformation language, and its implementation and illustrated its use.

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. The execution engine is not efficient, and we need to develop a technique for generating executable code from the transformation programs, in order to be competitive with other approaches. From practical experience we learned that there is a need for a debugging tool that allows the developer the tracking of the execution of the transformations. We plan to address these issues in further research.

7. Acknowledgements

The DARPA/IXO MOBIES program and USAF/AFRL has supported under contract F30602-00-1-0580, 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.

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)
{
   for each 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
      For each 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.
        for each 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.
            For each 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
            reutrn enpty list.
}
```