Journal of Universal Computer Science, vol. 9, no. 11 (2003), 1296-1321
submitted: 21/6/03, accepted: 5/9/03, appeared: 28/11/03

On the Use of Graph Transformation in the Formal Specification of Model Interpreters

Gabor Karsai
(Institute for Software Integrated Systems (ISIS)
Vanderbilt University, Nashville, TN, USA
gabor.karsai@vanderbilt.edu)
Aditya Agrawal
(Institute for Software Integrated Systems (ISIS)
Vanderbilt University, Nashville, TN, USA
aditya.agrawal@vanderbilt.edu)
Feng Shi
(Institute for Software Integrated Systems (ISIS)
Vanderbilt University, Nashville, TN, USA
feng.shi@vanderbilt.edu)
Jonathan Sprinkle
(Institute for Software Integrated Systems (ISIS)
Vanderbilt University, Nashville, TN, USA
jonathan.sprinkle@vanderbilt.edu)

Abstract: Model-based development necessitates the transformation of models between different stages and tools of the design process. These transformations must be precisely, preferably formally, specified, such that end-to-end semantic interoperability is maintained. The paper introduces a graph-transformation-based technique for specifying these model transformations, gives a formal definition for the semantics of the transformation language, describes an implementation of the language, and illustrates its use through an example.
Key Words: Graph grammars, graph transformations, Model-Integrated Computing, domain-specific modeling languages, model-driven architecture, formal specifications.
Category: D.2.2 Tools and Techniques

1 Introduction

The engineering of complex, Computer-Based Systems (CBS) both enables and necessitates formal approaches that support a model-based engineering process. As discussed in a previous paper [1], there are several motivating factors for doing this. (1) We wish to use a model-based approach for development. Using models implies the use of precisely defined, domain-specific modeling languages, which capture the formal specification of the system being designed. (2) The model-based approach includes a significant effort to perform design-time analyses on the models. Early detection of problems with the design allows saving work at integration time, and can significantly decrease the effort there. (3) In order to synthesize/generate an implementation from the design, one has to bridge the gap between the Domain-Specific Modeling Language (DSML) used in the design process and the semantic domain defined by the underlying software/hardware infrastructure or platform.

In practice, there are very few complete model-based tool-suites yet, although many packages [2][3][4] support portions of the model-based design process outlined above. We conjecture that this can be attributed to that fact that, although tools for specific points in the design process are available, the capability of moving designs from one tool to another is lacking. To illustrate the point, let us consider a design flow based on UML. In this case, UML tools are used to create models of the application, and, provided they are available, code generator tools will generate the application from the models. In practice, this latter step is often replaced by (or at least augmented with) hand-produced code, as current code generators do no typically "know" about the particulars of the target execution platform. Furthermore, if one wants to verify state machine models for the system, one has to rebuild the models in the input language of some analysis tool, like SMV [5] or KRONOS [6]. The lack of these capabilities in practical model-based engineering of CBS has motivated us to look for solutions that allow the *transformation* of design information. Obviously, these transformations can always be implemented by writing translators by hand, but this approach, in addition to being inefficient, has yet another serious drawback: the semantic mapping between the input and the output is vaguely specified. In order to create the design translators in a correct-by-construction manner, we have to find approaches that allow the formalization of the transformation itself. Naturally, the formal specification must have an executable semantics, too, as we would like to facilitate the transformation based on its formal model.

In this paper we present a formal approach for specifying translators that allow one to capture the semantic mapping between the designs captured in various design tools. We claim that in addition to supporting the CBS design process through the "semantic bridges" between tools, it can actually also be used to formally define the semantics of DSML-s. The reasoning is as follows: assume that a "base" semantics is defined for an underlying platform. In other words, the platform defines a semantic domain, into which the design models have to be mapped. For instance, we have the formal specification of a platform that supports Finite State Machines. We conjecture, that given a DSML and its formal mapping onto the platform's semantic domain, one can formally define the semantics of the DSML. This idea has been presented in [7], in the context of Statecharts and can easily be generalized.

The paper introduces a manifestation of the model-based engineering process for CBS: Model-Integrated Computing (MIC), and reviews graph transformation techniques. Next it introduces a graph transformation language we have developed, formally specifies its semantics, and it shows how it can be used for representing specific transformations. The paper concludes with a summary and suggestions for further research.

2 Background

Model-Integrated Computing (MIC, for short) is a software and system development approach that advocates the use of domain specific models to represent relevant aspects of a system. The models capturing the design are then used to synthesize executable systems, perform analysis or drive simulations. The advantage of this methodology is that it speeds up the design process, facilitates evolution, helps in system maintenance and reduces the cost of the development cycle [8].

The MIC development cycle (see Figure 1) starts with the formal specification of a new application domain. The specification proceeds by identifying the concepts, their attributes, and relationships among them through a process called metamodeling. Metamodeling is enacted through the creation of metamodels that define the abstract syntax, static semantics and visualization rules of the domain. The visualization rules determine how domain models are to be visualized and manipulated in a visual modeling environment. Once the domain has been specified, the specification is used to generate a Domain Specific Design Environment (DSDE, for short). The DSDE can then be used to create domain specific designs/models; for example, a particular finite state machine is a domain specific design that conforms to the rules specified in the metamodel of the finite state machine domain. However, to do something useful with these models such as synthesize executable code, perform analysis or drive simulators, we have to convert the models into other formats like executable code, inputs to some analysis tool, or configuration files for simulators. This mapping of models to a more useful form is called model interpretation and is performed by model interpreters. Model interpreters are programs that convert models of a given domain into some other format, typically with a different semantic domain. The output of the transformation can be considered as another model that conforms to a different metamodel and thus model interpreters can be considered as tools facilitating a mapping between domains[8].

The premier MIC implementation is built around a metaprogrammable toolkit called Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems (ISIS), Vanderbilt University. It provides an environment for creating domain-specific modeling environments [9]. The metamodeling language of GME is based on UML class diagrams [10]. This language is used to define domain specific modeling languages by capturing the (abstract) syntax,

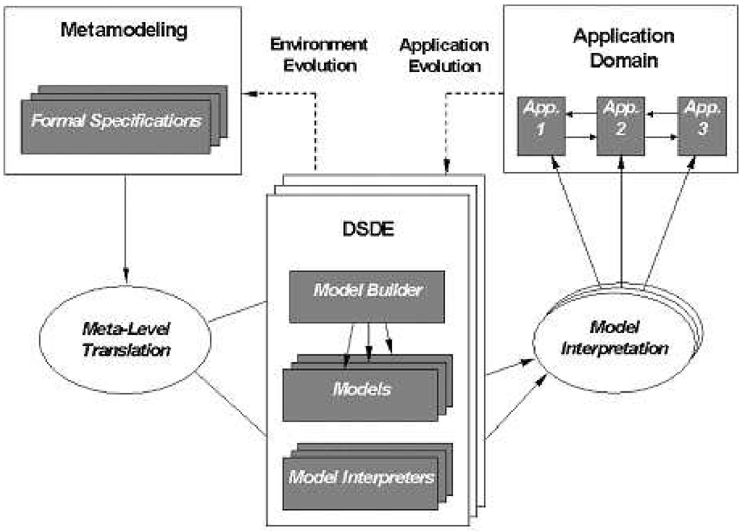

Figure 1: The MIC Development Cycle

semantics and visualization rules for the DSML in metamodels. A tool called the meta-interpreter interprets the metamodels and configures a new instance of GME via a file. 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. GME is used both in the metamodeling environment and the target environment.

GME has both a metamodeling environment and metamodel interpreter that generates a new modeling environment from the metamodels. Currently the mappings from models to a semantic domain are performed by model interpreters. These interpreters are written by hand. This is the most time consuming and error prone phase of the MIC approach. There is a need for higher-level methods and tools for building model interpreters. These generic tools should automatically generate domain specific model interpreters from models.

The MIC approach described above is gaining a lot of attention recently with the advent of the Model Driven Architecture (MDA) by Object Management Group (OMG) [11]. The MDA could be a particular application of the MIC approach where the domain language will be UML 2.0. However, a more general approach to the MDA problem will be to achieve domain specific model driven software development [41].

Graph grammars and graph rewriting [14][15] have been developed during the last 25+ years as formal techniques for modeling and very high-level programming. Graph grammars are the natural extension of the generative grammars of Chomsky from the domain of strings into the domain of graphs. The production rules for (string-) grammars could be generalized into production rules on graphs, which generatively enumerate all the sentences (i.e. the "graphs") of a graph language. String rewriting is a well-known technique, where replacement rules containing patterns and replacement strings convert matching strings into other strings. String rewriting can be generalized into graph rewriting as follows: a graph-rewriting rule consists of a pattern graph and a replacement graph. The application of a graph-rewriting rule is straightforward; the matching sub-graph of a (host) graph is replaced with another graph. For precise details see[14].

Beyond the ground-laying work in the theory of graph grammars and rewriting, the approach has found several applications as well. Graph rewriting has been used in formalizing the semantics of StateCharts[18], as well as various concurrency models[14]. Several tools -including programming environments- have been developed[16][17] that illustrate the practical applicability of the graph rewriting approach. These environments have demonstrated that (1) complex transformations can be expressed in the form of rewriting rules, and (2) rewriting rules can be compiled into efficient code. Programming via graph transformations has been applied in some domains[15] with reasonable success. In this paper, we argue that the graph transformation techniques offer not only a solid, well-defined foundation for model transformations, but can also be a be a practical solution.

The need for techniques for model transformations has been recently recognized in the UML world. For examples, see[21], [22],[23],[26], and[27]. Model transformation is an essential tool for many applications, including translating abstract design models into concrete implementation models[26], for specification techniques[23], translation of UML into semantic domains[27], and even for the application of design patterns[29]. The new developments in UML (see[24],[25]) emphasize the use of meta-models, and provide a solid foundation for the precise specification of semantics. Related efforts, like aspect-oriented programming[19] or intentional programming[20] could also benefit from using transformation techniques based on graph rewriting. A natural extension of these concepts is to use transformational techniques for translating models into semantic domains: a task for which graph transformation techniques are -arguably- well-suited.

3 Graph Grammars and Transformations

Domain specific modeling languages are specified with the help of UML class diagrams that capture the abstract syntax of the language. Therefore, domain specific models are networks of objects, where each object (link) belongs to a corresponding class (association) in the class diagram. From a mathematical

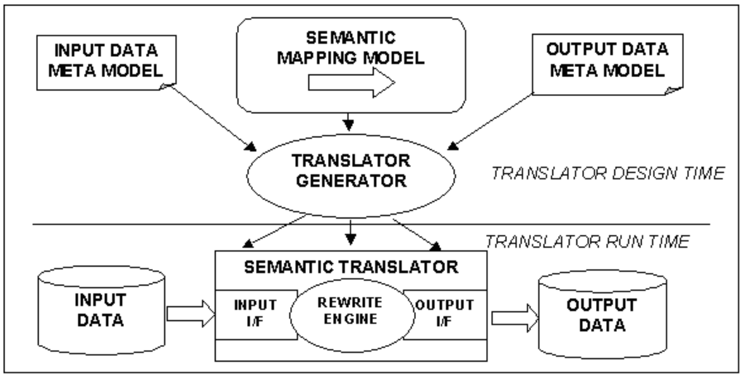

Figure 2: Metamodel based model transformation

viewpoint, one can recognize that domain specific models are graphs, to be more precise: typed multi-graphs, where the labels denote the corresponding entities (i.e. types) in the metamodel. Thus, the model transformation problem can be converted into a graph transformation problem. We can then use the mathematical concepts of graph transformations to formally specify the intended behavior of model transformers.

Graph grammars and graph transformations (GGT) have been recognized as a powerful technique for specifying complex transformations that can be used in various situations in a software development process[31][32][33][34]. Many tasks in software development can be formulated using this approach, including weaving of aspect-oriented programs[35], application of design patterns[33], and the transformation of platform-independent models into platform specific models[13].

There exists a variety of graph transformation techniques described in [14][15] [16][17][18][36][37][38][39]. The prominent among these are node replacement grammars, hyperedge replacement grammars, algebraic approaches and programmed graph replacement systems. These techniques have been developed mostly for the specification and recognition of graph languages, and performing transformations within the same "domain" (i.e. graph), while we need a graph transformer that works on two different kinds of graphs. Moreover, these transformation techniques rarely use UML class diagrams for the specification of their graph schema.

In general, we pursue a scheme illustrated in Figure 2. We wish to create the metamodel of the input and target models, as well as a model of the mapping: the transformation between the two. Ultimately, from these models we wish to create the executable translator.

In summary, the following features are preferred in the transformation language:

- The language should use UML for the specification of the abstract syntax, and integrity constraints.
- There should be support for transformations that create an entirely different graph based upon a given graph. The two graphs may belong to different metamodels and have different integrity constraints.
- The approach should be expressive enough to specify model transformers that convert models of high-level graphical languages to low-level implementations, with no or minimal textual coding.
- The language should have efficient implementations of its programming constructs. The implementation should have comparable efficiency to equivalent hand-written code.
- The language should be "user friendly" and increase programmer productivity.

4 Graph Rewriting and Transformation Langauge (GReAT)

In this paper we will focus on a generalized graph transformation system called Graph Rewriting and Transformation language (GReAT). GReAT is a visual language developed using GME[9].

GReAT is divided into 2 major parts.

1. Graph Transformation Language
2. High-Level Control Flow Language

4.1 Graph Transformation Language

In model-interpreters that perform model transformations there exists a concern about structural integrity during the transformation process. Model-interpreters transform models from one domain to models that conform to another domain. This class of transformations makes the problem two-fold. The first problem is to specify and maintain two different models conforming to two different metamodels (as in MIC, metamodels are used to specify structural integrity constraints). The second problem is maintaining linkages between the two (source and target) models. These linkages, in general, can be vertices (i.e. objects) and links (i.e. instances of associations), which are created temporarily during the transformation process, however they are not legal in any of the metamodels. The linkages are required to correlate graph objects across the two domains.

A solution to these problems is to use both the source and target meta-models to explicitly specify the temporary vertices and edges. This creates a unified meta-model that incorporates temporary objects. The advantage of this approach is that we can then treat the source model, target model and temporary objects as a single graph. Standard graph-based techniques can then be used to specify the transformation.

In our approach, a transformation is built from elementary rewriting rules. Each rule contains a pattern graph and each pattern object's type conforms to a class or association in the unified metamodel. Only rules that do not violate the unified metamodel are allowed. At execution time, after finishing he transformation, the temporary objects are removed and the resulting model conforms exactly to its own meta-model. The transformation language is inspired by many previous efforts, such as [7][8][9][17][18].

The transformation algorithm is specified in the form of a partially ordered set of transformation rules. A simple transformation rule (an shown in Figure 3) contains a pattern graph that consists of pattern vertices and edges, called the pattern objects. These pattern objects conform to specific types introduced in the metamodel. Each pattern object has a qualifier attribute called *Action* that specifies the role it plays in the transformation. A pattern object can play three different roles as follows:

1. *Bind*: Match object(s) in the graph.
2. *Delete*: Match object(s) in the graph, then remove the matched object(s) from the graph.
3. *New*: Create new object(s) (provided the pattern matched successfully).

The execution of a rule involves matching every pattern object with the roles *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

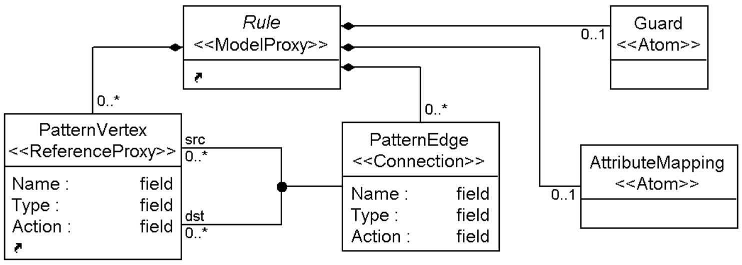

Figure 3: Simplified metamodel for Rule

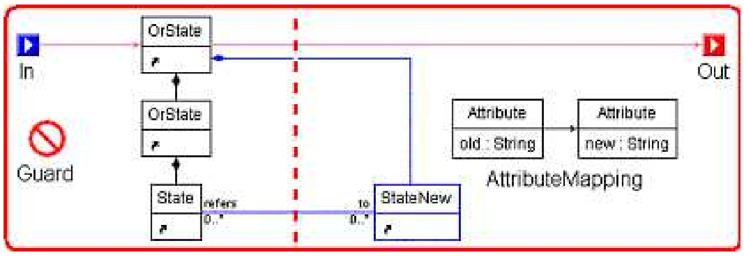

Figure 4: An example rule in the GReAT visual langauge

then the objects marked *new* are created. Sometimes the patterns by themselves are not sufficient to specify the exact graph parts to match and we need other, non-structural constraints in the pattern. An example for such a constraint is: "the value of an integer attribute of a particular vertex should be within a specific range." These constraints are expressed using the Object Constraint Language (OCL)[19]. There is also a need to provide values for attributes of newly created objects and/or modify attributes of existing objects. These needs are addressed by the "attribute mapping" language, which is based on a restricted set of the C language.

An example rule is shown in Figure 4. All objects on the left hand side of the dashed line have the role "bind", while the objects on the right hand side have the action of create. The rule specifies that starting from an *OrState* the matcher must find a child *OrState* that also has a child *State*. For each such match a new state *StateNew* must be created as a child of the original *OrState*.

4.2 Controlled Graph Rewriting and Transformation

In order to increase the effectiveness of the transformation language it is essential to have efficient implementations for the rule execution. The pattern matcher being the most time consuming operation needs to be made as efficient as possible. In order to make the search algorithm less time consuming, the matcher doesnt search the pattern in the entire graph but only within a context. The context is specified by an initial set of bindings for some of the pattern vertices and edges in a rule, reducing the time complexity of the search. The initial set of bindings is established by using *PORT* objects in the rewriting rules that form the interface of the rewriting rule.

The next concern is the application order of rewriting rules. Classical graph grammars can apply any feasible production, i.e. they are based on nondeterministic choice. This technique is good for generating and matching languages, but model-to-model transformations often need to follow an algorithm that re-

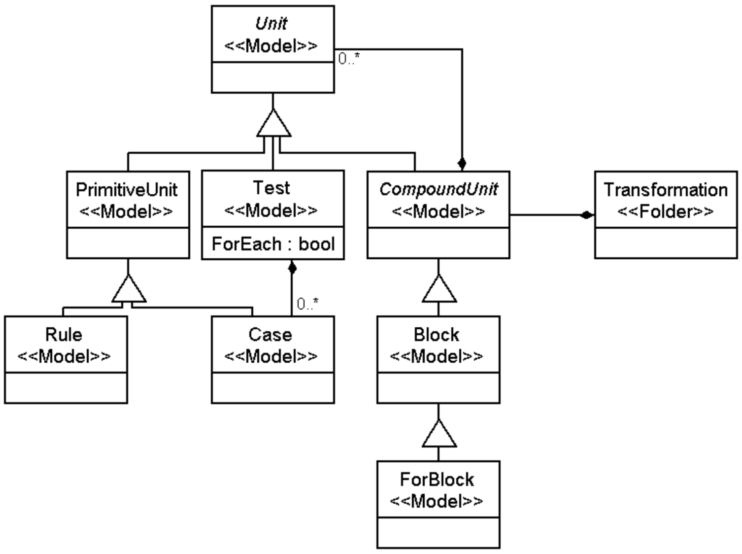

Figure 5: The GReAT Object Hierarchy

quires strict control over the execution of rules. Furthermore, by specifying a rule execution sequence the implementation can be made more efficient. There is a need for a high-level control flow language that can control the application of the productions and allow the user to manage the complexity of the transformation. This prompted us to add a high-level control flow language to GReAT. The control flow language supports the following features:

– *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 unspecified.

– *Hierarchy*: Compound rules can contain other compound rules or primitive rules.

– *Recursion*: A rule can call itself.

– *Test/Case*: A conditional branching construct that can be used to choose between different control flow paths.

Figure 5 shows the conceptual hierarchy of GReAT. The *CompoundUnit* is used for Hierarchy and recursion. *Test* and *Case* is for conditional branching. Sequencing and non-deterministic execution are achieved with the help of sequencing connections (not shown in figure). These connections (implicitly) specify the order of execution for the rules.

4.3 Formal Semantics of GReAT

A formal specification of the GReAT execution semantics is described in this section. We use the Object-Z notation [40] for the specification. The specification starts with the definition of a graph.

VERTEX

$$\begin{array}{l} name : NAME \\ type : TYPE \end{array}$$

EDGE

$$\begin{array}{l} name : NAME \\ type : TYPE \\ src : VERTEX \\ dst : VERTEX \end{array}$$

GRAPH

$$\begin{array}{l} vertices : \mathbb{P}\ VERTEX \\ edges : \mathbb{P}\ EDGE \\ \hline \forall\, edge : EDGE \in edges \\ \quad \bullet\ edge.src \in vertices \land edge.dst \in vertices \end{array}$$

Vertices and edges both have a type associated with them. These types must conform with the respective metamodels of the graphs. Both host graphs and pattern graphs are defined by the same data structure. The additional attributes of the pattern graph, like actions are captured separately using maps.

The *MATCH* class is a data structure that associates pattern graph elements with host graph elements. (The host graph is the graph in which we search for a match.) It contains a partial function from host vertices to pattern vertices and another partial function that maps host edges to pattern edges.

MATCH

$$
\begin{array}{l}
hostGraph : GRAPH \\
patternGraph : GRAPH \\
vertexBinding : [VERTEX \nrightarrow VERTEX] \\
edgeBinding : [EDGE \nrightarrow EDGE] \\
\hline
\forall (hv, pv) \in vertexBinding \bullet \\
\quad hv \in hostGraph.vertices \wedge \\
\qquad\quad pv \in patternGraph.vertices \\
\forall (he, pe) \in edgeBinding \bullet \\
\quad he \in hostGraph.edges \wedge \\
\qquad\quad pe \in patternGraph.edges \wedge \\
\qquad \exists (hvs, pvs), (hvd, pvd) \in vertexBindings \bullet \\
\qquad\quad he.src = hvs \wedge pe.src = pvs \\
\qquad\quad he.dst = hvd \wedge pe.dst = pvd
\end{array}
$$

Apart from the pattern graph, a rule also contains ports that allow it to interface with other rules. A port is simply used to connect with another rule. A non-empty set of ports form an interface. Each rule must contain an input and an output interface. The interface is used to pass along host graph elements. These elements are mapped to the ports of an interface to form a packet. A $PACKET$ contains a partial function that maps ports to host vertices.

PORT

$$name : NAME$$

INTERFACE

$$ports : \mathbb{P}\, PORT$$

PACKET

$$p2vMAP : (PORT \mapsto VERTEX)$$

The base class for all elements in the GReAT language that describes some operation on the graph is called $UNIT$. A $UNIT$ consists of (1) a (reference to the) host graph, (2) an input interface and (3) an output interface, (4) a set of input packets, and (4) a set of output packets. $UNIT$ is then specialized into $PRIMITIVE_UNIT$and $COMPOUND\_UNIT$. $PRIMITIVE\_UNIT$ is specialized to $RULE$ and $CASE$. These classes form the atomic building blocks of the GReAT langauge. The RULE performs an elementary transformation operation while CASE is used to check for matches (alternatives).

UNIT

$hostGraph : GRAPH$
$inputInterface : INTERFACE$
$outputInterface : INTERFACE$
$inPackets : \mathbb{P}\, PACKET$
$outPackets : \mathbb{P}\, PACKET$

$PRIMITIVE\_UNIT$ contains a pattern graph, binding of input ports to pattern elements and binding of pattern elements to output ports. It also contains many operations that are used by $RULE$ and $CASE$. The most important operation is *PatternMatcher*. This operation takes an input a partial match of the pattern on the host graph and generates the set of all possible, complete matches between the pattern and the host graph. This matcher algorithm implements the core activity performed during the execution of GReAT programs. The other operations include: *MakeInitialPartialMatch*, that takes a single input packet and converts it into a partial match using the input binding information, and *EvaluateGuard* that is used to evaluate an OCL expression on the matches returned by the matcher. All matches that fail the guard are discarded. For the sake of brevity the *EvaluateGuard* function is described in English.

PRIMITIVE_UNIT

$UNIT$
$patternGraph : GRAPH$
$inBindings : [PORT \nrightarrow VERTEX]$
$outBindings : [PORT \nrightarrow VERTEX]$
$matches : \mathbb{P}\, MATCH$
$guard : OCL\_EXPRESSION$

$\forall\, port \in dom(inBinding) \bullet port \in inputInterface.ports$
$\forall\, port \in dom(outBinding) \bullet port \in outputInterface.ports$
$\forall\, vertex \in range(outBinding) \vee range(inBinding) \bullet$
$\quad vertex \in patternGraph.vertices$

MakeInitialPartialMatch

$initialPartialMatch! : MATCH$

$inPackets' = inPackets - inPacket$
$\forall (p, hv) \in inPacket.p2vMap \bullet \exists (p, pv) \in inBinding \bullet$
$\quad (hv, pv) \in initialPartialMatch.vertexBinding$

PRIMITIVE_UNIT_contd...

PatternMatcher

$hostGraph? : GRAPH$
$patternGraph? : GRAPH$
$initialPartialMatch? : MATCH$

$\forall\, m \in matches \bullet m \supseteq initialPartialMatch$
$\quad \forall\, v \in patternGraph.vertices \bullet$
$\qquad \exists (hv, pv) \in m.vertexBinding \wedge hv = v$
$\quad \forall\, e \in patternGraph.edges \bullet$
$\qquad \exists (he, pe) \in m.edgeBinding \wedge he = e$
$\forall\, m1, m2 \in matches \bullet m1 \neq m2$

EvaluateGuard

$guard? : OCL\_EXPRESSION$

$\forall\, match \in matches \bullet$

Evaluate guard expression on match. If evaluation results false then remove match from matches.

PackageResult

$match? : MATCH$
$outPacket! : PACKET$

$\forall\, p \in outputInterface.ports \bullet$
$\quad \exists (p, pv) \in outBinding \wedge \exists (hv, pv) \in match$
$\qquad outPacket' = outPacket \oplus p \mapsto hv$
$\qquad outPackets' = outPackets \cup outPacket$

$$MakeInitialPartialMatches \mathrel{\widehat{=}} MakeInitialPartialMatch \mathbin{⨟} MakeInitialPartialMatches \;[]\; [inPackets = \{\}]$$

$$PackageResults \mathrel{\widehat{=}} PackageResult \mathbin{⨟} PackageResults \;[]\; [matches = \{\}]$$

A *CASE* is the simplest of all GReAT components. The Execute function of the case takes each input packet and calls the pattern matcher. The matches returned by the pattern matcher are then filtered using the guard expression. All successful matches are again packaged to form the output packets. The *CASE* is used only within a *TEST* component. *TEST* and *CASE* together are used together, to form a conditional execution and branching construct.

$$
\begin{array}{l}
\textit{CASE} \\
\hline
\textit{PRIMITIVE\_UNIT} \\
\textit{Execute} \mathrel{\widehat{=}} \textit{MakeInitialPartialMatches} \mathbin{⨟} \textit{PatternMatcher} \mathbin{⨟} \\
\qquad\qquad \textit{EvaluateGuard} \mathbin{⨟} \textit{PackageResults} \\
\qquad\quad [\!] \\
\qquad\quad [\textit{inputPackets} = \{\}]
\end{array}
$$

The execution of a $RULE$ is similar to that of a $CASE$. The exception is that in a $RULE$, after the matches are filtered using the guard, the matches are used to perform actions on the host graph. These actions can create and/or delete vertices and edges. After these actions are performed, the attribute mapping specification is used by *PerformAttributeMapping* operation to fill in and/or modify the attributes of graph vertices and edges. Again, for the sake of brevity, *PerformAttributeMapping* is described in English.

$$
\begin{array}{l}
\textit{RULE} \\
\hline
\textit{PRIMITIVE\_UNIT} \\
\textit{ACTION} == \{\textit{bind}, \textit{create}, \textit{delete}\} \\
\quad \textit{vertexAction} : [\textit{VERTEX} \nrightarrow \textit{ACTION}] \\
\quad \textit{edgeAction} : [\textit{EDGE} \nrightarrow \textit{ACTION}] \\
\quad \textit{attributeMapping} : \mathbb{P}\, \textit{ASSIGNMENT\_STATEMENTS} \\
\quad \textit{PerformAction} \\
\quad \forall\, \textit{match} \in \textit{matches} \bullet \\
\qquad \forall (v, a) \in \textit{vertexAction} \bullet \textit{ACTION} = \textit{create} \\
\qquad\quad \textit{hostGraph.vertices}' = \textit{hostGraph.vertices} \cup \textit{new\_v} : \textit{VERTEX} \\
\qquad\qquad \wedge\ \textit{new\_v.name} = \textit{v.name} \wedge \textit{new\_v.type} = \textit{v.type} \\
\qquad \forall (e, a) \in \textit{edgeAction} \bullet \textit{ACTION} = \textit{create} \\
\qquad\quad \textit{hostGraph.edges}' = \textit{hostGraph.edges} \cup \textit{new\_e} : \textit{EDGE} \wedge \\
\qquad\qquad \textit{new\_e.name} = \textit{e.name} \wedge \textit{new\_e.type} = \textit{e.type} \\
\qquad \forall (v, a) \in \textit{vertexAction} \bullet \textit{ACTION} = \textit{delete} \\
\qquad\quad \textit{hostGraph.vertices}' = \textit{hostGraph.vertices} - v \\
\qquad \forall (e, a) \in \textit{edgeAction} \bullet \textit{ACTION} = \textit{delete} \\
\qquad\quad \textit{hostGraph.edges}' = \textit{hostGraph.edges} - e \\
\quad \textit{PerformAttributeMapping} \\
\quad \forall\, \textit{match} \in \textit{matches} \bullet \\
\quad \text{Apply attribute matching statements on the match.} \\
\textit{Execute} \mathrel{\widehat{=}} \textit{MakeInitialPartialMatches} \mathbin{⨟} \textit{PatternMatcher} \mathbin{⨟} \\
\qquad\qquad \textit{EvaluateGuard} \mathbin{⨟} \textit{PerformAction} \mathbin{⨟} \\
\qquad\qquad \textit{PerformAttributeMapping} \mathbin{⨟} \textit{PackageResults}
\end{array}
$$

Sequential execution of expressions is expressed using the *SEQUENCE* class. This class maps ports of one *UNIT* to ports of another *UNIT*. *SEQUENCE* is usually used to map from the output interface of a *UNIT* to the input interface of another *UNIT*. However, we will see that in compound units *SEQUENCE* is also used to map the input interface of the compound unit to the input interface of contained units.

SEQUENCE

$$
\begin{array}{l}
inputInterface : INTERFACE \\
outputInterface : INTERFACE \\
in2out : \mathbb{P}[PORT \rightarrowtail PORT] \\
\hline
\forall\, i \in dom(in2out) \bullet i \in inputInterface \\
\forall\, i \in range(in2out) \bullet i \in outputInterface
\end{array}
$$

PassAllongPacket

$$
\begin{array}{l}
srcPackets? : \mathbb{P}\, PACKET \\
dstPackets! : \mathbb{P}\, PACKET \\
\hline
dstPackets = \{dstPacket : PACKET \mid \forall\, srcPacket \in srcPackets \\
\qquad\qquad \exists\, dstPacket \bullet \\
\qquad \forall (psrc, pdst) \in in2out \wedge (psrc, hv) \in srcPacket \bullet \\
\qquad (pdst, hv) \in dstPacket\}
\end{array}
$$

TEST

UNIT

$$
\begin{array}{l}
cases : seqCASE
\end{array}
$$

GetNextSequece

$$
\begin{array}{l}
case! : CASE \\
sequence! : SEQUENCE \\
srcPackets! : \mathbb{P}\, PACKET \\
\hline
cases' = cases \frown \langle case! \rangle \\
\exists\, sequence! \in sequences \bullet \\
\quad sequence!.inputInterface = case.inputInterface \wedge \\
\quad sequence!.outputInterface = case.outputInterface \\
srcPackets = inputPackets
\end{array}
$$

$$
\begin{array}{l}
Execute \mathrel{\widehat{=}} GetNextSequence \fatsemi sequence.PassAllongPacket \fatsemi case.Execute \\
\qquad\qquad \mathbin{[\!]} \\
\qquad\qquad [cases = \{\}]
\end{array}
$$

TEST is a *UNIT* that provides the language with a conditional execution and branching construct. A *TEST* contains an ordered sequence of *CASE*-es.

The execution semantics of the $TEST$ is that each $CASE$ within a $TEST$ is executed in order, starting from the first case in the sequence.

$COMPOUND\_UNIT$ is the base class of the two compound objects in GReAT: (1) $BLOCK$ and (2) FOR_BLOCK. These blocks are useful for encapsulating complex rule sequences. The only difference between a block and for block is in their execution semantics. The compound expressions use a stack machine semantics and thus have a *readyUnitStack* with push and pop operations.

$$
\begin{array}{l}
\textbf{COMPOUND\_UNIT} \\
\quad UNIT \\
\quad subExpressions : \mathbb{P}\, EXPRESSION \\
\quad sequences : \mathbb{P}\, SEQUENCE \\
\quad readyUnitStack : Seq UNIT \\
\\
\quad \textbf{PushUnit} \\
\qquad readyUnit? : UNIT \\
\qquad \hline \\
\qquad readyUnitStack' = readyUnitStack \frown \langle readyUnit \rangle \\
\\
\quad \textbf{PopStack} \\
\qquad readyUnit! : UNIT \\
\qquad \hline \\
\qquad readyUnitStack = readyUnitStack' \frown \langle readyUnit \rangle
\end{array}
$$

The $BLOCK$ is the simplest compound unit. It encapsulates a set of units along with their sequencing. The execution of the block starts with the *StartBlock* function that finds all the units that have a sequence from the input interface of the $BLOCK$. All these units are added to the *readyUnitStack* along with a copy of the input packets set of the BLOCK. The execution is then defined to pop the top of the stack, execute the unit with the input packets, use the sequence from the current rule to get the new ready-to-fire units, and add these to the stack. This process is repeated until the *readyUnitStack* is empty. Whenever a unit that has executed is connected to the output interface of the $BLOCK$, the outputs are copied to the output of the $BLOCK$.

$$
\begin{array}{l}
\textbf{BLOCK} \\
\hline
UNIT \\
\quad \begin{array}{|l}
\hline
tempSequence : \mathbb{P}\, SEQUENCE \\
tempUnits : \mathbb{P}\, UNIT \\
tempPackets : \mathbb{P}\, PACKET \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
StartBlock \\
srcPackets! : \mathbb{P}\, PACKET \\
\hline
srcPackets! = inputPackets \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
GetNextReadyUnits \\
interface? : INTERFACE \\
\hline
tempSequence' = \{sequence : SEQUENCE \mid \\
\qquad sequence.inputInterface = interface\} \\
tempUnits' = \{unit : UNIT \mid \forall\, seq \in tempSequence \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
MakeReadyUnit \\
readyUnit! : UNIT \\
sequence! : SEQUENCE \\
srcPacket! : PACKET \\
\hline
tempUnits = tempUnits' \frown \langle readyUnit \rangle \\
tempSequence = tempSequence' \frown \langle sequence \rangle \\
srcPackets = tempPackets \\
\hline
\end{array} \\
PushReadyList \mathrel{\widehat=} MakeReadyUnit \mathbin{⨾} PushUnit ⨾ \\
\qquad sequence.PassAllongPacket \\
\qquad [\!] \\
\qquad [tempUnits = \{\}] \\
Iteration \mathrel{\widehat=} PopUnit ⨾ \\
\qquad unit.Execute[tempPackets / unit.outputPackets, \\
\qquad\qquad interface / unit.outputInterface] ⨾ \\
\qquad GetNextReadyUnits \mathbin{⨾} PushReadyList \\
\qquad [\!] \\
\qquad [readyUnitStack = \langle\rangle] \\
Execute \mathrel{\widehat=} StartBlock \mathbin{⨾} GetNextSequence[inputInterface / interface] \\
\qquad ⨾ Iteration \\
\hline
\end{array}
$$

The *FOR_BLOCK* is similar to the *BLOCK* with a subtle difference. The execution of the *FOR_BLOCK* starts the unit execution stack with only the first input packet. When the stack is empty the process is repeated with the next packet until all packets are exhausted. The *FOR_BLOCK* provides a depth first execution of all the contained units while the *BLOCK* provides a breadth first execution.

$$
\begin{array}{|l}
\hline
\textit{FOR\_BLOCK} \\
\textit{BLOCK} \\
\quad \begin{array}{|l}
\hline
\textit{StartForBlock} \\
\textit{srcPackets}! : \mathbb{P}\,\textit{PACKET} \\
\hline
\textit{inputPackets}' = \textit{inputPackets} - \textit{inputPacket} \land \\
\textit{srcPacket}! = \{\textit{inputPacket}\} \\
\hline
\end{array} \\
\textit{Execute} \mathrel{\widehat{=}} \textit{StartForBlock} \fatsemi \textit{GetNextSequence}[\textit{inputInterface}/\textit{interface}] \\
\qquad\qquad \fatsemi \textit{Iteration} \\
\qquad\quad [\!] \\
\qquad\quad [\textit{inputPackets} = \{\}] \\
\hline
\end{array}
$$

5 The run-time system architecture of GReAT-E

The Graph Rewrite and Transformation Engine (GReAT-E) 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 GReAT-E system takes the input graph, applies the transformations to it, and generates the output graph. Inputs to the GReAT-E are (1) the UML class diagrams for the input and output graphs (i.e. the metamodels), (2) the transformation specification and (3) the input graph, with the appropriate initial match(es) selected. GReAT-E executes the rules according to the sequencing and produces an output graph based upon the actions of the rules.

The architecture of the run time system is shown in Figure 6. GReAT-E accesses the input and output graph with the help of a common API that allows the traversal of the input and the construction of the output graph using a high-level API. The rewrite rules are stored using a common data structure, which is constructed from the visual models of transformation steps and can be accessed using yet another common API. The GReAT engine is metamodel-driven, and uses a reflective/persistent data structure package, called UDM[30].

GReAT-E 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 from the specification of the transformation, and for each rule it calls the RE. The RE internally calls the PM with the "pattern" portion 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 Sequencer traverses the transformation rules according to the sequencing information to determine the next rule to execute. It also has to evaluate test cases (if they are used) to determine the next rule for execution.

The Pattern Matcher finds the subgraph(s) in the input graph that are isomorphic to the pattern specification. In case of a match, it binds a vertex/edge

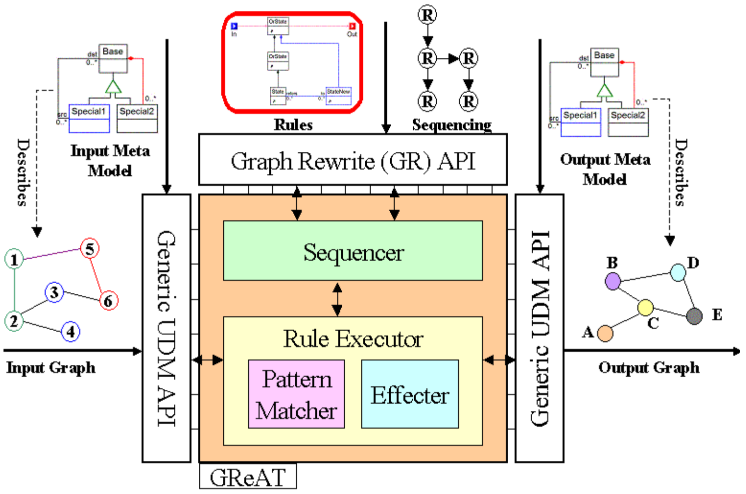

Figure 6: The GReAT Engine

in the pattern to a corresponding vertex/edge in the input graph. The matcher starts with an initial binding supplied to it by the Sequencer. Then it incrementally extends the bindings until 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 in binding all such edges,it then finds an edge with one vertex bound and binds the edge and its unbound vertex. This process is repeated until all the vertices and edges are bound.

The output generator, which is called after the matches are found, creates and extends the output graph corresponding to each rule. The generator determines whether new objects should be created, or existing objects referenced, whether there is a need to insert new associations, and how attributes of output objects and associations have to be calculated.

6 An Example

The creation of a CBS rarely occurs in one step. Rather, several design iterations take place, and different tools are used at different stages of the design. Design tools that can interoperate can increase the productivity of designers by not requiring them to perform the manual entry of the system models for each tool, but rather using the same models for all design tools. This necessitates the use of model transformations in a tool chain.

The benefits of a working tool chain can be demonstrated in a design problem as follows. Suppose we design a system that requires the canonical client/server relationship to implement a data publishing/subscription service. The exact implementation is not important to the client (e.g., the user does not care that the mailing list to which he is subscribed is hosted on one machine or several), but the implementation may be important to the designers, who are interested in the performance of the system, as well as robustness, access control, and others.

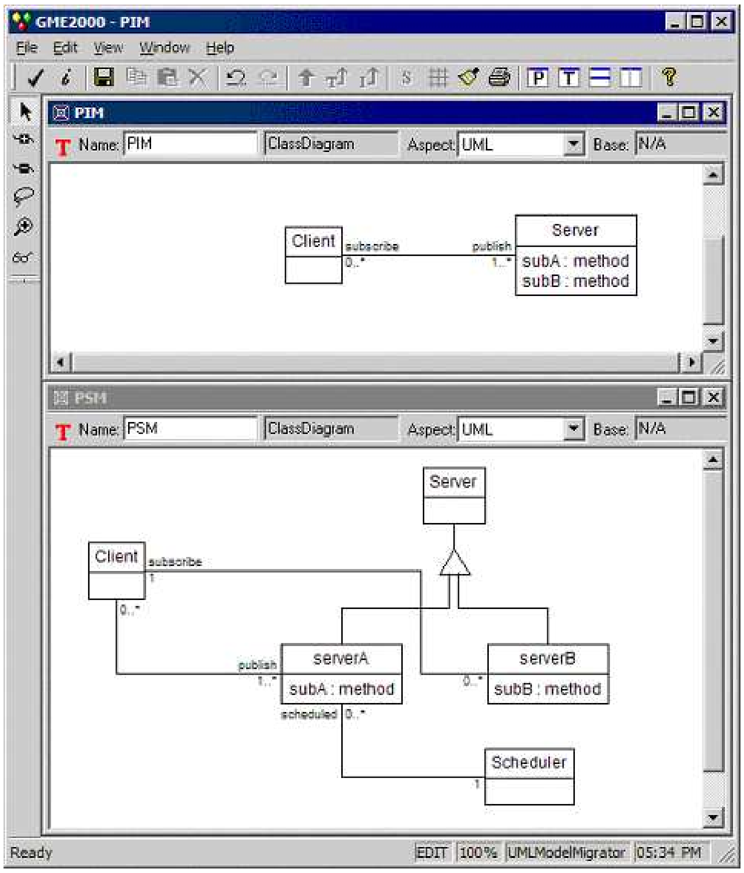

Figure 7: A sample input and output graph for the Publish/Subscribe problem

Figure 7 shows two different models of a client/server framework. The top is the interface model - the information important to a subscriber. The subscriber can subscribe to one or more publishers, and the publisher must be able to notify zero or more subscribers when updates are available. According to this diagram, the only players in the CBS are a publisher and subscriber. The bottom model shows a more advanced design of the same CBS. In this design, the subscriber can still subscribe to one or more publishers, but the publisher does not directly notify the subscribers in the event of an available update. Instead, the publisher server delegates this responsibility to a different machine, which in turn publishes the available data, as determined by a scheduler.

An observant designer of CBS will notice the similarities of the second design with the first. It is possible to use a graph transformation technique to transform models that were built using the first formal specification into models that use the second formal specification. In this way the design is specialized, and the design artifacts produced in previous evolutions are modified to pass down the tool chain. The algorithm for migrating from the first to the second design is as follows:

1. For each server, create two servers, one of type A, and one of type B
2. Create a scheduler that will be in charge of executing server A when data becomes available for publication
3. Create a new client that replaces the old one

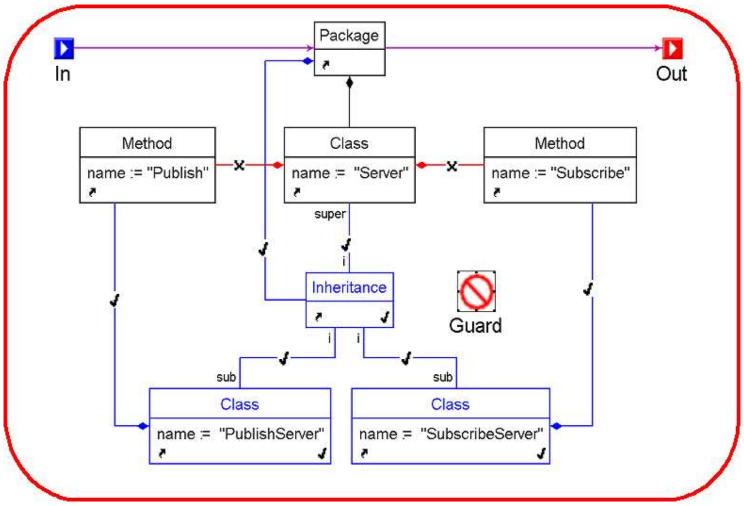

Figure 8: CreateServers: A rule to Convert a publish and subscribe server into to derived servers

The formal description of this algorithm is found in Figure 8 and Figure 9. The sequence of the algorithm is shown in the top of the figure (the connected

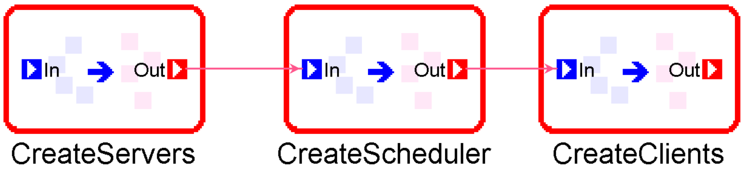

Figure 9: A sequence of rules that solve the Publish/Subscribe problem

rewriting rules). Each of these rewriting rules contains a specification that formalizes exactly how models are to be transformed. The bottom portion of Figure 8 shows how the two servers are created. A metamodel of UML is used to specify these transformation rules. The rule states that for a given package find all server classes that have both a subscribe and publish methods. For each such server class, two new classes called SubscribeServer and PublishServer are created. The newly created classes are derived from the original class. The publish and subscribe methods are removed from the original class and added to the respective classes. Then any vertex or edge with the "X" mark or with no mark will be matched. Then the objects with "tick" mark will be created and those with X will be removed.

This example shows that when a design evolution occurs, models created in earlier stages of the design need not be abandoned or rebuilt simply due to the complexities of transforming the models. GReAT-E can be used to rapidly produce a translation that will enable multiple design evolutions throughout the development of a CBS.

7 Conclusions

In this paper we have illustrated how a metamodel-based graph transformations can be used in the construction of CBS. We have also shown the formal semantics of the a graph transformation language that can be used to express model transformation algorithms. We claim that the design transformation process specified this way is formal, and it assigns a semantics to the input models in terms of the target domain. We believe that one can also formally reason about the transformation programs, prove interesting properties about them, and verify their correctness with respect to some criteria. These types of formally-specified model transformations are also useful in various other steps, for instance when transforming models into artifacts suitable for verification.

Currently we have a well-defined method for building model transformers, and we have created a set of tools that allow experimentation with the approach. In the next stage, we will look into addressing the performance aspect of the transformations, and try to generate code from the transformation specs (thus bypassing the need for the GReAT-E-like interpreter).

Formally specified transformations on models are a fruitful area of research. Graph transformations, in addition to providing a very high-level programming language for specifying complex manipulations, offer the opportunity to reason about those algorithms. One of the goals of applying formal techniques in CBS is to achieve the "correct-by-construction" property. It is conceivable that if the constructions steps are formally specified, then the correctness of a design can be verified based on the correctness of the steps. We believe that the technique we have described in this paper provides the first steps in this direction, but further research is necessary to provide a full solution.

References

[1] Karsai G., Nordstrom, G., Ledeczi A., Sztipanovits J.: "Towards Two-Level Formal Modeling of Computer-Based Systems, Journal of Universal Computer Science"; Vol. 6, No. 11, pp. 1131-1144, November, 2000.
[2] Matlab/Simulink/Stateflow tools from Mathworks,Inc.
[3] "POLIS: A Framework for Hardware-Software Co-Design of Embedded Systems"; available from http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html
[4] RHAPSODY, available from http://www.ilogix.com
[5] McMillan K. L.: "Symbolic Model Checking: an approach to the state explosion problem"; CMU Tech Rpt. CMU-CS-92-131.
[6] KRONOS, available from http://www-verimag.imag.fr/TEMPORISE/kronos/
[7] Maggiolo-Schettini A., Peron A.: "Semantics of Full Statecharts Based on Graph Rewriting"; Springer LNCS 776, 1994, pp. 265–279.
[8] Sztipanovits J. and Karsai G.: "Model-Integrated Computing"; Computer, Apr. 1997, pp. 110-112
[9] Ledeczi A., et al. :"Composing Domain-Specific Design Environments"; Computer, Nov. 2001, pp. 44-51.
[10] Rumbaugh J., Jacobson I., and Booch G.: "The Unified Modeling Language Reference Manual"; Addison-Wesley, 1998.
[11] "The Model-Driven Architecture"; http://www.omg.org/mda/ , OMG, Needham, MA, 2002.
[12] "Request For Proposal: MOF 2.0 Query/Views/Transformations"; OMG Document: ad/2002-04-10, 2002, OMG, Needham, MA.
[13] 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.
[14] Rozenberg G. (ed.): "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations"; Vol.1-2. World Scientific, Singapore, 1997
[15] Blostein D., Schürr A.: "Computing with Graphs and Graph Transformations"; Software - Practice and Experience 29(3): 197-217, 1999.
[16] Assmann U.: "How To Uniformly Specify Program Analysis and Transformation"; in: 6th Int. Conf. on Compiler Construction (CC '96), T. Gyimthy (rd.), Lect. Notes in Comp. Sci., Springer-Verlag, Linkping, Sweden, 1996.
[17] Schürr A.: "PROGRES for Beginners"; RWTH Aachen, D-52056 Aachen, Germany.

[18] Taentzer G.: “AGG: A Tool Enviroment for Algebraic Graph Transformation”; Proc. of Applications of Graph Transformation with Industrial Relevance, Kerkrade, The Netherlands, LNCS,Springer, 2000.
[19] Maggiolo-Schettini A., Peron A.: “Semantics of Full Statecharts Based on Graph Rewriting”; Springer LNCS 776, 1994, pp. 265–279.
[20] Kiczales G., Lamping J., Lopes C. V., Maeda C., Mendhekar A., Murphy G.: “Aspect-Oriented Programming”; ECOOP’97, LNCS 1241, Springer. (1997)
[21] Simonyi C.: “Intentional Programming: Asymptotic Fun?”; Position Paper, SDP Workshop Vanderbilt University, December 13 - 14, 2001. http://isis.vanderbilt.edu/sdp
[22] Milicev D.: “Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments,”; IEEE Transaction on Software Engineering, Vol. 28, No. 4, April 2002, pp. 413-431
[23] Wai-Ming Ho, Jean-Marc Jzquel, Alain Le Guennec, and Franois Pennaneac’h.: “UMLAUT: an extendible UML transformation framework,”; Proc. Automated Software Engineering, ASE’99, Florida, October 1999.
[24] David H. Akehurst: “Model translation: A uml-based specification technique and active implementation approach”; PhD thesis, Computer Science at Kent University (UK), December 2000.
[25] Tony Clark, Andy Evans, Stuart Kent: “Engineering Modelling Languages: A Precise Meta-Modelling Approach”; FASE 2002: 159-173
[26] Tony Clark, Andy Evans, Stuart Kent: “The Metamodelling Language Calculus: Foundation Semantics for UML”; FASE 2001: 17-31.
[27] Lemesle, R.: “Transformation Rules Based on Meta-Modeling EDOC”; ’98, La Jolla, California, 3-5, November 1998, pp.113-122.
[28] Heckel, R. and Kster, J. and Taentzer, G.: “Towards Automatic Translation of UML Models into Semantic Domains”; Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002), Grenoble, France, 2002, pp. 11 - 22.
[29] Karsai G.: “Tool Support for Design Patterns”; New Directions in Software Technology 4 Workshop, December, 2001. Available from http://www.isis.vanderbilt.edu.
[30] Bakay A., Magyari E.: “The UDM Framework”; www.isis.vanderbilt.edu/Projects/MoBIES/.
[31] 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.
[32] 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.
[33] A. Radermacher :“Support for Design Patterns through Graph Transformation Tools”; Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, Sep. 1999.
[34] 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.
[35] U. Assmann: “Aspect Weaving by Graph Rewriting”; Generative Component-based Software Engineering (GCSE), p. 24-36, Oct 1999.
[36] 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.
[37] H. Gottler: “Diagram Editors = Graphs + Attributes + Graph Grammars”; International Journal of Man-Machine Studies, Vol 37, No 4, Oct. 1992, pp. 481-502.
[38] J. Loyall and S. Kaplan: “Visual Concurrent Programming with Delta-Grammars”; Journal of Visual Languages and Computing, Vol 3, 1992, pp. 107-133.

[39] 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.

[40] Roger Duke, Gordon Rose and Graeme Smith: "Object-Z: a Specification Language Advocated for the Description of Standards"; TR 94-95, December 1994, Software Verification Research Centre, Department Of Computer Science, The University Of Queensland, Queensland 4072, Australia.

[41] Agrawal A., Karsai G., Ledeczi A.: An End-to-End Domain-Driven Software Development Framework, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Domain-Driven Development Track, Anaheim, CA, October, 2003.