The Design of a Language for Model Transformations

ADITYA AGRAWAL, GABOR KARSAI, ZSOLT KALMAR, SANDEEP NEEMA, FENG SHI, ATTILA VIZHANYO

Institute for Software Integrated Systems, Vanderbilt University

Nashville, TN 37235, USA.

Email: {aditya.agrawal, gabor.karsai, zsolt.kalmar, sandeep.neema, feng.shi, attila.vizhanyo}@vanderbilt.edu

ABSTRACT. Model-driven development of software systems envisions transformations applied in various stages of the development process. Similarly, the use of domain-specific languages also necessitates transformations that map domain-specific constructs into the constructs of an underlying programming language. Thus, in these cases, the writing of transformation tools becomes a first-class activity of the software engineer. This paper introduces a language that was designed to support implementing highly efficient transformation programs that perform model-to-model or model-to-code translations. The language uses the concepts of graph transformations and metamodeling, and is supported by a suite of tools that allow the rapid prototyping and realization of transformation tools.

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

1. Introduction

The model driven development of systems [34] 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 [2][36][41][8]. Many tasks in software development have been formulated using this approach, including weaving of aspect-oriented programs [3], application of design patterns [41], and the transformation of platform-independent models into platform specific models [1]. A special class of transformations arises in Model Integrated Computing (MIC) [1]. MIC is an approach in which domain-specific modeling languages and

generator tools are developed and then the domain-specific language is used for creating, analyzing, and evolving the system (or a product-line of systems) through modeling and generation. During the last decade, MIC has gained acceptance through several fielded systems [23][34], 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. Model transformation tools are essential to MIC: they 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 [34].
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 sublanguage, a graph transformation sublanguage and a high-level control flow sublanguage and has been designed to address 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, and Section 6 describes comparison with other, similar systems. Section 7 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 functionality, and are used to synthesize executable systems, perform analysis or configure simulators. 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 resulting metamodel of the domain is used to generate a Domain-specific Design Environment (DSDE), which is then used to build domain-specific models. 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 (analysis) model can be in the form of Petri-nets, suitable for predicting the performance of the network. Note that the result of the transformation is just another model that conforms to a different metamodel: the metamodel of the target domain [1].

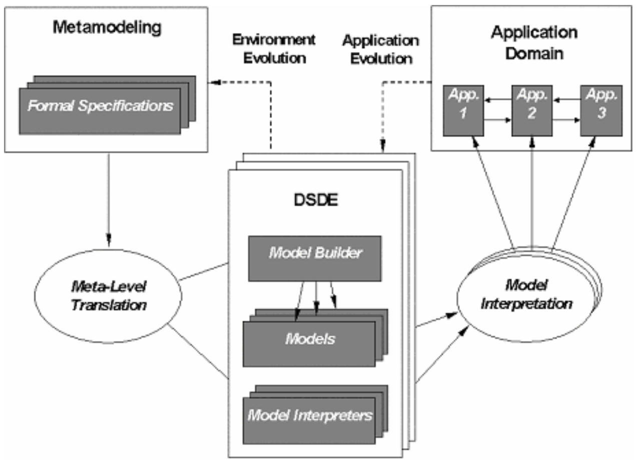


Figure 1 The MIC Development Cycle [1]

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 [27], DOME [51], Moses [36], Metaedit[27], and GME [1]. Each implementation has a metamodeling layer that allows the specification of a domain-specific modeling languages and a modeling layer that supports the construction 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 [1]. An important distinguishing property of the metamodeling environment of GME is that it is based on UML class diagrams [34]; an industry standard, which are used in GME to describe domain-specific modeling languages and their corresponding modeling environment by capturing the syntax, semantics and visualization rules of the target domain. The abstract syntax is captured using in UML class diagrams, the visualization techniques through the use of stereotypes, and the static semantics (i.e. the well-formedness constraints) using OCL expressions. A tool called the meta-interpreter verifies and translates the metamodels and constructs 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 GME is used both as the metamodeling environment *and* the domain-modeling environment; the metamodeling language is just another domain-specific language that the common editing engine supports.

While GME is equipped with a meta-interpreter, until recently there were no generic, high-level tools to assist in the construction of domain-specific model transformers. Each model transformer had to be hand-coded in an imperative programming language: a time consuming and error-prone activity. 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 has gained attention recently with the advent of the Model Driven Architecture (MDA) by Object Management Group (OMG) [34]. MIC can be considered as a particular manifestation of MDA, which is tailored towards system construction via domain-specific modeling languages [1].

Recent efforts (described in [23]) indicate the widespread interest in MIC-related approaches.

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: the input and the output languages, 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 [46][8][22][34][2][48]. 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 but 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 [46], while in AGG these are represented using type graphs [52].

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 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 of 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 an edge in the target set for each vertex and edge in the source set:

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

(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 simple, 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 are often hard to use for the specification and implementation of model transformers as described. Hence, new approaches are needed that target the specific needs of model-to-model transformation. A novel 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-links between graph domains should be supported through well-formed, preferably graphical language constructs.
- The language should have efficient implementations. The 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 a pragmatic 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 recent papers have shown how graph transformation techniques can be used for (1) specification of program transformations [2], (2) defining the semantics of hierarchical state machines [36], (3) tool support for design patterns [41], and (4) tool integration [8]. Other recent work [32] [11] shows how model transformation can be implemented using graph transformation techniques, and illustrates how interesting properties, like termination, consistency, confluence, etc. can be proven using existing results. Our goal is that the language be able to implement the ideas presented in these papers.

3. A Language for Graph Rewriting and Transformations: GReAT

The transformation language we have developed to address the needs discussed above is called GReAT, short for “Graph Rewriting and Transformation language”.

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.

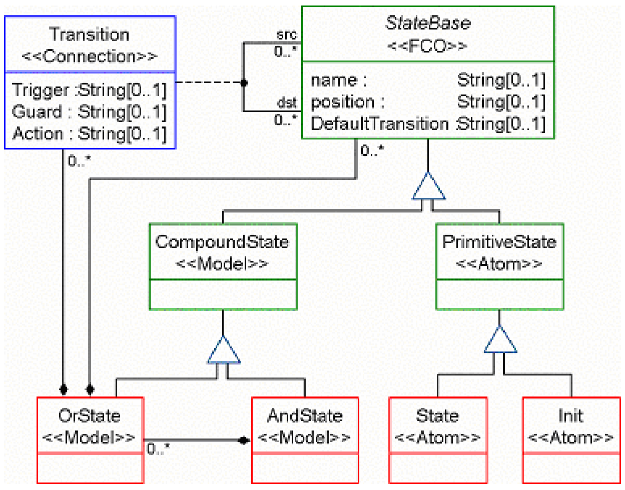


Figure 2 Metamodel of Hierarchical Concurrent State machine using UML class diagrams

In model-to-model transformations the input and output graphs are object networks whose “schema” can be represented using UML class diagrams and constraint expressions in the Object Constraint Language (OCL) [41]. 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 2 shows a UML class diagram that represents the domain of Hierarchical Concurrent State

Machines (HCSM) and Figure 3 shows the metamodel of a simple Finite State Machine (FSM).

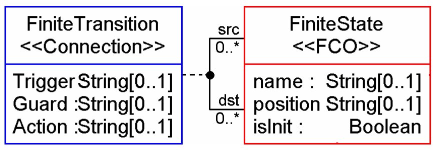


Figure 3 Metamodel of a simple finite state machine

Note that one domain is typically described by multiple UML class diagrams, and classes with the same name (but different semantics) may appear in different domains. 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 4 shows a metamodel that defines associations between classes from 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 associates *States* and *FiniteStates* and *Transitions* and *FiniteTransitions.* Another edge type *associatedWith* is defined and it links *State* objects.

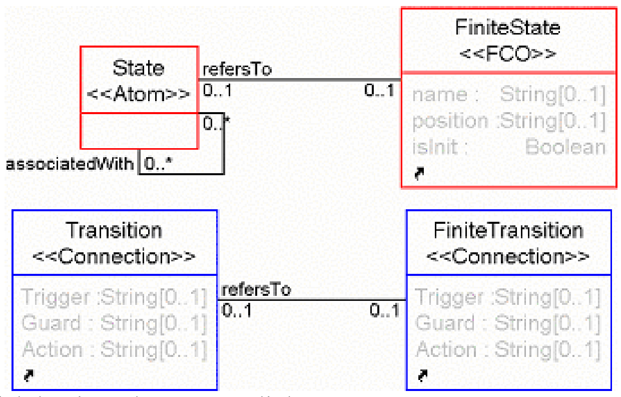


Figure 4 A metamodel that introduces cross-links

Cross-links can be defined not only between different domains but can also be used to extend a specific 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. Note that this approach is related to the techniques used in triple-graph grammars [49], where explicit mappings between domain elements are specified.

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. We assume that for each graph there is a UML class diagram that defines classes and associations that act as "types" for vertices and edges, respectively. Classes could define attributes as pairs of names and attribute types.
A *vertex V* is a 3-tuple *(class, id, attrs)*, where *class* is a UML class, *id* is a unique label, and *attrs* is a map that maps each attribute (defined in the class) into a value. For convenience, we define the function *vtype(V)* that returns the *class* of *V*. We also define the Boolean-valued type-compatibility function *tcomp(c1, c2)* that returns true if class *c1* is identical to or a subclass of class *c2*. If *tcomp(c1,c2)* evaluates to true, then we *c1* and *c2* are said to be type-compatible. We will use this to define the pattern matching for objects that are subtypes of base type pattern elements. This is different from to the approach in [5] that distinguishes abstract and concrete rules.
An *edge E* is a 5-tuple: *(assoc, id, src, dst, attrs)*, where *assoc* is the simple association or association class of a UML class diagram the edge belongs to, *id* is a unique label, and *src* and *dst* are vertices. *attrs* is a map which is non-empty if *assoc* is a UML association class, and it maps each defined attribute of that association class into a value. When *assoc* is an association class the edge must be unique: there can be only one edge of type *assoc* between two participating objects. *Src* and *dst* are the vertices that the edge is incident upon and the type of these vertices must be identical to the endpoint classes of the edge. For convenience, we define the function *etype(E)* that returns the *assoc* of *E*.
A *graph G* is pair (GV, GE), where GV is a set of vertices in the graph and GE is the set of edges, such that $\forall e = (assoc, id, src, dst, attrs) \in GE, src \in GV \wedge dst \in GV$
Note that the metamodel (the UML class diagram) is also a graph, but its metamodel is the model describing the UML language. That, in turn, relies on a

meta-metamodel, which is a subset of the MOF (Meta-Object Facility) of OMG [39], and it's metamodel is itself. Thus, we follow the standard four-layer metamodeling approach.

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.

$$
\begin{aligned}
&\forall EB \in MEB\,, where \\
&EB = (pe, he), pe = (passoc, psrc, pdst, pattrs), he = (hassoc, hsrc, hdst, hattrs) \\
&\bullet\, etype(pe) = etype(he) \wedge \\
&\exists VBS \in MVB \wedge \exists VBD \in MVB, where \\
&VBS = (psrc, hsrc), VBD = (pdst, hdst) \\
&\bullet\, tcomp(vtype(hsrc), vtype(psrc)) \wedge tcomp(vtype(hdst), vtype(pdst))
\end{aligned}
$$

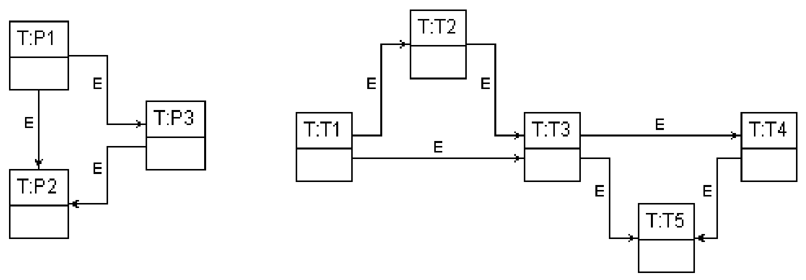
The match doesn't have a restriction that would specify that each pattern object must have a binding. This is intentional, as the match is also used to specify partial matching of pattern graphs. Note that a host graph vertex matches a pattern a vertex if they are type compatible. Note also that a match is injective: one pattern element maps onto one host graph element, and we disallow non-injective matches (i.e. one host graph element can be bound to at most one pattern element). However, the collections of matches are not injective: the same host graph element can appear in multiple matches, and thus the identity condition does not apply across matches, only within one match.

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 [46][8][22][9][9][48][9] 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. In other words, an object graph is correct with respect to a metamodel if (1) there is a morphism between the metamodel elements and the object graph, and (2) all well-formedness rules (constraints) evaluate to true over the object graph.

3.3.1. Simple Patterns

A simple pattern is one in which the pattern represents the exact subgraph. 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 of the matching 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), where vertices are labeled as *C:N*, *C* being a class name and *N* being an instance name. The figure describes a pattern that has three vertices P1, P2 and P3, each of type. 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. Naturally, 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.

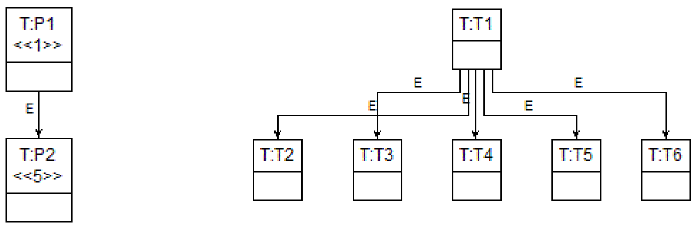
Returning all the matches however, has a time complexity of $O(C_h^{C_p})$, where C_h is the number of host vertices and C_p is the number pattern vertices. To make the pattern matching usable we need to optimize it. One approach is to start the pattern matcher with an initial context. By context we mean an initial partial match that the pattern matcher is started with. 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. Note that the algorithm works connected graphs only, unlike algorithms used in more sophisticated tools [52][41]. 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

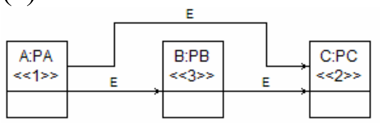
The same argument holds for graphs, and a similar technique can be used. The pattern vertex definition can be changed to a pair (class, cardinality), where cardinality is an integer. Vertex binding can also be redefined as a pair (PV, HVS), where PV is a pattern vertex and HVS is a set of host vertices. 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 vertex bindings in the match are {(P1,T1), (P2,{T2, T3, T4, T5, T6})}, and there is one edge binding between the pattern edge and all the edges in the host graph.

The fixed cardinality pattern matching also exhibits non-determinism. However, even in this case the issue can be dealt with by returning all the possible matches. If all the possible matches are returned the resulting set could be quite large. 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.

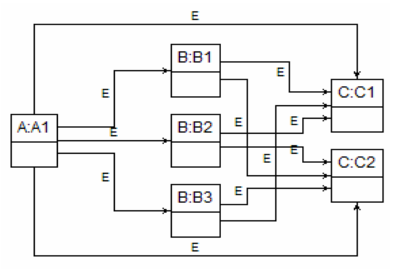
A more immediate concern is how this notion of cardinality truly extends to graphs. For strings we have the advantage of a strict ordering from left to right, while for graphs we don't. For instance, extending the example in Figure 6 with another pattern vertex will result in an ambiguous specification.

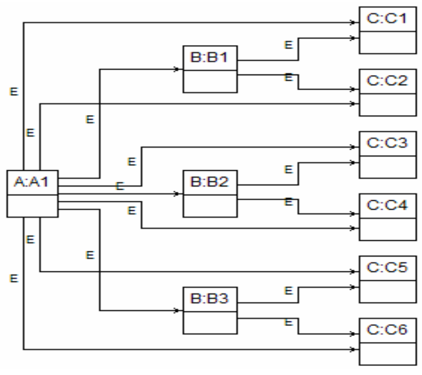
In Figure 7(a) we show a pattern having three vertices. There are different interpretations (or semantics) that can be associated with the pattern. One possible semantics is to consider each pattern vertex *pv* to have a set of matches equalling the cardinality of the vertex. Then an edge between two pattern vertices *pv1* and *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 semantics, when used with the pattern in Figure 7(a), gives the graph in Figure 7(b).

(a) Pattern with three vertices



(b) Result with set semantics





(c) Result with 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 each vertex bound to *pv1* will be adjacent to *c2* vertices bound to *pv2*. Let *b1 = (pv1,V1)* and *b2 = (pv2,V2)* be the bindings for *pv1* and *pv2* respectively. Then

Relation 1

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

This semantics, when applied with the pattern gives Figure 7(c). The tree semantics 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.

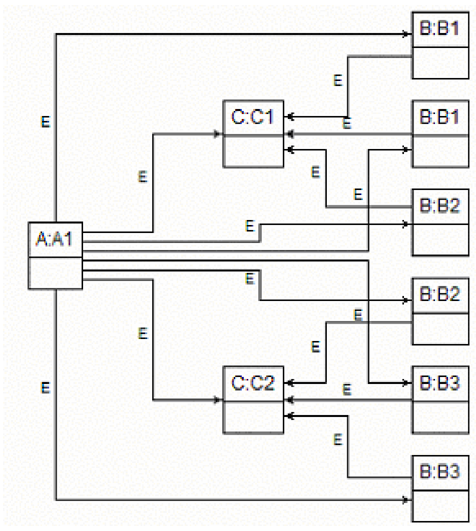


Figure 8 Conflicting match for the tree semantics

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.

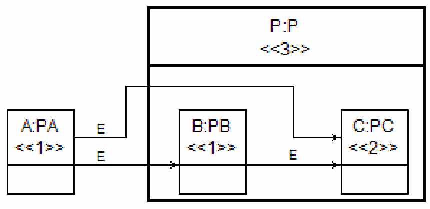
Fortunately, there is a pragmatic solution that solves all the problems: to use a more expressive, extended set notation.

3.3.3. Extending the Set Semantics

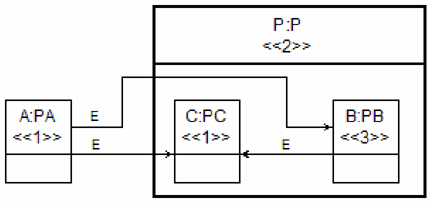
As an example, consider the use of regular expressions to represent strings. For example, in a string “sxyxyxy” “xy” is repeated 3 times. Using a notation mentioned previously we would express it as “s3(xy)”. Using parenthesis we were able to represent the fact that the “xy” sequence should occur 3 times. 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 subpattern and then a larger pattern can be constructed using these subpatterns as vertices. If a group consists of a subgraph and has the cardinality *n* then the *n* subgraph 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 that 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 subpatterns 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.

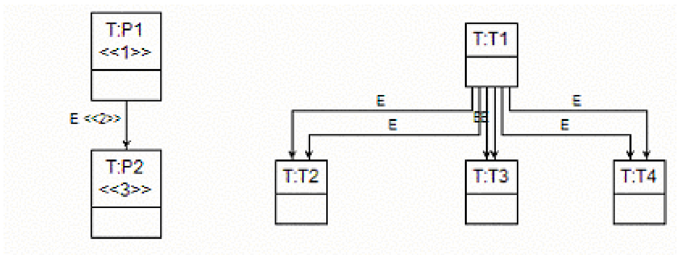
3.3.4. Cardinality For Edges

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 semantics is an extension of Relation 1. Let b1=(pv1,V1) and b2=(pv2,V2). Then

Relation 2

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

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.

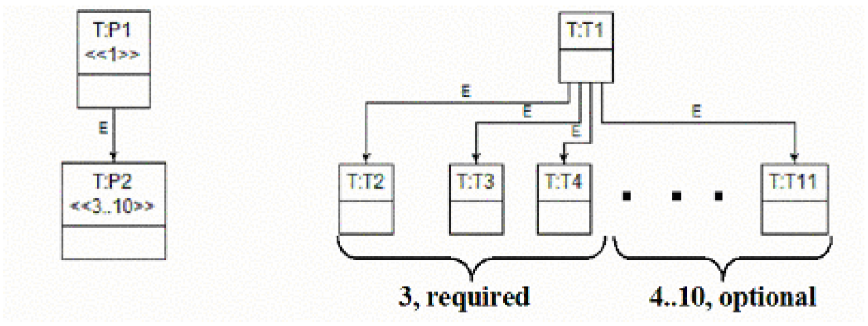


(a) Pattern (b) Matching Host graph

Figure 10 Pattern with cardinality on edge

3.3.5. Variable Cardinality

Sometimes, the subgraph to be matched is not fixed but is a member 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. This family of strings 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).



(a) Pattern (b) Family of graphs

Figure 11 Variable cardinality pattern and family of graphs

Figure 11 shows 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 consider 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 subgraphs with cardinality 3, 4 up to cardinality 10. The way we solve this problem is as follows: if more than one match is produced, then a match will be returned only if it is not a proper subgraph of another, larger match in the same set of matches. A match *m1* is larger than a match *m2* if it, *m1*, contains more elements of the host graph than *m2*. Thus the matches returned would each be maximal and consistent with respect to the pattern. This construction yields a precise and consistent language, which can be used to specify complex patterns in a concise manner. We conjecture that the language is powerful enough to express application conditions as described in [24].

3.3.6. Pattern Graph and Match Definition

After the discussion on the specification of patterns we can now generalize the definitions for pattern vertices, edges and graphs with cardinalities. We note that the pattern matching concept for nested subpatterns with multiplicities introduced here is similar to the one described in [52].
A pattern vertex *PV* is a pair: *(class, card)*, where class is a UML class defined in the heterogeneous metamodel and *card* is a pair of *(lower, upper)*. The function *lower(pv)* applied to a pattern vertex *pv* returns the lower field of *card* of that *pv*, and the function *upper(pv)* is similarly defined. For a set *S #(S)* means the cardinality of that set. A pattern edge *PE* is a 4-tuple *(assoc, src, dst, card)*, where *assoc* is the simple association or association class the edge belongs to, and *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 *assoc*. 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, such
that $\forall pe = (assoc, src, dst, card) \in GPE, src \in GPV \wedge dst \in GPV$.

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 *(MVB,MEB)* must satisfy the following properties:

$$
\begin{aligned}
&\forall EB \in MEB, where \\
&EB = (pe, HE), pe = (pclass, psrc, pdst, (lower, upper)): \\
&lower \leq \#(HE) \leq upper \\
&\bullet \forall he \in HE, where \\
&he = (hassoc, hsrc, hdst): \\
&etype(pe) = etype(he) \wedge \\
&\exists VBS \in MVB \wedge VBD \in MVB, where \\
&VBS = (psrc, HSRC), VBD = (pdst, HDST): \\
&\bullet\, hsrc \in HSRC \wedge hdst \in HDST \wedge \\
&lower(psrc) \leq \#(HSRC) \leq upper(psrc) \wedge lower(pdst) \leq \#(HDST) \leq upper(pdst) \wedge \\
&\forall hs \in HSRC \wedge hd \in HDST: \\
&tcomp(vtype(hs), vtype(psrc)) \wedge tcomp(vtype(hd), vtype(pdst))
\end{aligned}
$$

3.4. Graph Rewriting/Transformation Language

The graph transformation language GreAT was inspired by many previous efforts such as [8][22][34][48][9]. The language is built upon the notion of the basic transformation entity: a production (or rule). A production contains a pattern graph where the pattern objects each conform to a type: class or association from the metamodel. Additionally, 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. The *delete* operation deletes the object, as well as the links incident upon it, similarly to the Single Pushout approach [46]. Since the pattern matcher returns all matches for the pattern, it is possible that matches overlap, and there can be a case where a host graph object is deleted from a match while a subsequent 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 could be described using Object Constraint Language (OCL) [41], as it is a widely used standard and is directly related to UML: the basis for metamodeling in GME. If a match returns multiple vertices (edges) for a pattern vertex (edge) then the value of a pattern variable will be a container (in the OCL sense), and thus the expression has to be written accordingly. 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". Because of practical considerations, we have chosen C++ as the implementation language for both guards and the attribute mapping code (although GME has a built-in OCL interpreter).

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

- *Pattern graph* is a graph (defined in Section 3.3.6).
- The *pattern roles* map each pattern vertex and edge to an element of the role set: {*bind, delete, new*}.
- *Guard* is a Boolean-valued expression that operates on the vertex and edge attributes of the matched host graph elements. If the guard is false, then the production will not execute any operations.
- *Attribute mapping* is a set of assignment statements that set values for attributes on new edges and vertices, 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, while (2) maintaining the consistency of the models with respect to their meta-models. Hence, it is important that the language allows the user to construct only rules that conform to the meta-models. As discussed earlier, sometimes it is necessary to construct vertices and edges that do not belong to a metamodel (of the input or the output) hence there is a need for having metamodels for these temporary elements. Therefore, we follow the process below when constructing GReAT transformation programs:

- The user first imports metamodels the source and target models.
- Next, the user constructs a metamodel that defines all the types for the temporary vertices and edges that he/she will need in the transformation. Note that the user may introduce associations between classes belonging to two different metamodels, and can even introduce completely new classes.
- After building these metamodels, the user can construct the productions that are "legal", i.e. compliant with a metamodel.

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 collection of metamodels, and this reference means that the pattern object must match with a graph object that is an instance of the class (or of one of its subclasses) 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.

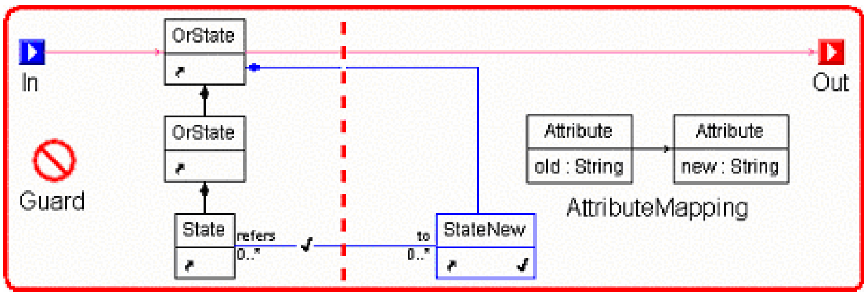


Figure 13 An example rule with patterns, guards and attribute mapping

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 via *ports* that form the input and output interface of the production. If one considers a production a function, then a port is a formal parameter: input parameters (ports) are read during function execution and output parameters (ports) are written to. Before a rule is executed a "values": a host graph node must be provided for each input port. The pattern vertices that are connected to the input ports are bound to these host graph vertices before the actual pattern matching is computed. After rule execution, the binding of those pattern vertices that are connected to output ports is then used to form the output values: again, host graph nodes. These nodes are then passed along to a subsequent rule. In Figure 13 the *In* and *Out* icons are input and output ports respectively. The collection of input (output) ports is called the input (output) interface, respectively. A rule receives a set of bindings: one host node for each port, and produces another set of bindings: one host node for each output port. These sets are called the input and output *packets*, respectively. Thus rules operate on and produce *packets,* which are sets of (port, host graph vertex) pairs. Note that these packets are produced by predecessor rules (as explained below), while for the first rule in the transformation program the programmer has to provide the initial bindings.

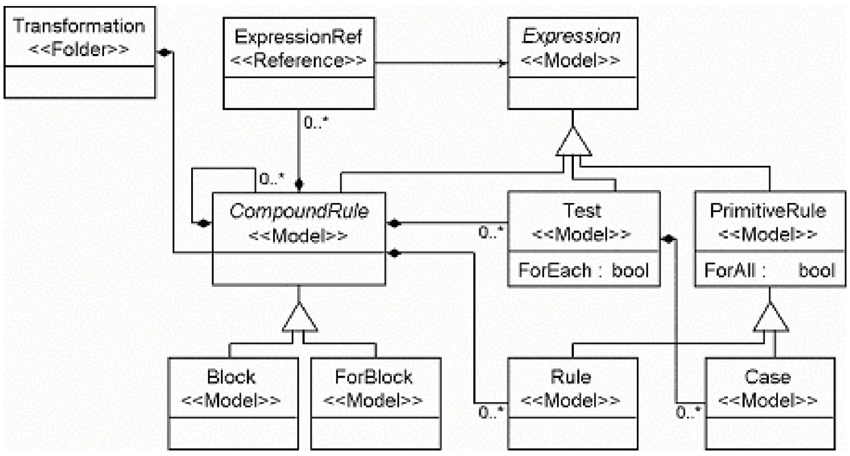


Figure 14: UML class diagram for the core transformation classes GReAT

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 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 constructs 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.

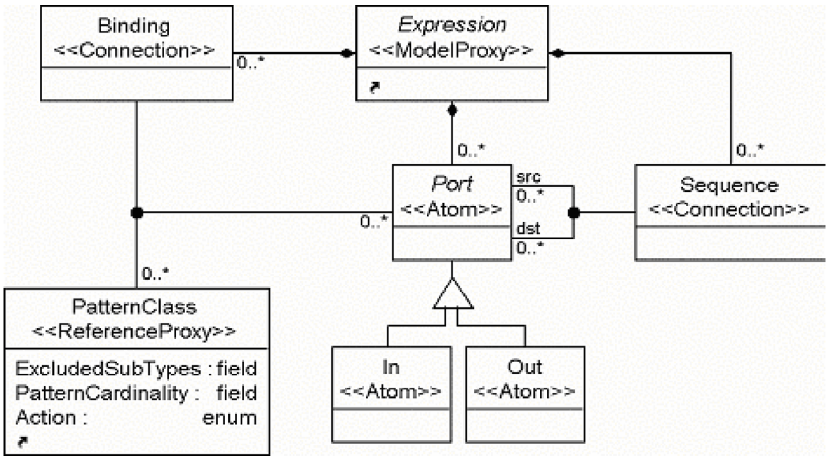


Figure 15: UML class diagram for the abstract syntax classes of GreAT: The interface

A *CompoundRule* may 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 captures a high-level algorithm for rule execution.

```
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

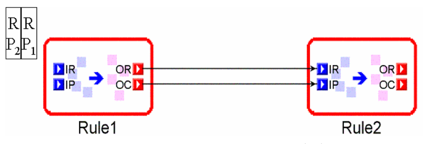
The control flow language has the following basic control flow concepts.

- *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 nondeterministic.
- *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.

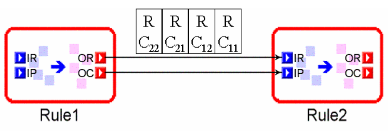
Note that the approach followed here can be considered as a highly specialized version of the transformation unit concepts introduced in [27], and follow the concepts of programmed graph grammars introduced by [52]. The hierarchical constructs can be viewed as graph transformation modules, but in GReAT the control condition is restricted. Also, GreAT does not address the issue of transactions, as all rule execution is assumed to be single-threaded. Furthermore, the rule execution semantics is similar to the execution semantics of asynchronous dataflow graphs and DEVS [54], but with a difference in the hierarchical rule execution, as discussed below. In this sense, the class diagrams Figure 14 and Figure 15 introduce the same concepts as found in DEVS.

3.5.1. Sequencing of Rules

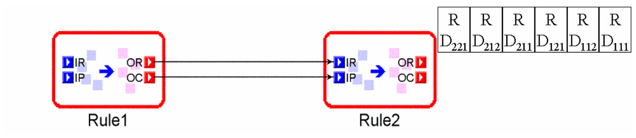
If the output interface (ports) of a rule is connected to the input interface (i.e. the input ports) of another rule, then the execution of the first rule is followed by the execution of the second rule. The connectivity of the rules implies the "flow of packets" from one rule to the next. Figure 17 illustrates this flow of packets through the rules with names inside rules (e.g. IR, IP, etc.) for labeling the ports of the rules. The packets are shown as a vertical group of names where each name refers to a host graph vertex. For instance, (R P_2) forms one packet, (R P_1) forms another one, etc. Packets for the first rule in the transformation program are provided by the top-level configuration, and intermediate packets are produced by rule execution: pattern matching and object creation. Note that one input packet could produce zero, one, or more than one output packets. The last case happens when the pattern matcher delivers multiple matches. The objects within the packets are bound to the corresponding input ports in the vertical layout (i.e. R is bound to IR, P1 is bound to IP, and in the next packet R is bound to IR, and P2 is bound to IP, etc.) Figure 17(a) shows a state during rule execution, where there are two input packets available on the input interface of Rule 1. Rule 1 is executed first: it runs once for each of its input packets. Suppose it produces four output packets as shown in Figure 17(b). Then rule 2 will fire to process all its input packets, and it produces six output packets, as shown in Figure 17(c).



(a)



(b)



(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*. We consider these rules, because from the viewpoint of other rules connected to the containers both *Block* and *ForBlock* have the same semantics: they consume and produce packets as described above. 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
    for each next_rule in block.next_rules()
    {   if(next_rule is_a 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 in current.next_rules()
        {   if(next_rule is_a block)
              outputs.add(inputs )
            else
              ready_rule.Push(next_rule,inputs)
        }
    }
    return outputs
}
```
Figure 18 Block execution algorithm

The *Block* has the following semantics: it will forward all its incoming packets to the first internal rule (i.e. it operates with 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 produce 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 source and it cannot be attached to two different places.

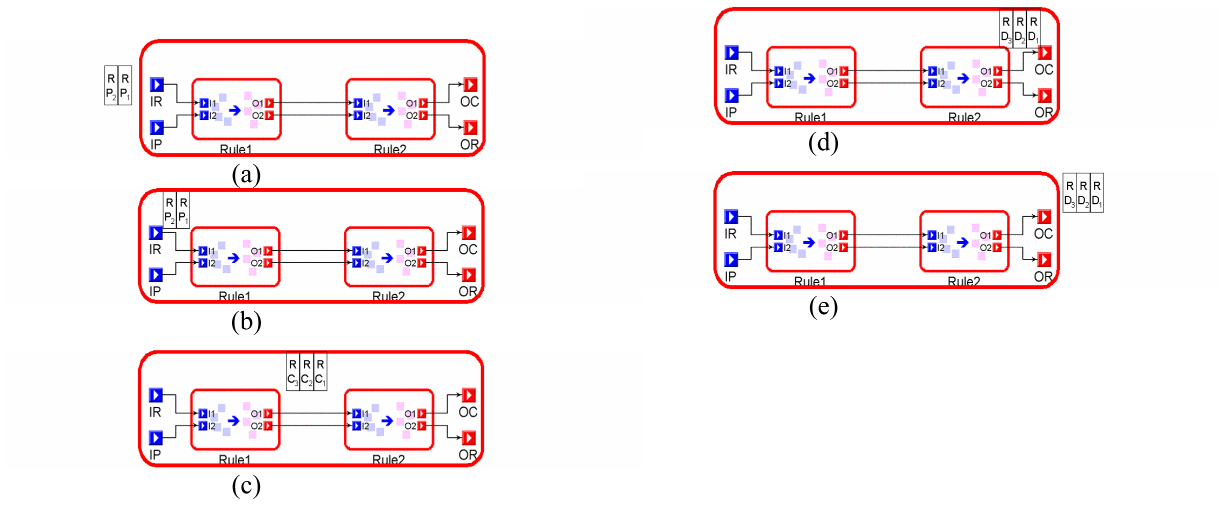


Figure 19 Rule execution of a Block

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.

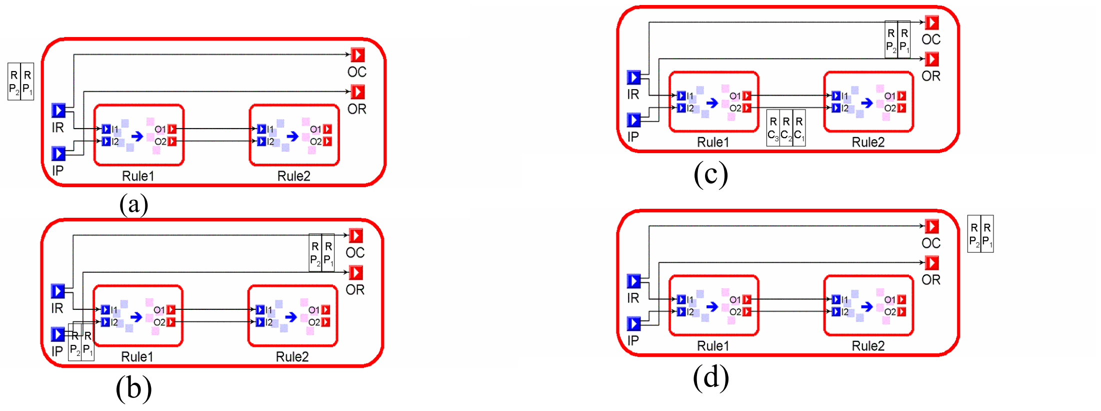
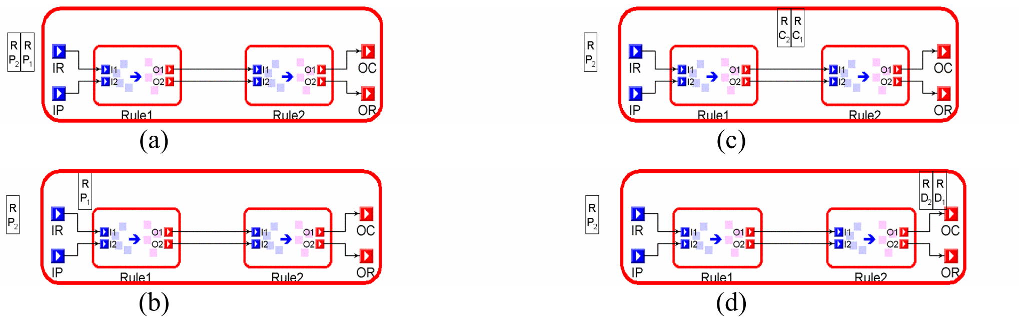


Figure 20 Sequence of execution within a *Block*

The *ForBlock* has a different execution semantic: if there are *n* (> 0) 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 pushed through, etc. The semantics is illustrated with the help of an example on Figure 21.



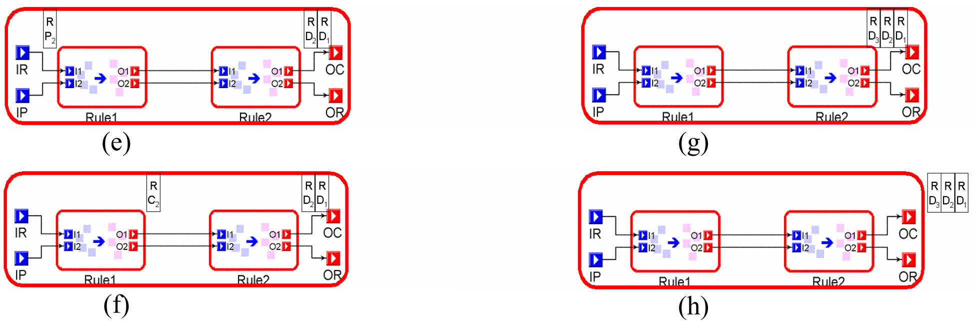


Figure 21 Rule execution sequence of a *ForBlock*

Similarly 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 the block.

```
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 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 semantics of a *Test/Case* is similar to any other rule. When fired, it consumes all its input packets to produce some output packets. However, for Test/Cases one can have multiple output interfaces. In Figure 23 a test is shown that has two cases. The *Test* has input interface ({IR,IP}) and two output interfaces ({OR1, OP1} and {OR2, OP2}). When the test is executed each incoming packet will be tested by an embedded *Case,* and placed on the corresponding output interface.

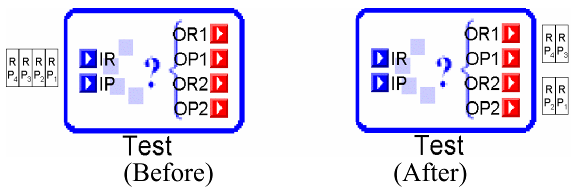
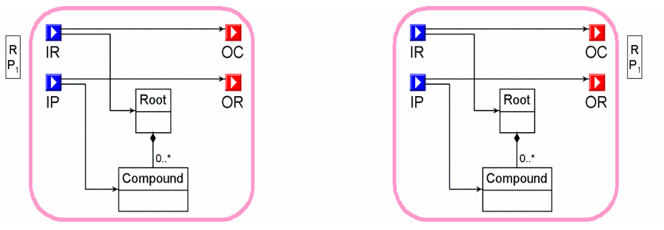


Figure 23: Execution of a *Test/Case* construct

The test must contain at least one *Case* which is a rule with no actions (i.e. no side effects). A *Case* contains a pattern (containing *bind* objects only), a guard condition, and an input and an 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.



(Before) (After)

Figure 24 Execution of a single (successful) *Case*

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

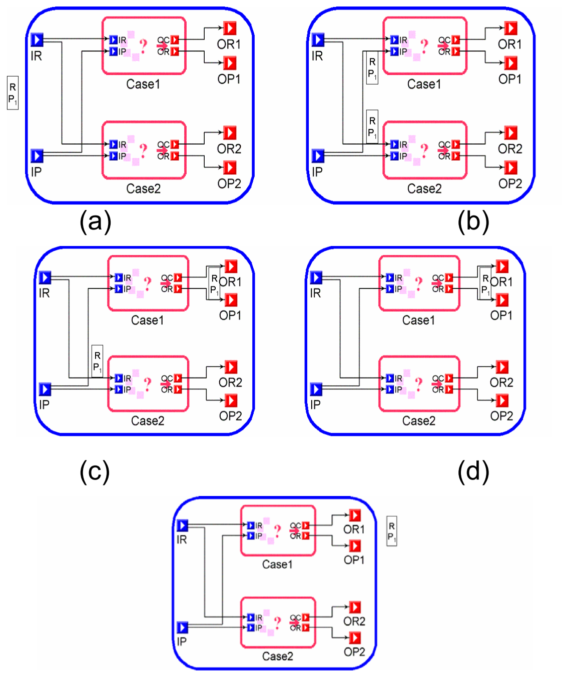


Figure 25 Inside the execution of a *Test*

When a test has many cases, then each input packet is propagated to each case to find which cases are satisfied for the particular packet and the resulting packets are 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 is that

an input packet will be tested with all the cases, more than one case may succeed and this may lead to non-determinism.
A variation on the default behavior is achieved by using the Boolean "cut" attribute of a *Case*. When a *Case* has its "cut" behavior enabled and the case succeeds on a given input, then 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 and 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 returns is not empty)
                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, the execution becomes non-deterministic. The execution engine chooses a path non-deterministically, and the chosen path is executed completely before the next path is chosen.

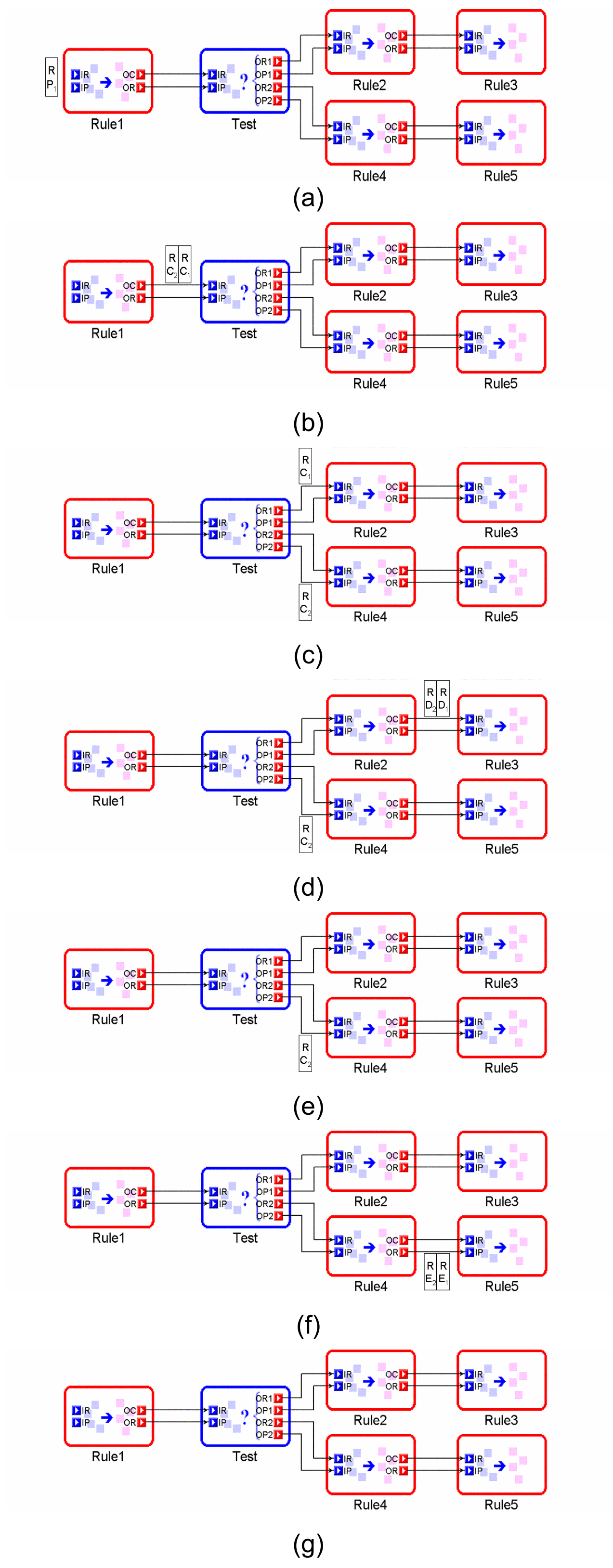


Figure 27 A non-deterministic execution sequence

Figure 27 shows a non-deterministic execution sequence. Here the non-deterministic execution is due to a test/case (but it could also have been caused by a rule connected to more than one other rule). After the branch, there are packets on both 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 at all, 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 meta-program —resulting in a visual modeling environment that allows creating and editing transformation programs. The three sublanguages were defined as three separate, but related class diagrams, thus yielding a modular design for the language. The metamodel composition capabilities of GME [1] allowed this. Such a modular design also enables changing and evolving the sublanguages independently.
The language was *implemented* using an interpreter first, but later a code generator was developed 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).
The underlying technology used for the implementation of GReAT is the Universal Data Model (UDM) package [3]. 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. 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.

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 28. 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.

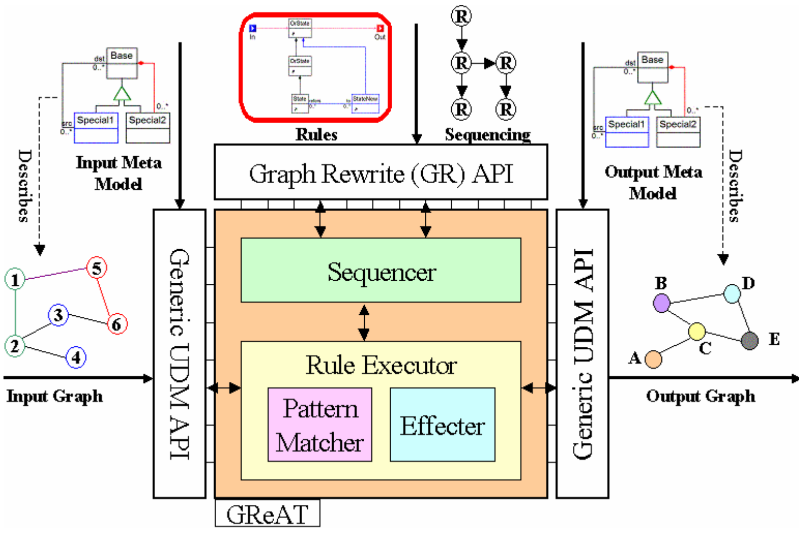


Figure 28 The GReAT interpreter

Once the transformation has stopped, the resulting graph will be compliant with the output metamodel. This is enforced by the UDM package, as it does not allow creating type-incompatible constructs. However, the checking of the OCL constraints over the result is not automatic: the programmer should include provisions for invoking this step after the transformation stops. Note that the result may not be unique: two runs on the same input may produce different results. The main cause of this is the pattern matcher: the ordering of matches delivered by the matcher is not deterministic. We are working on extending the language to address this issue.

5. Examples and Results

To test GReAT and to measure its usability we chose some challenge problems that 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 introduced interesting challenges. To map concurrent state machines to a flat state machine there is a need for complex operations that include computing a Cartesian product over the state spaces of the concurrent machines. This particular transformation required a depth-first, bottom up approach, and also proved that the system can allow different traversal schemes.
- The next example was to generate an equivalent Hybrid Automata [24] model from a given Matlab Simulink-Stateflow model. This was another non-trivial

example as the mapping is not a straightforward one-to-one mapping. 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 behaviour. 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 was to build a code-generator for a domain-specific modeling language. The specific transformation program converted Stateflow models into C++ code. For this, we have created a metamodel with classes representing specific C++ code fragments (e.g. "GuardExpression") that were instantiated with appropriate strings for attributes generated during the transformation. This example was chosen because it is a relevant, practical problem.

All these challenge problems have been solved along with other simple example problems using the GReAT language and interpreter. The state machine flattening example was solved using a recursive depth-first bottom up algorithm that first calls flattening on its children before flattening itself. A simpler, reachability analysis problem uses the mark and sweep algorithm [38].

The goal of using a graph transformation based specification language is to increase programmer's productivity. Table 1 shows some results by comparing the size of and time taken to develop GReAT specifications for model transformation problems to the 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-hrs)	Est. LOC
Mark and sweep algorithm on Finite State Machine	7/2	~2	100

Hierarchical Data Flow to Simple Data Flow	11/3	~3	200
Hierarchical Concurrent State Machine to Finite State Machine	21/5	~8	500
Simulink Stateflow to C code	70/50	~25	2500
Matlab Simulink/ Stateflow to Hybrid Automata	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.

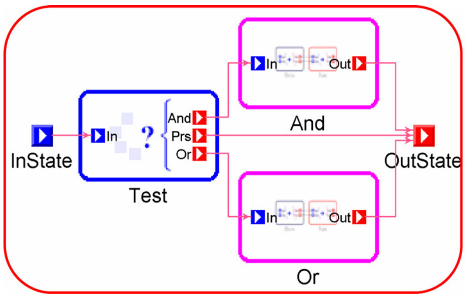


Figure 29 The top-level rule

The flattening algorithm is depth-first/bottom-up. This is achieved using a recursive block *Top-level* (shown in Figure 29). 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.

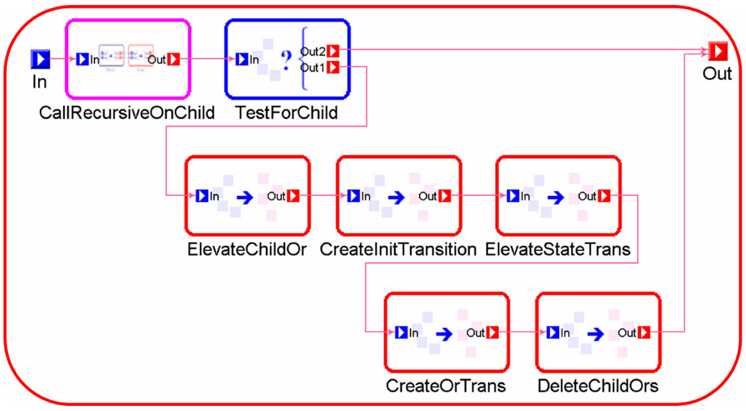


Figure 30 Inside the OR rule

Figure 30 shown the rules inside the *Or* block of Figure 29. 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 29) 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 31).

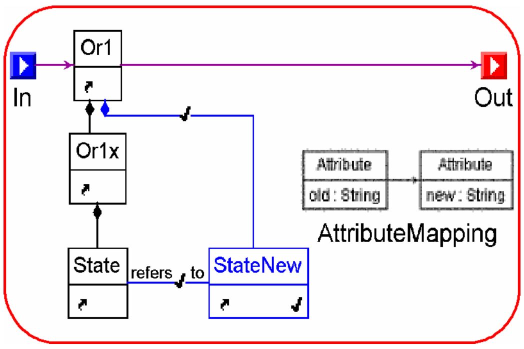


Figure 31 *ElevateChildOr* rule

Figure 31 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 more rules. For the sake of brevity we have not described the flattening of the *and-states*. However, the full transformation is available as an example in the software distribution, available from our website.

6. Comparison with related work

GReAT has been developed as a new tool for building model transformation applications. As such, it relies on the intellectual heritage of other transformation tools. Here we give a short overview of these tools and their relationships to GReAT.

PROGRES [49] is arguably the first widely used tool for specifying transformations on structures represented as graphs. PROGRES has sophisticated control structures for controlling the rewriting the process, in GReAT we have used a similar, yet different approach: explicitly sequenced rules that form control flow diagrams. PROGRES also supports representing type systems for graphs, in GReAT we use UML diagrams for this purpose. The very high-level notion of graph transformations used in PROGRES necessitates sophisticated techniques for efficient graph matching; in GReAT we mitigate this problem by using less powerful rules and (effectively) perform a local search in the host graph.

Fujaba [27] is similar to GREAT in the sense that it relies on UML (the tool was primarily built for transforming UML models) and uses a technique for explicitly sequencing transformation operations. However, Fujaba follows the state-machine-like “story diagram” approach [20] for scheduling the rewriting operations; a difference from GReAT.

AGG [52] is a graph transformation tool that relies on the use of type graphs, similar to (but different from) UML diagrams. Recent work related to AGG introduced a method for handling inheritance, as well as a sophisticated technique for checking for confluence (critical pair analysis). In GReAT, inheritance is handled in the pattern matching process, and the confluence problem is avoided by using explicit rule sequencing. AGG has support for controlling the transformations in the form of layered grammars; a problem solved differently in GReAT.

VIATRA [9] is yet another graph transformation tool that interesting capabilities for controlling the transformations (state machines), and the composition of more complex transformations. In GReAT similar problems were addressed via the explicit control flow across rules and the formulation of blocks. Higher-order transformations were also introduced in VIATRA, there is no similar capability in GReAT currently.
In summary, PROGRES, AGG, Fujaba, and VIATRA together with GReAT belong to the same family of model transformation tools. Where GReAT is different from these tools can be summarized as follows: (a) GReAT supports an unlimited number of input and output graphs that the transformation can operate on, (b) it follows the UML (class diagram) style for specifying the type system for the graphs (with the implications for the pattern matcher), (c) it uses explicitly sequenced rewriting operations that always operate within a context provided by the initial bindings for pattern variables, (d) one can introduce new, "temporary" type systems for the transformations that allow, for instance, cross links across graph elements from different UML class diagrams, (e) it uses control flow diagrams and structuring constructs for the detailed programming of the transformations, (f) it uses limited-context pattern matching for efficiency, and (g) it supports checking the well-formedness of the results via the constraints specified in the UML class diagram of the target domain.
GReAT solves some theoretical issues through its implementation. For instance, confluence problems are avoided through the use of explicit sequencing, the resulting graphs are always compliant with their metamodels, hence they are consistent with respect to the UML class diagram, and the OCL constraints can be checked against the result of the transformation after. However, there are a number of other issues that necessitate further investigation, such as compliance with OCL constraints and termination of the transformation.
GReAT can also be compared to the recent QVT submission [43] to the OMG MDA standardization process. However, we should emphasize that GReAT was meant to be a research tool and not and industry standard. Before comparing GReAT to the QVT proposal, it should be compared to the Request for Proposal. We believe GReAT satisfies most of the requirements of the QVT RFP, with the exception of the bi-directional transformations: GReAT transformations are inherently uni-directional. With respect to the QVT submission, the biggest

difference is in the highly declarative nature of the QVT: it focuses on relation mappings. This is a very high-level approach, and it is far from the pragmatic, efficiency-oriented approach followed in GReAT. We conjecture that describing a transformation in QVT is probably more compact, but the same transformation in GReAT is more efficient. We have done a number of non-trivial transformations in GReAT, and it would be an interesting experiment to compare these to their equivalents in QVT.

In a more general setting, we need compare GReAT and the environment it is hosted in: the MIC tools that include GME to other similar tools. Honeywell's DOME [18], MetaCASE's MetaEdit [27], and the ATOM3 [30] environment are the most relevant examples that support domain-driven development. Our main difference is the use of UML and OCL for metamodeling and they way these metamodels are simultaneously used for instantiating a visual modeling environment. Also, our transformations follow a high-level method for describing the transformation steps expressed in the context of the metamodels. With exception of ATOM3, all the above tools use a scripting language, in contrast.

Recent work on semantic anchoring [10] describes how the metamodeling in the UML/OCL style can be coupled to providing a semantic foundation for domain-specific modeling languages via the model transformation approach described in this paper.

7. Conclusion and Future Work

This paper has shown a technique and a language 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 and edges linking nodes from different domains are needed, (2) inheritance must be handled in the pattern matching process, and (3) explicit sequencing of transformation rules is desirable.

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, including 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 sublanguages: (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 roles 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 the informal 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. The language needs to be based on a formal theory, such that theoretical results could be used to verify interesting properties of the transformations. More experiments need to be performed that measure efficacy of the GReAT implementation with respect to hand code. The primary goal of this research was 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.

8. 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.

9. References

[1] 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.

[2] 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.

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

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

[5] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation, in M. Wermelinger and T. Margaria-Steffens (eds.), Proc. Fundamental Aspects of Software Engineering 2004, volume 2984. Springer LNCS, 2004.

[6] 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.

[7] D. Blostein and A. Schuerr. Computing with Graphs and Graph Rewriting. Software Practice and Experience, 29(3):1--21, 1999.

[8] 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.

[9] Horst Bunke: Programmed Graph Grammars. Graph-Grammars and Their Application to Computer Science and Biology 1978: 155-166.

[10] Kai Chen, J. Sztipanovits, S. Neema, M. Emerson and S.Abdelwahed, “Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages,” 5th ACM International Conference on Embedded Software (EMSOFT'05), 2005, in press.

[11] Björn Cordes, Karsten Hölscher, Hans-Jörg Kreowski: UML Interaction Diagrams: Correct Translation of Sequence Diagrams into Collaboration Diagrams. AGTIVE 2003: 275-291.

[12] Gyorgy Csertan, Gabor Huszerl, Istvan Majzik, Zsigmond Pap, Andras Pataricza, Daniel Varro. "VIATRA — Visual Automated Transformations for Formal Verification and Validation of UML Models," IEEE Conference on Automated Software Engineering, p. 267, 2002.

[13] DOME Guide, Honeywell, Inc. Morris Township, N.J, 1999.

[14] Dorr, H. 1995. Efficient Graph Rewriting and its implementation. Lecture Notes in Computer Science, 922. Springer.

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

[16] H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. in F. Parisi-Presicce, P. Bottoni, and G. Engels (eds.), Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), LNCS 3256, pages 161-177, Rome, Italy, October 2004. Springer.

[17] 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).

[18] Engstrom, E. Krueger, J.: “Building and rapidly evolving domain-specific tools with DOME”, IEEE International Symposium on Computer-Aided Control System Design, pp. 83-88, 2000.

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

[20] T. Fischer, Jörg Niere, L. Torunski, and Albert Zündorf, 'Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language', in Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn,

Germany (G. Engels and G. Rozenberg, eds.), LNCS 1764, pp. 296--309, Springer Verlag, November 1998.

[21] 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.

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

[23] J. Greenfield, K. Short, S. Cook, S. Kent: “Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools”, Wiley, 2004.

[24] Reiko Heckel, Annika Wagner: Ensuring consistency of conditional graph rewriting - a constructive approach. Electr. Notes Theor. Comput. Sci. 2: (1995).

[25] T. A. Henzinger. The Theory of Hybrid Automata. In Proc. of IEEE Symposium on Logic in Computer Science (LICS'96), pages 278--292. IEEE Press, 1996.

[26] 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.

[27] Kelly, S., Tolvanen, J-P, "Visual domain-specific modelling: Benefits and experiences of using metaCASE tools", Proceedings of International workshop on Model Engineering, ECOOP 2000, (ed. J. Bezivin, J. Ernst), 2000.

[28] Thomas Kleín, Ulrich A. Nickel, Jörg Niere, and Albert Zündorf, 'From UML to Java And Back Again', Tech. Rep. tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[29] 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.

[30] Lara, J. d. and Vangheluwe, H. 2002. AToM3: A Tool for Multi-formalism and Meta-modelling. In Proceedings of the 5th international Conference on Fundamental Approaches To Software Engineering (April 08 - 12, 2002). R. Kutsche and H. Weber, Eds. Lecture Notes In Computer Science, vol. 2306. Springer-Verlag, London, 174-188.

[31] 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.

[32] J. de Lara and G. Taentzer. “Automated Model Transformation and its Validation using AToM3 and AGG,” in Alan Blackwell, Kim Marriott, and Atsushi Shimojima (eds.), Diagrammatic Representation and Inference. Springer, LNAI 2980, 2004.

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

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

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

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

[37] 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.

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

[39] Meta-Object Facility, available from www.omg.org/mof.

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

[41] Ulrich Nickel and Jorg Niere and Albert Zundorf: The FUJABA environment, Proc. of International Conference on Software Engineering, pp.742-745, 2000.

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

[43] OMG QVT submission, available from www.omg.org/ad/2003-08-08 .

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

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

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

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

[48] G.Schmidt, R. Berghammer (eds.), "Proc. Int. Workshop on Graph-Theoretic Concepts in Computer Science", (WG '91), LNCS 570, Springer Verlag (1991).

[49] A. Schürr; Specification of Graph Translators with Triple Graph Grammars; Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 903, Berlin: Springer Verlag; June 1994; 151-163.

[50] A. Schurr, A. Winter, and A. Zundorf. Graph grammar engineering with PROGRES. In Proc. ESEC `95, LNCS 989, pages 219-234, Barcelona, Spain, Sept. 1995.

[51] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", Computer, Apr. 1997, pp. 110-112

[52] Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. Ph.D.Thesis, TU Berlin, Shaker Verlag, Germany, 1996

[53] G. Taentzer.: AGG: A Graph Transformation Environment for Modeling and Validation of Software.In J. Pfaltz, M. Nagl, and B. Boehlen (eds.), Application of Graph Transformations with Industrial Relevance (AGTIVE'03), volume 3062. Springer LNCS, 2004.

[54] Zeigler, B.P., DEVS Representation of Dynamical Systems : Event-Based Intelligent Control, Proceedings of the IEEE, V77, N1, Jan. 1989, pp 72-80.

[55] A. Zundorf. Graph pattern matching in PROGRES. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci., volume 1073 of Lecture Notes in Computer Science, pages 454--468. Springer, 1996.

Appendices

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)
        { bind pattern and host graph edge, add binding to p_match
          delete the pattern edge from the pattern
           return an empty match list
        }
      }
      edge = get pattern edge with one vertex bound in host graph
      if(edge exists)
      { vertices = collect vertices of the host graph adjacent to the
                   bound vertex
        make a copy of pattern 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,new_match)
          add ret_match to matches
        }
        return matches
      }
      if(all pattern edges are bound)
      { add p_match to matches
        return matches
      }
      else
        return empty list
}
```

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)
       { add edge binding to p_match
         delete edge from new_pattern
         return empty list
       }
     }

 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 vertex 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 pattern matches are bound)
 {     add p_match to matches
       return matches
 }else
       return empty list
}
```