On the use of Graph Transformations in the Formal Specification of Computer-Based Systems

Gabor Karsai, Aditya Agrawal, Feng Shi, Jonathan Sprinkle
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235

Introduction

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

In the practice, there are very no full-blown 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, *there is a lack of capabilities from moving design information from one tool to the other*.

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 execution platform they have to generate code for. Furthermore, if one wants to verify, for instance the state machine models for the system, one has to rebuild those 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 in the engineering process. Obviously, these transformations can always be created 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 capturing the semantic mapping between the design information captured in the tools. We claim, that in addition to supporting the CBS design process through the “semantic bridges” between the 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. 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 to the semantics of the platform, one can formally define the semantics of the DSML. This idea has been presented in [7], in the context of Statecharts, but it can be easily generalized.

The paper introduces the version of the model-based engineering process for CBS: Model-Integrated Computing (MIC), and reviews various graph transformation techniques. Next it introduces a graph transformation language we have developed, and it shows how it can be used

for representing specific transformations. The paper concludes with a summary and suggestions for further research.

Backgrounds

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 defined, the specification of the domain is used to generate a Domain Specific Design Environment (DSDE). The DSDE can then be used to create domain specific designs/models; for example, a particular state machine is a domain specific design that conforms to the rules specified in the metamodel of the state machine domain. However, to do something useful with these models such as synthesize executable code, perform analysis or drive simulators, we have to convert the models into another format like executable code, input format of some analysis tool or configuration files for simulators. This mapping of models to a more useful form is called model interpretation and is performed by model interpreters. Model interpreters are programs that convert models of a given domain into another format. For mapping each domain to output format a unique model interpreter is required. The output can be considered as another model that conforms to a different metamodel and thus these model interpreters can be considered to be mappings between models[8].

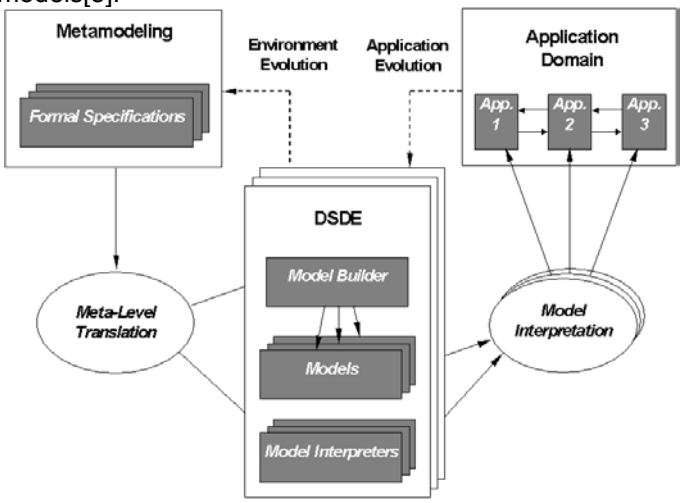

Figure 1: The MIC Development Cycle

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 environment of GME is based on UML class diagrams [10]. It is used to describe a domain specific modeling language and a corresponding environment by capturing the syntax, semantics and visualization rules of the target environment. A tool called the meta-interpreter interprets the metamodels and generates a configuration file for GME. This configuration file acts as a meta-program for the (generic) GME editing engine, so that it makes GME behave like a specialized modeling environment supporting the target domain. Thus the core of GME is used both as the metamodeling environment and the target environment.

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

The MIC approach described above is gaining a lot of attention recently with the advent of the Model Driven Architecture (MDA) by Object Management Group (OMG) [11]. The MDA is a particular application of the MIC approach where the domain language will be UML 2.0. However, a more general approach to the MDA problem will be to achieve domain specific model driven software development [12].

Graph grammars and graph rewriting [14][15] 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 [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) graph 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 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 [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 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 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 graph transformation language

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 we introduce some terminology that will be used 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 associate with a corresponding vertex in the output graph. In order to apply a rule we need to find the matching input subgraph in the input graph.It is well-known that subgraph isomorphism is NP-complete with order complexity $O(n_1^{n_2})$, where n_1 is 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. In order 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 consider and the exponent is reduced to only the number of unbound vertices in the pattern. Another issue is the sequencing of rule 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 2: Input and Output metamodels

Let us consider a simple example. The input and output metamodels are shown in Figure 2. 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 3. *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.

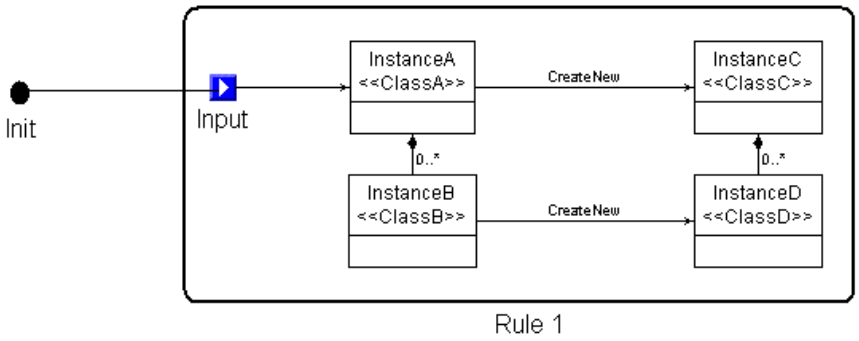

Figure 3: The transformation

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 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 4. The GRE 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. 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 GRE is fully meta-model-driven, and uses a reflective/persistent data structure package, called UDM [30].

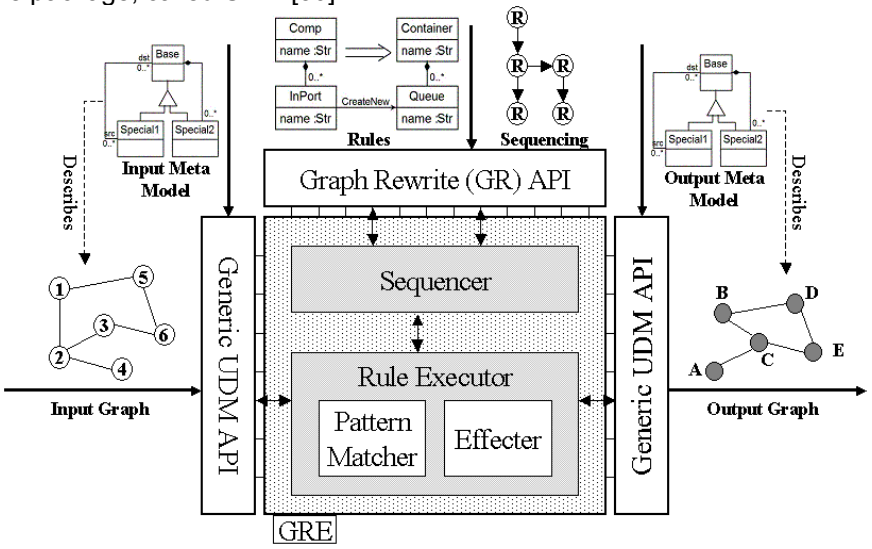

Figure 4: Run time architecture of the Graph Rewrite Engine

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 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 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 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, 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 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.

(a)

(b)

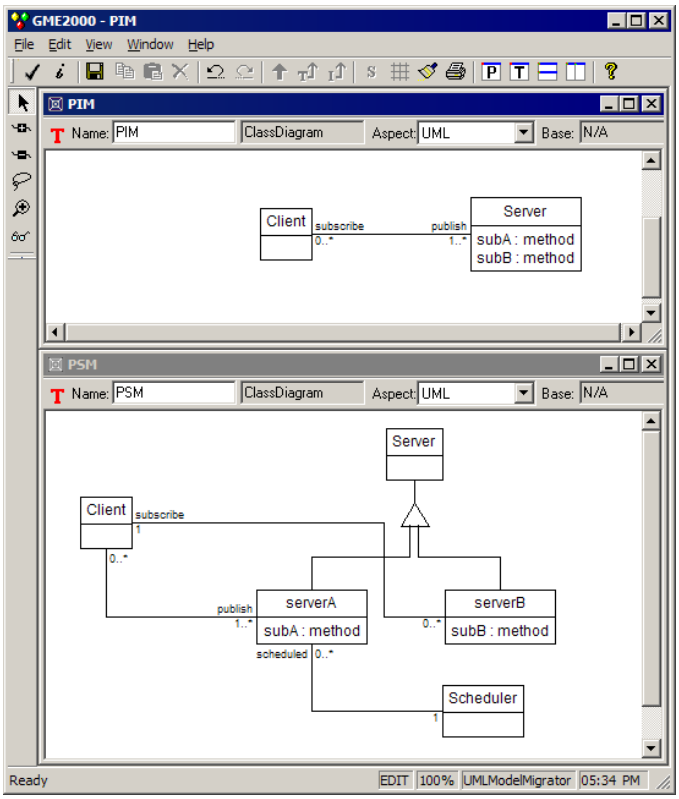

Figure 5: Two different design iterations of a publisher/subscriber CBS

Example

The creation of a CBS rarely occurs in one step. Rather, several design iterations usually take place, and different tools are used at different stages of the design. As is true with any system implementation, the functional interface of the system should not be dependent on its low-level implementation – thus design tools that can interoperate can increase the productivity of designers by not requiring them to perform hand entry of the system models for each tool, but rather using the same models for all design tools.

An example of the benefit of a working tool chain can be found in the following example of a CBS that implements the canonical client/server relationship. The exact implementation of the publish/subscribe relationships of the client and server 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 CBS designers, who are interested in

the performance of the system, as well as warehousing of data, and the scheduling algorithm of notification tasks.

Figure 5 shows two different metamodels of a client/server framework. The top is the interface metamodel – 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 metamodel 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.

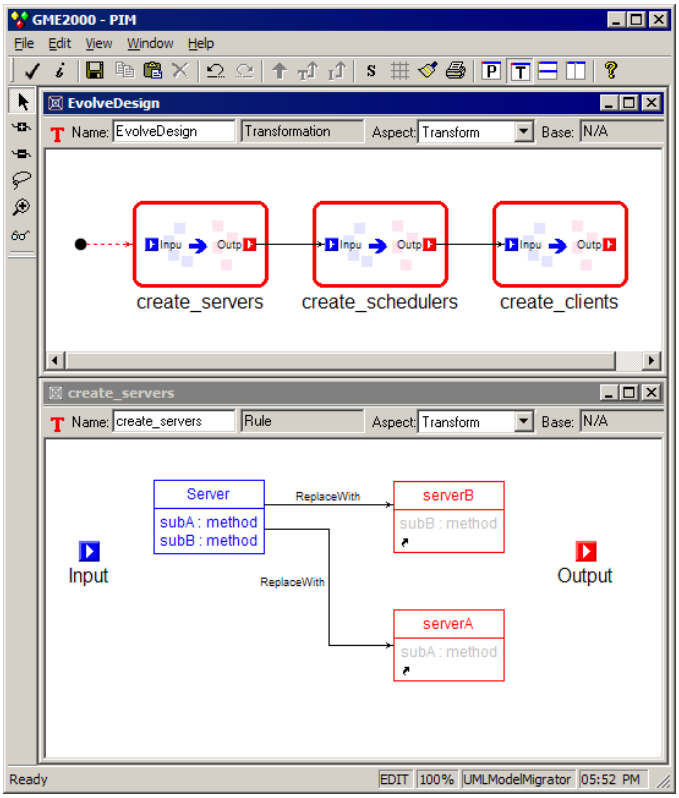

Figure 6: GRE formalization of the publisher/subscriber specialization

An observant designer of CBS will notice the similarities of the second design with the first. It is possible to use the GRE 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

The formal description of this algorithm is found in Figure 6. The sequence of the algorithm is shown in the top of the figure (the connected rewriting rules). Each of these rewriting rules contains a specification that formalizes exactly how models are to be transformed. The bottom portion of Figure 6 shows how the two servers are created. The semantics are that each publisher should be replaced with two servers – one to handle the subscriptions, and the other to handle the publications.

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. The GRE can be used to rapidly produce a translation that will enable multiple design evolutions throughout the formal specification process of a CBS.

Summary and conclusions

In this paper we have illustrated how meta-model-based graph transformations can be used in the construction of CBS. We claim that the design transformation process specified this way are completely 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. This type 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 GRE-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 algorithms, offer the opportunity for formally reasoning 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.

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.

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] K. L. McMillan, 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] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", Computer, Apr. 1997, pp. 110-112
[9] A. Ledeczi, et al., "Composing Domain-Specific Design Environments", Computer, Nov. 2001, pp. 44-51.
[10] J. Rumbaugh, I. Jacobson, and G. Booch, "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] Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Transformations. Software - Practice and Experience 29(3): 197-217, 1999.
[16] 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.
[17] A. Schürr. PROGRES for Beginners. RWTH Aachen, D-52056 Aachen, Germany.
[18] 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.
[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 Jézéquel, Alain Le Guennec, and François Pennaneac'h.: UMLAUT: an extendible UML transformation framework, in 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 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.
[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] The UDM tools, available from http://www.isis.vanderbilt.edu/Projects/MoBIES/ .