Generative Programming via Graph Transformations in the Model-Driven Architecture

Aditya Agrawal[1], Tihamer Levendovszky, Jon Sprinkle, Feng Shi, and Gabor Karsai[2]
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

The Model-Driven Architecture of OMG envisions a development paradigm where designers create a Platform-Independent Model (PIM) of the design, which is then refined into a Platform-Specific Model (PSM). This paper argues that this approach lends itself well to generative programming techniques, and that tools are needed to support this transformation. The paper shows how a technique based on graph transformations could be applied to automate the process, as well as make it user-extendible.

Introduction

The bold vision outlined in Model-Driven Architecture of OMG [1] places great emphasis on the use of modeling in the software development process. What has been started in the Unified Modeling Language, MDA is taking to its logical next step: the fully model-based development process. In this process models are created for capturing not only requirements, but also designs and implementations. The models are not merely artifacts of documentation, but "living documents" that are transformed into implementations. This view radically extends the current prevailing practice of using UML: UML is used for capturing some o the relevant aspects of the software, and some of the code (or its skeleton) is automatically generated, but the main bulk of the implementation is developed by hand. MDA, on the other hand, advocates the full application of models, in the entire life-cycle of the software product.

Note that the crucial components in a model-based development process are (1) the tools used in creating the models, (2) the tools used in transforming the models into some executable form, and (3) the platform on which the executable form of the models is executed. For modeling tools, there is a wide variety available, ranging from (pure) UML modeling tools [2] to meta-programmable modeling environments [3]. Also, there are industry-strength platforms for execution, ranging from middleware packages like CORBA of COM to sophisticated frameworks like .NET or EJB. However, the connection between modeling tools and execution platforms: the model transformation technology is an active area or research.

We argue that current model transformation technology as manifested in industrial tools has not achieved its potential. Code generation is a special case of model transformation, where the product is executable code. However, code generators that produce skeleton code from UML models tend to be simplistic, while code generators for more focused domains (like Mathworks' Real-Time Workshop [4]) fare better in their own domain. However, the practical use of generators is not as widespread as the potential impact they can make on software development. Arguably, the reason for this is economical: it is very expensive to create a good generator, and the average software developer may find It simpler to create code by hand than by developing a generator first and then using it systematically.

Arguably, the model transformation approach will succeed only if it becomes easy to create model transformation tools that can produce high-quality products (including code). There seems to be a need for the configurability of generators: as developers tend to be very skilled in domain-specific optimizations and —by applying these— can produce high quality code by hand. This need for configurability is also present in the context of the MDA.

[1] aditya.agrawal@vanderbilt.edu
[2] gabor@vuse.vanderbilt.edu

MDA introduces the concept of the Platform Independent Model (PIM) and the Platform Specific Model (PSM). A PIM is an abstract model of the software design that omits any platform (i.e. implementation-specific details). A PSM, on the other hand, is another model that includes implementation-specific details. The PSM is obviously dependent on the PIM, and arguably one (the PSM) can be derived automatically from the other one (the PIM). However, this derivation process is highly domain- and application-specific: different domains might need different methods for implementing the derivation.

This paper introduces a technique and a prototype tool for creating highly configurable model transformation tools that can be applied in the MDA context. The technique (and the tool) is based on a well-established theoretical framework based on graph transformations. The paper first briefly reviews graph transformation techniques and their applications, next it casts the model transformation problem in graph-oriented framework, then it discusses implementation techniques and provides an illustrative example, and concludes with a summary and an outline for further research needed.

Backgrounds

Graph grammars and graph rewriting [5][6] have been developed during the last 25+ years as techniques for formal modeling and tools for very high-level programming. Graph grammars are the natural extension of the generative grammars of Chomsky 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 grammar. One can also define replacement rules on strings, which consist of a pattern and a replacement string. The replacement rule's pattern is matched against an input string, and the matched sub-string is replaced with the replacement string of the rule. Similarly, 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 similar to the application of a string rewriting rule on strings, only the matching sub-graph is replaced with another graph. For precise details see [5].

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 [10], as well as various concurrency models [5]. Several tools —including programming environments— have been developed [8][9] 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) graph rewriting rules can be compiled into efficient code. Programming via graph transformations has been applied in some domains [6] 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 they can be also applied in the practice.

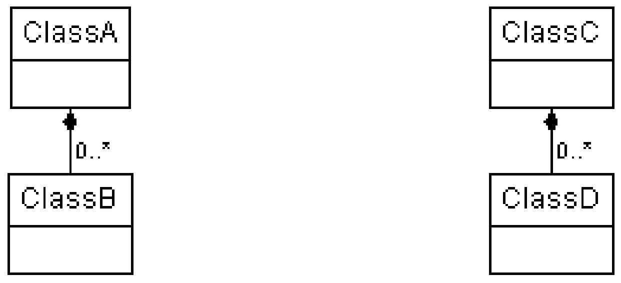
The need for techniques for model transformations has been recently recognized in the UML world. For examples, see [13], [14], [15], [18], and [19]. Model transformation is an essential tool for many applications, including translating abstract design models into concrete implementation models [18], for specification techniques [15], translation of UML into semantic domains [19], and even for the application of design patterns [20]. The new developments in UML (see [16], [17]) emphasize the use of meta-models, and provide solid foundation for the precise specification of semantics. Related efforts, like aspect-oriented programming [11] or intentional programming [12] could also benefit from using transformation technique 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.

The Transformation Language

In MDA, the transformation engine that transforms the domain specific PIM into implementation-specific PSM plays a central role. Furthermore, the PSM could also be translated into some executable form. We argue that the PIM to PSM transformation is domain specific, and to speed up the development cycle it is important to be able to develop these transformers in as less time as possible. Furthermore, in a product-line situation, the PSM and the PIM can be considered as "sentences" in some Domain-Specific Modeling Language (DSML). Therefore, the PIM to PSM transformation is a transformation between two DSMLs. In this paper we will focus on a generalized graph transformation system called the Graph Rewrite Engine (GRE) that is able to transform models based upon a description of a transformation provided to it. The transformation itself is specified in a visual language.

Before describing the transformation language let us first introduce some terminology that we will be using extensively in this paper. A *Metamodel* is the UML class diagram that describes a domain specific modeling language (DSML). The word *Paradigm* is interchangeable with DSML. *Models* are sentences of a particular modeling language. For example, UML instance diagrams can be called models. *Input Graph* or *Input Model* refers to the models to be transformed by the transformer. *Output Graph* or *Output Model* refers to the output of the transformer. Usually the metamodel describing the input graph differs from that of the output graph.

The transformation language used by GRE consists of three major components (1) *rules*, (2) *test-cases,* and (3) *sequencing* for the rules. A rule is an atomic transformation operation, which describes a single transformation step. A rule consists of the basic parts: (1) input subgraph (also referred to as the pattern, or the LHS), (2) output subgraph (also called the RHS), (3) mapping of input graph elements to output graph elements and (4) actions. A rule specifies the actions to perform if the described input subgraph exists in the input graph. One of the features of this language is that it allows one to associate input vertices and edges (of the input graph) with output vertices and edges (of the output graph). Thus an input vertex can reference to a corresponding vertex in the output graph. In order to apply a rule we need to find the input subgraph in the input graph. However, it is well-known that subgraph isomorphism is NP-complete with order complexity $O(n_1^{n_2})$, where n_1 are the number of vertices of the host graph and n_2 is the number of vertices in the pattern graph. However, for a particular rule the pattern graph will not change and thus n_2 can be considered a constant and thus making the search actually polynomial, though the exponent of the polynomial can vary from one rule to another. Since the time complexity is an exponent in terms of the pattern the matching algorithm is an expensive operation. However, to avoid this problem we allow users to specify initial bindings between some pattern vertices and input graph vertices. This helps to reduce the size of the host graph to conceder and the exponent is reduced to only the number of unbound vertices in the pattern. Another issue is the sequencing of rule and their execution. This is left to the user and he/she can specify the order of execution for these rules. The user can also specify different sequences based upon conditional *test-case* steps, which differ from the rules as they have only patterns but no actions. Furthermore, input and output graph objects can be passed from one rule to another one. This is necessary, as each rule needs to have at least one pattern vertex bound to the input graph for efficiency. Thus, by choosing which objects to pass along the user can choose the traversal of the graph. For instance, the user could choose depth first traversal or he/she could choose to traverse the spanning tree of the graph.



(a) Input metamodel **(b) Output metamodel**

Figure 1: Input and Output metamodels

Let us consider a simple example. The input and output metamodels are shown in Figure 1. Suppose, the transformation needs to create an object graph such that for each instance of *ClassA* in the input there will be a corresponding instance of *ClassC* in the output. Similarly for each *ClassB* instance a *ClassD* instance should be created. The starting point of the transformation is an instance of *ClassA* in the input model. The transformation will look like Figure 2. *Init* is the starting point of the transformation and it refers to the instance of *ClassA*. This is then passed to Rule1. Rule1 specifies a pattern of the input graph and specifies that the corresponding objects should be created in the output graph. There are edges form the input graph to the output graph; these are called the action edges. The *CreateNew* action specifies that a new object should be created in the output graph. The action edge will also establish a reference between the source and destination object. Thus, in this example the particular instance of *ClassA* that was matched will have a reference to the newly created instance of *ClassC*. This is useful for subsequent operations: once the “image” of an input object is created, subsequent rules can access that. Another type of action edge is called *Refer* that asserts that the output object has been previously created and the same object is to be used.

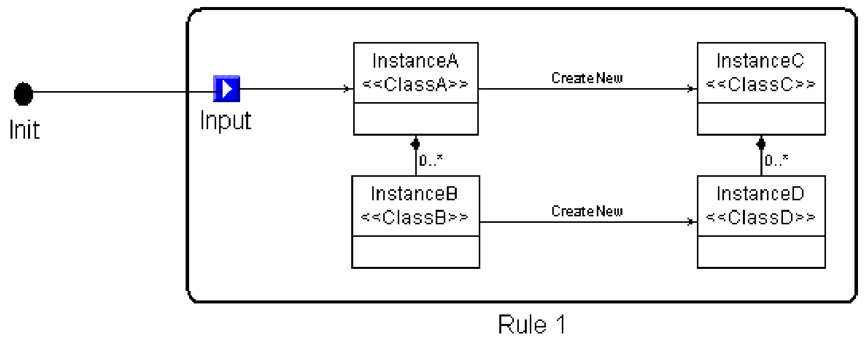


Figure 2: The transformation

The run-time system architecture of GRE

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

The architecture of the run time system is shown in Figure 3. The GRE accesses the input and output graph with the help of a generic UDM API (explained in detail in the next section) that allows the traversal of input and output graph. The rewrite rules are stored using yet another DSML, called Graph Rewrite (GR) (also explained in the next section) and can be accessed using the GR specific UDM API.

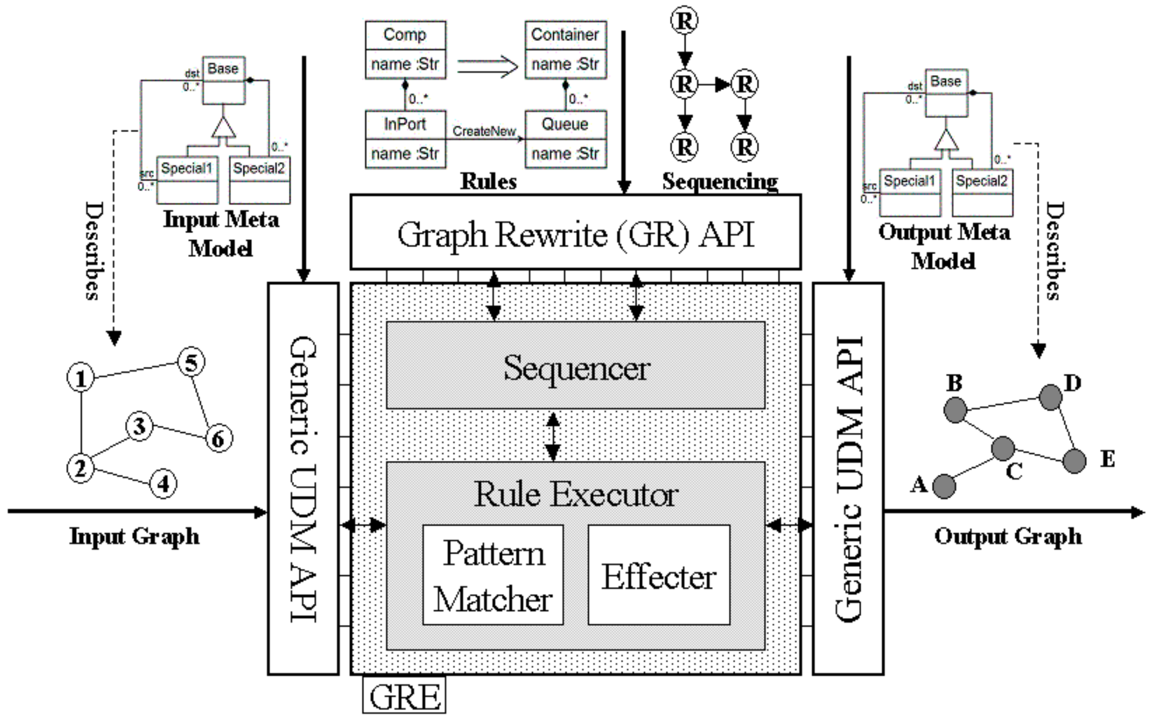


Figure 3: Run time architecture of the Graph Rewrite Engine

The GRE is composed of two major components, (1) Sequencer, (2) Rule Executor (RE). The Rule Executor is further broken down into (1) Pattern Matcher (PM) and (2) Effector (or “Output generator”). The Sequencer determines the order of execution for the rules from the specification of the transformation, and for each rule it calls the RE. The RE internally calls the PM with the LHS of the rule. The matches found by the PM are used by the Effector to manipulate the output graph by performing the actions specified in the rules.

The 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 high-level algorithm of the Sequencer is given in Figure 4.

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 Figure 5.

Function Name : Sequencer

Inputs : 1. Set of Rules
2. Sequencing
3. Input graph

Outputs : 1. Output graph

Output graph = function RuleTraversal (Set of Rules, Sequencing, Input graph)

```
{   R = GetStartRule(Set of Rules)
    Output graph = new empty Graph        /* Using the Output meta-model */
    while( $R_{NEXT}$ = GetNextRule(R)) {
        PropagateBindings(R, $R_{NEXT}$)       /* Propagate existing bindings */
        R = $R_{NEXT}$
        $R_{LHS}$ = GetLHS(R)
        B = GetRuleBinding(R)
        Set of Matches = PatternMatcher($R_{LHS}$, Input graph, B) /* Find match */
        Effector(R, Set of Matches, Output graph) /* Create/modify output as needed */
    }
    return Output graph
}
```

Figure 4: High level Algorithm for Sequencer

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, if there is a need to insert new associations, and how attributes of output objects and associations has to be calculated.

```
Function Name  : PatternMatcher

Inputs         :  1. Pattern
                  2. Input graph
                  3. Partial Match
Outputs        :  1. Set of Matches

Set of Matches = function PatternMatcher (Pattern, Input graph, Partial Match)
{   edge = get pattern edge with both vertices having valid bindings
    while(edge exists) {
        if(corresponding edge doesn't exists between host graph vertices) then return the empty set
        Bind pattern and host graph edge
        Delete the pattern edge
        edge = get pattern edge with both vertices having valid bindings
    }

    edge = get pattern edge with one vertex bound to host graph
    If(edge exists) {
        vertices = vertices of the host graph adjacent to the bound vertex
        New Pattern = copy of Pattern
        Delete edge from New Pattern
        For(Each vertex of the set of vertices) {
            New Match = partial Match + new binding(unbound pattern vertex, vertex)
            Return Match = PatternMatcher(New pattern, Host graph, New match)
            Add Return Matches to Set of Matches
        }
        Return Set of Matches
    }

    If(all patern edges are bound) {
        Add Partial match to Set of Matches
        Return Set of Matches
    }
    reutrn empty set
}
```

Figure 5: High-level Algorithm for Pattern Matching

The Implementation Framework

The core techniques applied in the development of the GRE prototype rely on the UDM package [21] and an intermediate form for the rewriting models.

The **U**niversal **D**ata **M**odel (UDM) is a meta-programmable package [21] that includes a development process and a set of supporting tools to generate C++ accessible interfaces from UML class diagrams of data structures. The generated APIs can use a variety of implementations, including memory-based, XML-based, and one based on te internal data structures of a visual modeling environment. All the implementations are accessed through a uniform, generic interface, thus the actual backend is transparent to the programmer. UDM provides a convenient programmatic access and can be used to build generators or translators for different data structures described in UML class diagrams. Note that the programmer has two different interfaces: one of them is a domain-specific one, which is generated based on the UML class diagrams, and another, generic one, which allows manipulating objects using symbolic names (class names, attribute names, association role names, etc.). The typical process of using the UDM is as follows:

- A UML class diagram (metamodel) is created in either of the two supported modeling tools (Visio or GME). The UML class diagram is the converted into an XML representation with the help of a UDM tool.
- The XML file is then used to generate a C++ API (pair of a source and a header file) specific to the particular class diagram, as well as an XML DTD (to be used in the XML backend). The generated C++ files are then compiled and linked with the generic UDM library and one of the implementation specific UDM libraries. The user can easily create, modify and traverse object graphs described by the class diagram.
- Alternatively, the generated XML file can be directly used to create, modify and traverse object graphs corresponding to the particular metamodel using the generic UDM API

The Tool chain in the UDM process is described below. Figure 6 shows a simplified UDM based development scenario. Note that UDM includes a reflection package, as the meta-models (obtained from the UML class diagram) are explicit in the form of initialized data structures.

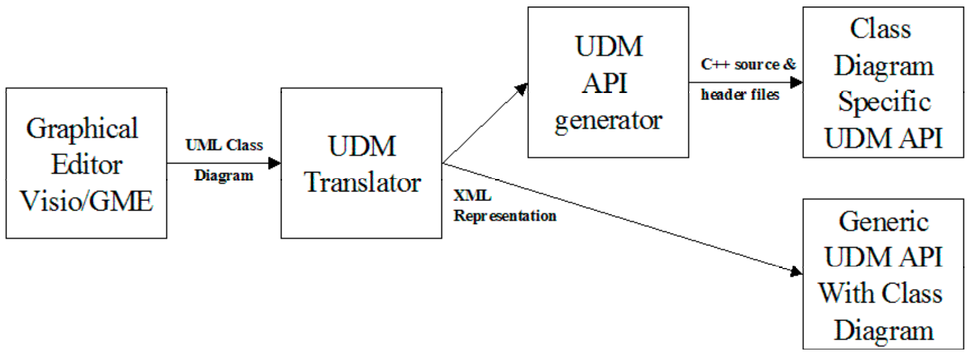

Graphical Editor Visio/GME
UML Class Diagram
UDM Translator
UDM API generator
C++ source & header files
Class Diagram Specific UDM API
XML Representation
Generic UDM API With Class Diagram

Figure 6: Tool chain for generation of UDM API

We use UDM as the implementation environment for GRE. As shown in Figure 3, UDM is GRE's interface to the input graph, output graph and rewrite rules. For accessing the input and output graph the generic UDM API is used. The reason is that the input and output metamodels will vary from one transformation problem to another, thus the GRE cannot know about the input and output metamodels before hand. Hence, the GRE takes the input and output metamodels as input arguments and uses the generic API. On the other hand the Graph Rewrite (GR) metamodel is doesn't change from one problem to another that thus the GR specific UDM API is compiled into the GR. This obviously entails a performance penalty, and we plan to solve the problem in another way in the future.

The Graph Rewrite (GR) paradigm was created as a standardized format for the specification of transformation rules and sequencing. Hence, there can be different front-end graphical editors for the specification of these rules. The native format of the editor can then be mapped to the GR format to be fed as input to the GRE. Figure 7 shows the UML class diagram of the GR paradigm.

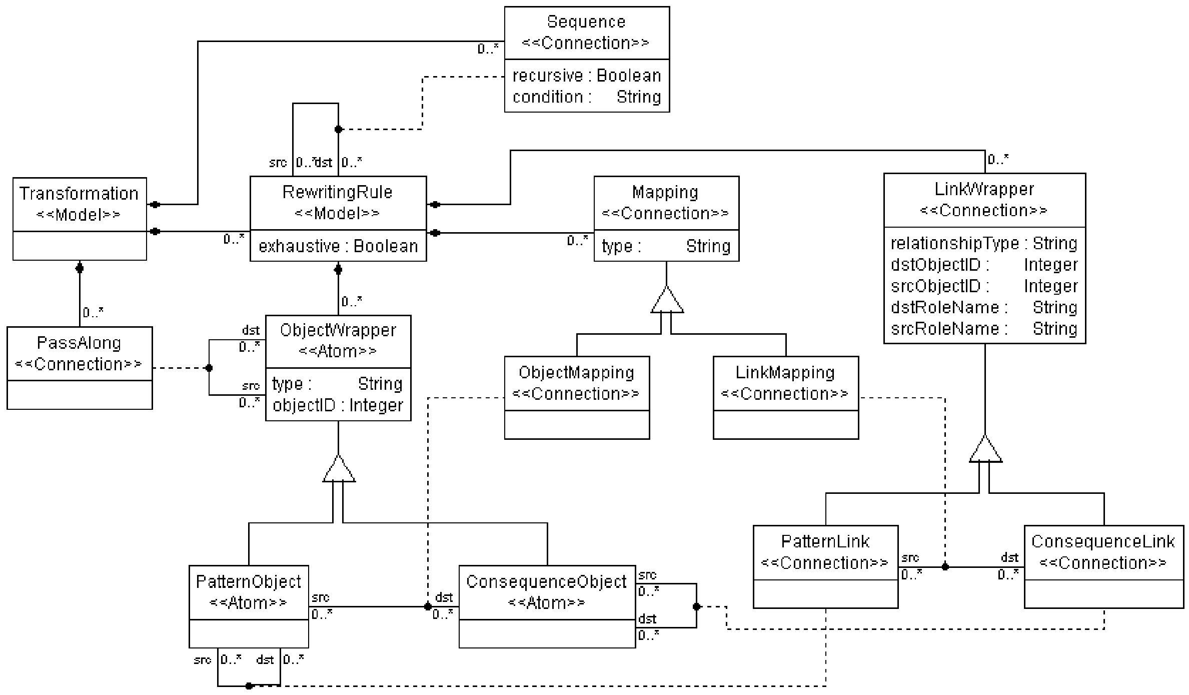
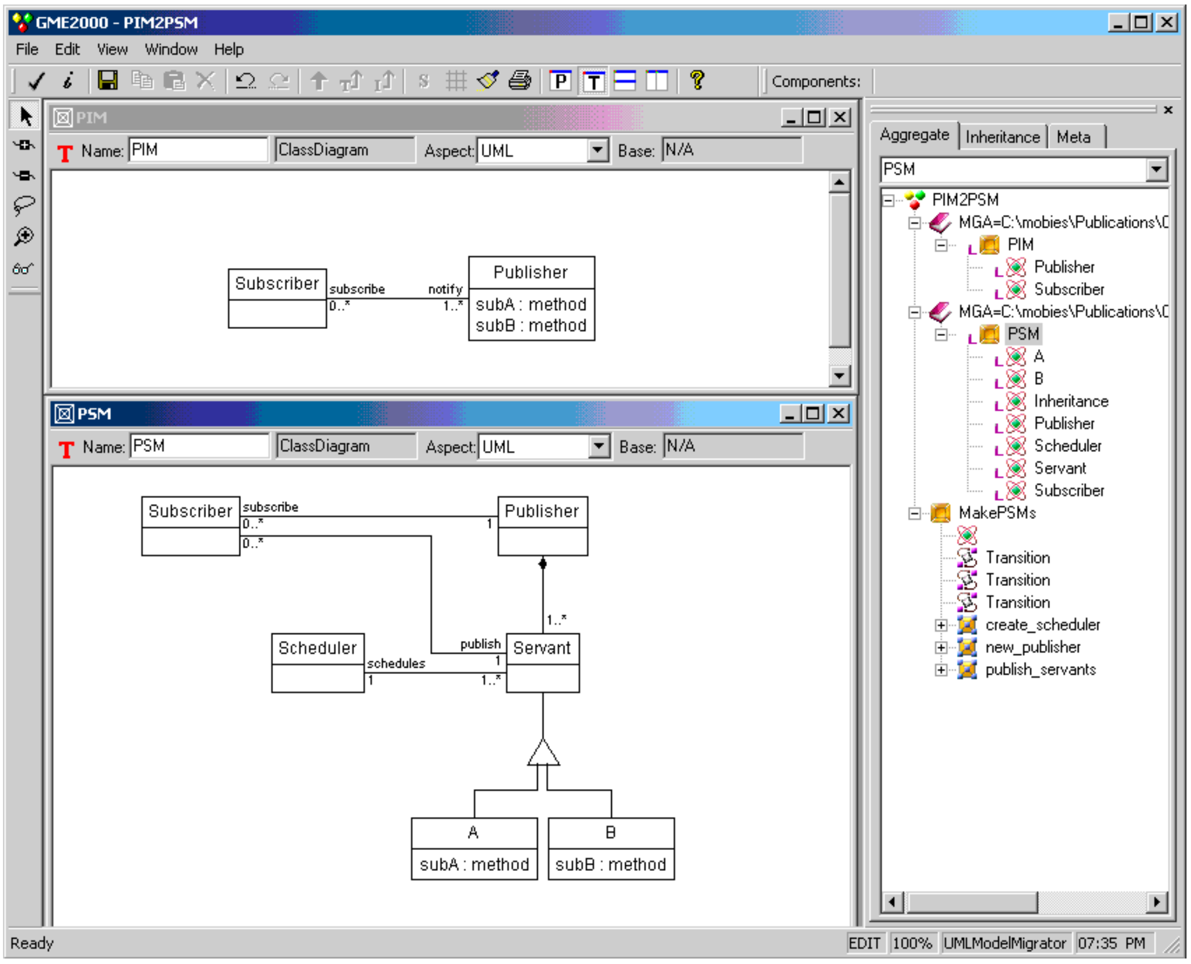


Figure 7: UML Class Diagram of the Graph Rewrite (GR) paradigm

In this paradigm, the *Transformation* contains *RewritingRule*, *Sequence* (which describes the rules ordering) and *PassAlong* (which describes the passing of input and output subgraph objects between rules). There are three main entities in each *RewritingRule*: the *PatternObject* and *PatternLink* specify the pattern to look for in the input graph, the *ConsequenceObject* and *ConsequenceLink* specify the relationship between output objects, and the *ObjectMapping* and *LinkMapping* describe the corresponding actions input to output graphs.


GME2000 - PIM2PSM

PIM — Name: PIM ClassDiagram Aspect: UML Base: N/A

PSM — Name: PSM ClassDiagram Aspect: UML Base: N/A

Ready EDIT 100% UMLModelMigrator 07:35 PM

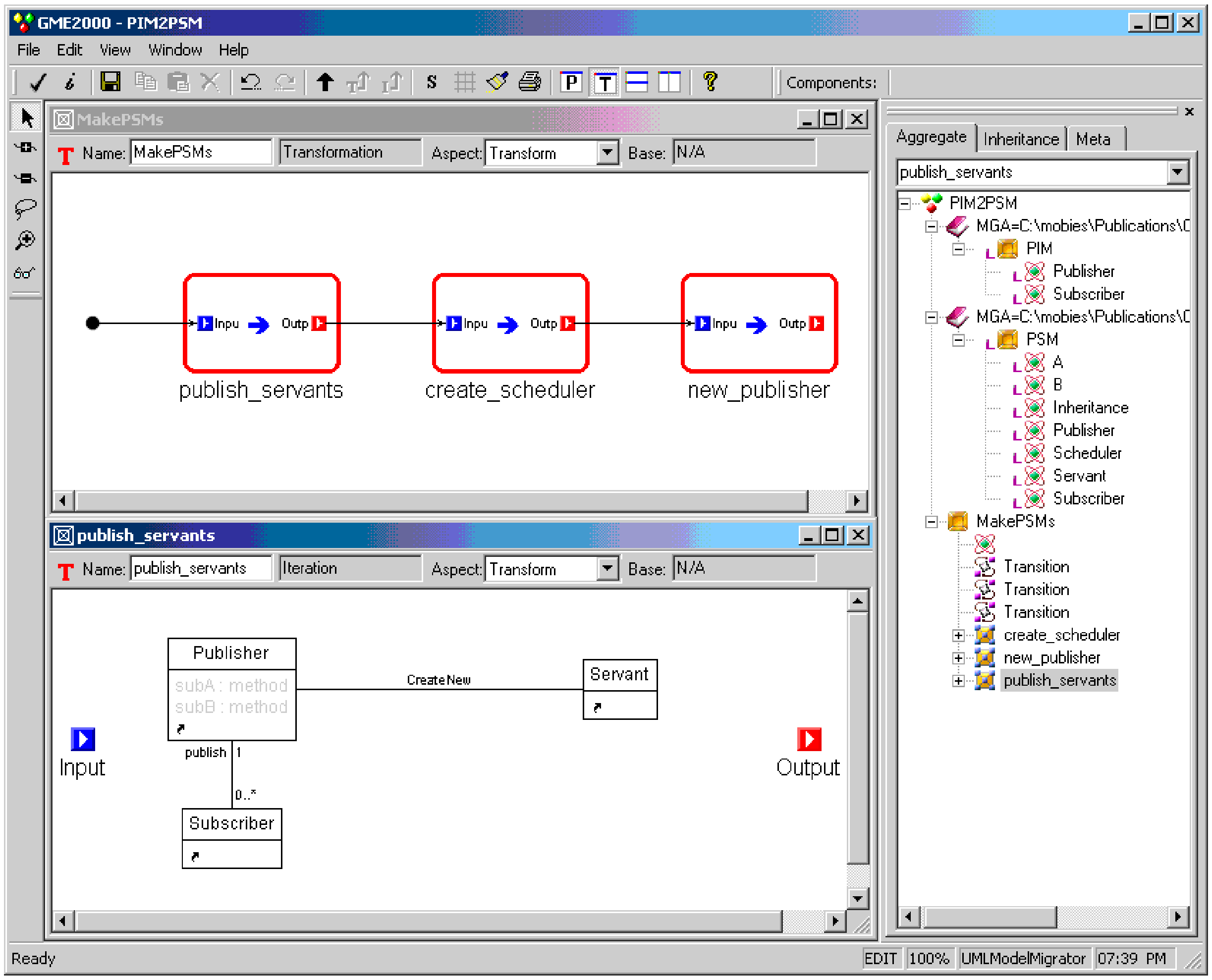
Figure 8: The Platform-Independent Model (top) and the Platform-Specific Model (bottom)

Example

Let us examine an example transformation. We will trace the transformation of domain models from a more abstract, generic model to a model with more specialized components. On the top, Figure 8 shows the platform-independent model (PIM) class diagram. This model shows that multiple Subscribers can subscribe to one of the multiple services provided by a Publisher.

Staring from the PIM, the transformer applies design patterns and some implementation details to build a more detailed platform-specific model (PSM). From Figure 8, only one instance of a Publisher is instantiated (Single design pattern). The publisher class then provides interface defining a different kind of operation for each kind of servant to be created (AbstractFactory design pattern). The publisher also hands over the subscriber's location so that a servant can notify its subscriber directly. Moreover, in this implementation, only one servant is assumed to be running at a time. Hence, for scheduling multiple servants the Scheduler class has been added to the PSM.

Transforming these models takes three steps. The first step is to transform all Publishers into Servants. After the appropriate publishers have been changed, a scheduler must be created. Finally, the new Publisher (which will be the only one in the new model) is created.


GME2000 - PIM2PSM

MakePSMs — Name: MakePSMs | Transformation | Aspect: Transform | Base: N/A

publish_servants — Name: publish_servants | Iteration | Aspect: Transform | Base: N/A

Aggregate: publish_servants

```
PIM2PSM
  MGA=C:\mobies\Publications\C
    PIM
      Publisher
      Subscriber
  MGA=C:\mobies\Publications\C
    PSM
      A
      B
      Inheritance
      Publisher
      Scheduler
      Servant
      Subscriber
  MakePSMs
    Transition
    Transition
    Transition
    create_scheduler
    new_publisher
    publish_servants
```

Ready | EDIT | 100% | UMLModelMigrator | 07:39 PM

Figure 9: The rule execution order (above) along with the rewriting definition to change all Publishers to Servants (below)

Figure 9 shows two levels of the transformation specification. The top window, “MakePSMs”, specifies the ordering of the execution of the transformation rules. The bottom window specifies how to change a Publisher into a Servant. The left side of the diagram (Publisher and Subscriber) is the prescribed pattern to match, and it is composed of items from the PIM. The right side of the diagram represents concepts in the PSM, and how these “target” objects are to be created from the PIM.

At this point, all the necessary information to begin the graph transformation process is present (see Figure 3). Inputting the proper input graphs (PIM domain models) and processing the encoded rewriting rules to the GRE will result in updated output graphs (models for the PSM domain).

Conclusions and future work

The Model Driven Architecture has come of age and is ready to become an integral part of the software development process. Models are not only a way of documenting initial ideas but are the formal representation of the actual system. The gap from models to executable code can be bridged using graph transformation techniques.

In this paper we have described a language to express transformation on graphs. We have also noted that the sequencing of rules and pivoting of patterns can greatly reduce the search time. Furthermore, we have shown the feasibility of the transformation approach by building a prototypical graph transformation system that is able to apply the transformation techniques and helps us test the validity of our approach. With the help of an example we have shown that the graph transformation techniques can be used to transform PIM's to PSM's.

There still is more to do in the GRE. To start with, the transformation language needs to be extended to support more sophisticated patterns and to allow the modification of the input graph using graph replacement. Secondly, the pattern matcher is very simplistic and a more sophisticated pattern-matching algorithm is needed to utilize the power of the sophisticated pattern specification language. Finally, the GRE is interpretive in the sense that the program interprets the rules at runtime and executes then. This needs to change such that the GRE can be generated, and it runs efficient code that will perform the transformation.

Acknowledgement

The DARPA/IXO MOBIES program and USAF/AFRL has supported under contract F30602-00-1-0580, in part, the activities described in this paper. The authors would like to thank one anonymous reviewer who pointed out a problem in the first version of the paper.

References

[1] The Model-Driven Architecture, http://www.omg.org/mda/ .
[2] Rational Rose tools, http://www.rational.com .
[3] The Generic Modeling Environment (GME 2000) documentation, available from http://www.isis.vanderbilt.edu/projects/gme/Doc.html
[4] Matlab tools, http://www.mathworks.com .
[5] Rozenberg,G. (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol.1-2. World Scientific, Singapore, 1997
[6] Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Transformations. Software - Practice and Experience 29(3): 197-217, 1999.
[7] U. Aßmann, ``How To Uniformly Specify Program Analysis and Transformation'', in: 6th Int. Conf. on Compiler Construction (CC '96), T. Gyimóthy (réd.), Lect. Notes in Comp. Sci., Springer-Verlag, Linköping, Sweden, 1996.
[8] A. Schürr. PROGRES for Beginners. RWTH Aachen, D-52056 Aachen, Germany.
[9] Taentzer, G.: AGG: A Tool Enviroment for Algebraic Graph Transformation, in Proc. of Applications of Graph Transformation with Industrial Relevance, Kerkrade, The Netherlands, LNCS,Springer, 2000.
[10] Maggiolo-Schettini, A., Peron, A.: Semantics of Full Statecharts Based on Graph Rewriting, Springer LNCS 776, 1994, pp. 265--279.
[11] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.: " Aspect-Oriented Programming," ECOOP'97, LNCS 1241, Springer. (1997)
[12] Simonyi, C.: "Intentional Programming: Asymptotic Fun?" Position Paper, SDP Workshop Vanderbilt University, December 13 - 14, 2001. http://isis.vanderbilt.edu/sdp
[13] 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
[14] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h.: UMLAUT: an extendible UML transformation framework, in Proc. Automated Software Engineering, ASE'99, Florida, October 1999.
[15] David H. Akehurst: Model translation: A uml-based specification technique and active implementation approach. PhD thesis, Computer Science at Kent University (UK), December 2000.
[16] Tony Clark, Andy Evans, Stuart Kent: Engineering Modelling Languages: A Precise Meta-Modelling Approach. FASE 2002: 159-173
[17] Tony Clark, Andy Evans, Stuart Kent: The Metamodelling Language Calculus: Foundation Semantics for UML. FASE 2001: 17-31.
[18] Lemesle, R. Transformation Rules Based on Meta-Modeling EDOC,'98, La Jolla, California, 3-5, November 1998, pp.113-122.
[19] Heckel, R. and Küster, 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.
[20] Karsai G.: Tool Support for Design Patterns, New Directions in Software Technology 4 Workshop, December, 2001. Available from http://www.isis.vanderbilt.edu .
[21] Bakay, A.: The UDM Framework, http://www.isis.vanderbilt.edu/Projects/mobies/