Graph Transformations in OMG's Model-Driven Architecture

Gabor Karsai[1] and Aditya Agrawal[1]

Institute for Software Integrated Systems (ISIS),
Vanderbilt University, Nashville, TN, USA
{gabor.karsai, aditya.agrawal}@vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. The Model-Driven Architecture (MDA) vision of the Object Management Group offers a unique opportunity for introducing Graph Transformation (GT) technology to the software industry. The paper proposes a domain-specific refinement of MDA, and describes a practical manifestation of MDA called Model-Integrated Computing (MIC). MIC extends MDA towards domain-specific modeling languages, and it is well supported by various generic tools that include model transformation tools based on graph transformations. The MIC tools are metaprogrammable, i.e. they can be tailored for specific domains using metamodels that include metamodels of transformations. The paper describes the development process and the supporting tools of MIC, and it raises a number of issues for future research on GT in MDA.

Graph grammars, graph transformations, Model-Integrated Computing, domain-specific modeling languages, model-driven architecture, formal specifications.

1 The MDA Vision

The Model-Driven Architecture initiative of OMG has put model-based approaches to software development into focus. The idea of creating models of software artifacts has been around for quite some time. However, this is the first time when mainstream software developers are willing to embrace the concept and demand tools that support this process. MDA is a "good thing" because it helps us develop software on a higher level of abstraction and - hopefully - will provide a "toolbox" that helps to keep the monster of complexity in check. This seems justified, as one can envision that multiple, yet interconnected models that represent requirements, design, etc. on different levels of abstraction, will offer a better way to work on complex software than today's UML models (used often only for documentation) and source code (spread across thousands of files).

Naturally, we need models to enact MDA; multiple, different kinds of models. Models capture our expectations ("requirements"), how the software is actually constructed ("design"), what kind of infrastructure the software will run on ("platform"), and other such details. There are - at least - two important observations that we can make about these models: (1) they are (or should be) linked to each other, (2) models can often be

computed from each other via model transformation processes. Among the advocates of MDA an agreement seems to be forming that model transformations play a crucial role and tool support is needed, but this need is often understood in the context of PIM-to-PSM mappings only.

MDA introduces the concepts of the Platform-Independent Models (PIM), and Platform-Specific Models (PSM). The rationale for making this distinction can be found in the requirement that a model-driven development process must be platform-independent, and the resulting software artifacts must exist in a form that allows their specializations (perhaps optimization) for different kinds of software platforms (e.g. CORBA CCM, .NET and EJB). Hence, PIM is a representation of a software design, which captures the essence and salient properties of the design, without platform-specific details, while the PSM is an extended, specialized representation that does include all the platform-specific details. The two models are related through some transformation process that can convert a PIM to its semantically equivalent PSM.

2 Refining the MDA Vision

While the MDA vision provides a roadmap for model-based software development, we can and must step beyond the "canonical MDA", where PSM-s (which are closed, practical implementations on a particular platform) are automatically created from PIM-s (which are closer to abstract design) via transformations (which capture platform-specific details). We argue that this "one-shot" view of transformations is very limited, and there is a much broader role for transformations. We envision a model-driven process where engineers develop multiple, interlinked models that capture the various aspects of the software being produced, and, at the same time, they also develop transformations that relate models of different kind to each other, and apply these transformations whenever necessary. Transformations become "bridges" that link models of different nature, and maintain consistency between models: when an "upstream" model is changed (by the designer or by a tool), the "downstream" model is automatically updated.

But the question arises: Where are the models coming from and what do they exactly capture? We argue that models should capture the various aspects of software design in a domain-specific manner. Domain-specificity in development is being widely recognized as a potential way of increasing productivity in software engineering. The idea is simple: instead of having a programmer constantly translate domain-specific knowledge into low-level code, first a "language" is built for the domain, next a translator is created that maps the language into other, possibly executable artifacts, and then the software products are built in the form of domain-specific models, which are then often transformed into code. The goal is to raise the level of abstraction to a level such that programming happens closer to the application domain and away from the implementation domain. Arguably, the model-driven process offers a natural habitat for realizing domain-specific software development, and we can call this integration of concepts from MDA and domain-specific development as Model-Integrated Computing [1], or Domain-Specific Model-Driven Architecture.

Definition: MIC is a domain-specific, model-driven approach to software development that uses models and transformations on models as first class artifacts, where models are sentences of domain-specific modeling languages (DSML-s). MIC captures the invariants of the domain, the fixed constructs of the DSML-s (i.e. the "grammar"), and the variabilities of the domain in the models (i.e. the "sentences").

MIC advocates development in a linguistic framework: the developer should define a DSML (including a transformations that interpret its "sentences"), and then use it to construct the final product: the software. This approach to software development brings forward some questions such as (1) how to define new languages, and (2) how define transformation tools for those languages.

3 Tools for Domain-Specific Model-Driven Architecture: MIC

First, we need a language to formally and precisely define DSML-s. Formally, a DSML is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S, and M_C) [18]:

$$L = \{C, A, S, M_S, M_C\} \tag{1}$$

The concrete syntax (C) defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The abstract syntax (A) defines the concepts, relationships, and integrity constraints available in the language. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. (It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently called "static semantics".) The semantic domain (S) is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The mapping M_C: $A \rightarrow C$ assigns concrete syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The semantic mapping M_S: $A \rightarrow S$ relates syntactic constructs to those of the semantic domain. The definition of the (DSM) language proceeds by constructing metamodels of the language (to cover A and C), and by constructing a metamodel for the semantics (to cover M_C and M_S).

3.1 Defining the syntax

A meta-programmable visual modeling environment, GME (see [2] for details)is available that provides a language called "MetaGME" for defining the abstract and concrete syntax of DSML-s. The abstract syntax is specified using a UML class diagram [4] editor that captures the abstract syntax in the form of a diagram, and static semantics in the form of OCL [11] expressions. The metamodel of MetaGME (i.e. the meta-metamodel) is MOF compliant [3]. Note that the UML class diagram is used to represent a "grammar" whose sentences are the "object graphs" that conform to it. Concrete syntax is captured in MetaGME using idioms: patterns of classes and stereotypes, which have a have a specific meaning for the GME visualization and editing engine. Currently the

GME editor supports a fixed set of visual modeling concepts. In the future, GME will be changed to enable adding modeling concepts and new visualization and manipulation techniques. Once the abstract and concrete syntax are defined, i.e. the metamodel of the language is built, a MetaGME “interpreter” translates this metamodel into a format that (the generic) GME uses to morph itself into a domain-specific GME that supports that (and only that) language which is defined by the metamodel. This GME instance strictly enforces the language “rules”: only models that comply with the abstract syntax and the static semantics can be built. Figure 1 shows an illustrative metamodel and a compliant model.

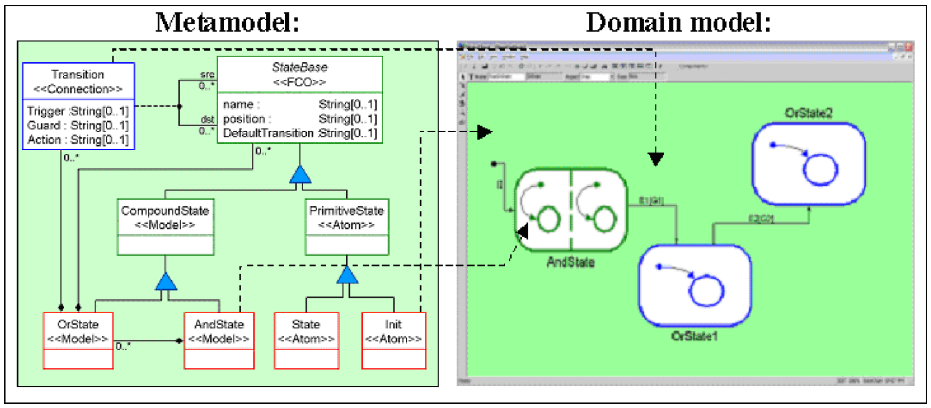

Fig. 1. Metamodel and model in MIC

3.2 Defining the semantics

For mapping the domain specific models into a semantic domain we have chosen a pragmatic approach: we assume that there is always a “target platform” whose semantics is well-known. This approach defining semantic: “semantics via transformations,” has been used in the past for the formal specification of semantics [8]. Note that the target platform also has an abstract syntax (with static semantics), and the transformation between the domain-specific models and target platform models establishes the semantics of the DSM-s in terms of the target models. One can observe that conceptually this is the same process employed in MDA’s PIM to PSM transformation: the transformation provides semantics to platform-independent models via their mapping to platform-specific models. In MIC, just like in MDA, transformations play a crucial role: they specify the (dynamic) semantics of domain-specific models.

On a more general note, one can observe that in a model-based development process transformations appear in many, different situations. A few representative examples are as follows:

– Refining the design to implementation; this is the basic case in the PIM/PSM mapping.
– Pattern application; expressing design patterns as locally applied transformations on the software models [10, 13].
– Aspect weaving; the integration of aspect code into functional code is a transformation on the design [17].
– Analysis and verification; analysis algorithms can be expressed as transformations on the design [20].

One can conclude that transformations can and will play an essential role, in general, in model-based development, thus there is a need for highly reusable model transformation tools. These tools must be generic, in our terms: meta-programmable; i.e. their function should be determined by a "meta-program", which defines how models are transformed.

There exist well-known technologies today that seem to satisfy these requirements, and do not require sophisticated metamodeling: for instance XML and XSLT [12]. XML provides a structured way to organize data (essentially as tagged/typed data elements, organized in a hierarchy and untyped references that cut across the hierarchy), while XSLT provides a language to define transformations on XML trees. However, XSLT is not adequate for implementing sophisticated model transformations: (1) it lacks a type system, (2) it does not support reasoning about transformations, (3) and its performance is often not sufficient for practical development [19].

3.3 Defining Semantics via transformations

We have created a model transformation system called "Graph Rewriting and Transformation" (GReAT). GReAT consists of a model transformation language called UML Model transformer (UMT), a virtual machine called Graph Rewrite Engine (GRE), a debugger for UMT called Graph Rewriting Debugger (GRD) and a Code Generator (CG) that converts the transformation models into efficient, executable code [6, 7].

In UMT transformations are specified on metamodel elements. This helps to strongly type the transformations and ensures the syntactic correctness of the result of the transformations, within the specification. The transformation rules consist of a graph pattern (which is matched against an input graph), a guard condition (a precondition, which is evaluated over the matched subgraph), a consequence pattern (which expresses the creation and deletion of target graph objects), and a set of attribute mapping actions that are used to modify the attributes of input or target objects). These transformations specify the mapping of input models (i.e. the input "graph") to target models (i.e. the "output graph"). For efficiency reasons rewriting rules accept "pivot" points: initial bindings for pattern variables. This reduces the search in the pattern matching process (effectively reducing it to matching in a rooted tree). One can also explicitly sequence the execution of the rules, and sequential, parallel, and conditional composition of the rules is also available.

GRE works as an interpreter: it executes transformation programs (which are expressed in the form of transformation metamodels) on domain-specific models to generate target models. GRE is slow compared to hand-written code but is still useful while

creating and modifying the transformation. The GRD provides debugging capabilities on top of GRE such as setting break points, single step, step into, step out and step over functions. A visual front end to the debugger is also available.

After the transformations have been created, debugged and tested the Code Generator (CG) can be used to generate efficient code that executes the transformations. The generated code improves the performance of the transformations by at least two orders of magnitude, early experiments show. An in-depth coverage of the GReAT system is provided in the next section.

4 The GReAT Model Transformation System

This section provides a description of the details of the GReAT tool. First, the transformation language is described, followed by a description of the GRE.

4.1 The UML Model Transformer (UMT) Language

UMT consists of three sub languages: (1) the pattern specification language, (2) the graph rewriting language, and (3) the control flow language.

4.2 The Pattern Specification Language

At the heart of a graph transformation language is the pattern specification language and the related pattern matching algorithms. The pattern specifications found in graph grammars and transformation languages [6, 7, 22, 23] do not scale well, as the entire pattern to be matched has to be enumerated. The pattern matching language provides additional constructs for the concise yet precise description of patterns. String matching will be used to illustrate representative analogies.

Patterns in most graph transformation languages have a one-to-one correspondence with the host graph. Consider an example from the domain of textual languages where a string to match starts with an 's' and is followed by 5 'o'-s. To specify such a pattern, we could enumerate the 'o'-s and write "sooooo". Since this is not a scalable solution, a representation format is required to specify such strings in a concise and scalable manner. One can use regular expressions: for strings we could write it as "s5o" and use the semantic meaning that o needs to be repeated 5 times. The same argument holds for graphs, and a similar technique can be used. Cardinality can be specified for each pattern vertex with the semantic meaning that a pattern vertex must match n host graph vertices, where n is its cardinality. However, it is not obvious how the notion of cardinality truly extends to graphs. In text we have the advantage of a strict ordering from left to right, whereas graphs do not possess this property.

In figure 2(a) we see a pattern having three vertices. One possible meaning could be tree semantics, i.e., if a pattern vertex pv1 with cardinality c1 is adjacent to pattern vertex pv2 with cardinality c2, then the semantics is that each vertex bound to v1 will be adjacent to c2 vertices bound to v2. These semantics when applied to the pattern gives figure 2(b). The tree semantics is weak in the sense that it will yield different results for

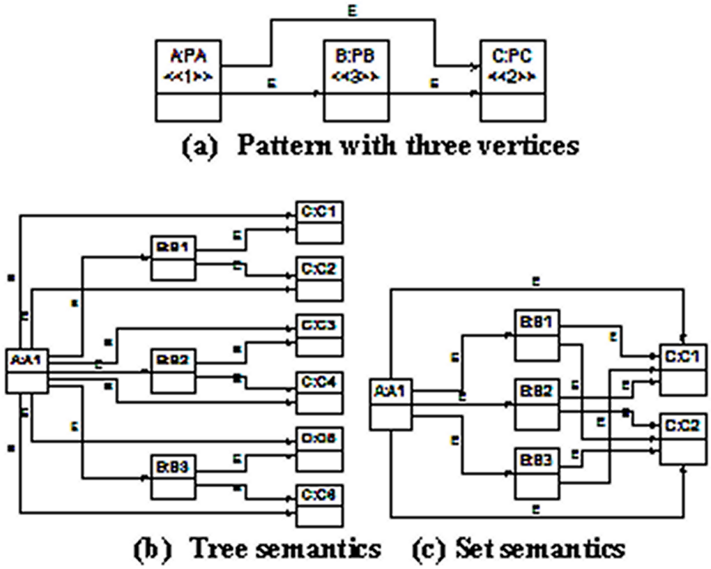

Fig. 2. Pattern with different semantic meanings

different traversals of the pattern vertices and edges and hence, it is not suitable for our purpose.

Another possible unambiguous meaning could use set semantics: consider each pattern vertex pv to match a set of host vertices equal to the cardinality of the vertex. Then an edge between two pattern vertices pv1 and pv2 implies that in a match each v1, v2 pair should be adjacent, where v1 is bound to pv1 and v2 is bound to pv2. This semantic when applied to the pattern in figure 2(a) gives the graph in figure 2(c). The set semantics will always return a match of the structure shown in figure 2(c), and it does not depend upon factors such as the starting point of the search and how the search is conducted.

Due to these reasons, we use set semantics in GReAT and have developed pattern-matching algorithms for both single cardinality and fixed cardinality of vertices.

4.3 Graph Transformation Language

Pattern specification is just one important part of a graph transformation language. Other important concerns include the specification of structural constraints in graphs and ensuring that these are maintained throughout the transformations [6]. These problems have been addressed in a number of other approaches, such as [22, 23].

In model-transformers, structural integrity is a primary concern. Model-to-model transformations usually transform models from one domain to models that conform to another domain making the problem two-fold. The first problem is to specify and maintain two different models conforming to two different metamodels, simultaneously. An

even more relevant problem to address involves maintaining associations between the two models. For example, it is important to maintain some sort of references, links, and other intermediate values required to correlate graph objects across the two domains.

Our solution to these problems is to use the source and destination metamodels to explicitly specify the temporary vertices and edges. This approach creates a unified metamodel along with the temporary objects. The advantage of this approach is that we can then treat the source model, destination model, and temporary objects as a single graph. Standard graph grammar and transformation techniques can then be used to specify the transformation.

The rewriting language uses the pattern language described above. Each pattern object's type conforms to the unified metamodel and only transformations that do not violate the metamodel are allowed. At the end of the transformation, the temporary objects are removed and the two models conform exactly to their respective metamodels. Our transformation language is inspired by many previous efforts, such as [6, 7, 22, 23]

The graph transformation language of GReAT defines a production (also referred to as rule) as the basic transformation entity. A production contains a pattern graph (discussed above) that consists of pattern vertices and edges. Each object in the pattern graph conforms to a type from the metamodel. Each object in the production has another attribute that specifies the role it plays in the transformation. A pattern can play the following three, different roles:

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 marked either bind or delete. If the pattern matcher is successful in finding matches for the pattern, then for each match the pattern objects marked delete are deleted from the match and objects marked new are created.

Sometimes the pattern alone is not enough to specify the exact graph parts to match and we need other, non-structural constraints on the pattern. An example for such a constraint is: "the value of an (integer) attribute of a particular vertex should be within some limits." These constraints or pre-conditions are captured in a guard and are written using the Object Constraint Language (OCL) [11]. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing object. Attribute Mapping is another ingredient of the production: it describes how the attributes of the "new" objects should be computed from the attributes of the objects participating in the match. Attribute mapping is applied to each match after the structural changes are completed.

A production is thus a 4-tuple, containing: a pattern graph, mapping function that maps pattern objects to actions, a guard expression (in OCL), and the attribute mapping.

4.4 Controlled Graph Rewriting and Transformation

To increase the efficiency and effectiveness of a graph transformation tool, it is essential to have efficient implementations for the productions. Since the pattern matcher is the

most time consuming operation, it needs to be optimized. One solution is to reduce the search space (and thus time) by starting the pattern-matching algorithm with an initial context. An initial context is a partial binding of pattern objects to input (host) graph objects. This approach significantly reduces the time complexity of the search by limiting the search space. In order to provide initial bindings, the production definition is expanded to include the concept of ports. Ports are elements of a production that are visible at a higher-level and can then be used to supply initial bindings. Ports are also used to retrieve result objects from the production (and pass them along to a downstream production).

An additional concern is the application order of the productions. In graph grammars there is no ordering imposed on productions. There, if the pattern to be matched exists in the host graph and if the pre-condition is met then the production will be executed. Although this technique is useful for generating and matching languages, it is not efficient for model-to-model transformations that are algorithmic in nature and require strict control over the execution sequence. Moreover, a well-defined execution sequence can be used to make the implementation 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. The control flow language of GReAT 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.

4.5 MIC Tools

Figure 3 shows the MIC tool suite included in GReAT, and how the tools rely on metamodels. In order to set up a specific MIC process one has to create metamodels for the (input) domain, the (output) target, and the transformations. One can then use the meta-level tools (such as the MetaGME interpreter and the Code Generator) to build a domain specific model editor and a model transformation tool. The model editor is then used to create and modify domain models, while the transformation tool is used to convert the models into target models.

We believe a crucial ingredient in the above scheme is the meta-programmable transformation tool: GRE that executes a transformation metamodel (as a "program") and facilitates the model transformation. Using the concepts and techniques of graph transformations allows not only the formal specification of transformations, but, arguably, reasoning about the properties of the transformations as well.

5 Example

The tools described above can also be used to implement PIM to PSM transformations of MDA as illustrated through the following example. The example shows the transfor-

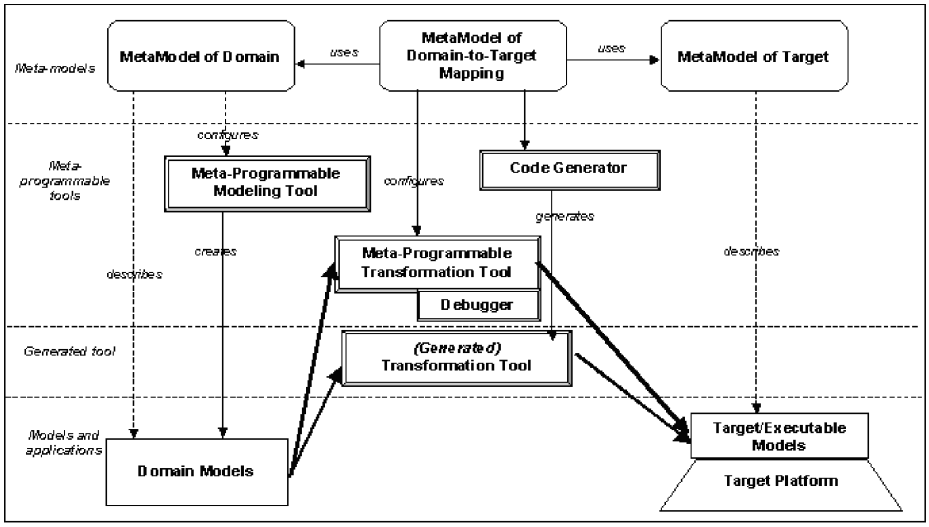

Fig. 3. Tools of Domain-Specific MDA

mation of software designs from a more abstract, generic model to a model with more specialized components. Figure 4(a) shows the platform-independent model: a UML class diagram. The model describes the entities *Publisher* and *Subscriber*, and the relationship between them. In this case the relationship specifies that multiple *Subscribers* can subscribe to one of the multiple services provided by a *Publisher*.

Starting from the PIM, the transformer applies design patterns and adds further implementation details to build a more detailed, platform-specific model (PSM). Figure 4(b) shows the refined model where there is only one *Publisher* (indicated by the cardinality on the *subscribes* association). This class could be implemented using the *Singleton* design pattern. The *Publisher* class creates a new, specific *Servant* for each Subscriber (the Servants could be created using the "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 created, a *Scheduler* must be created. Finally, the new *Publisher* (which will be the only one in the new model) is created.

Figure 5 shows two levels of the transformation specification. Figure 5(a), "MakePSMs" specifies the order of execution of the transformation rules. Figure 5(b), the specification for the transformation that converts a publisher into a servant is shown. On the bottom left side of figure 5(b), the *Publisher* and *Subscriber* form the (PIM) pattern

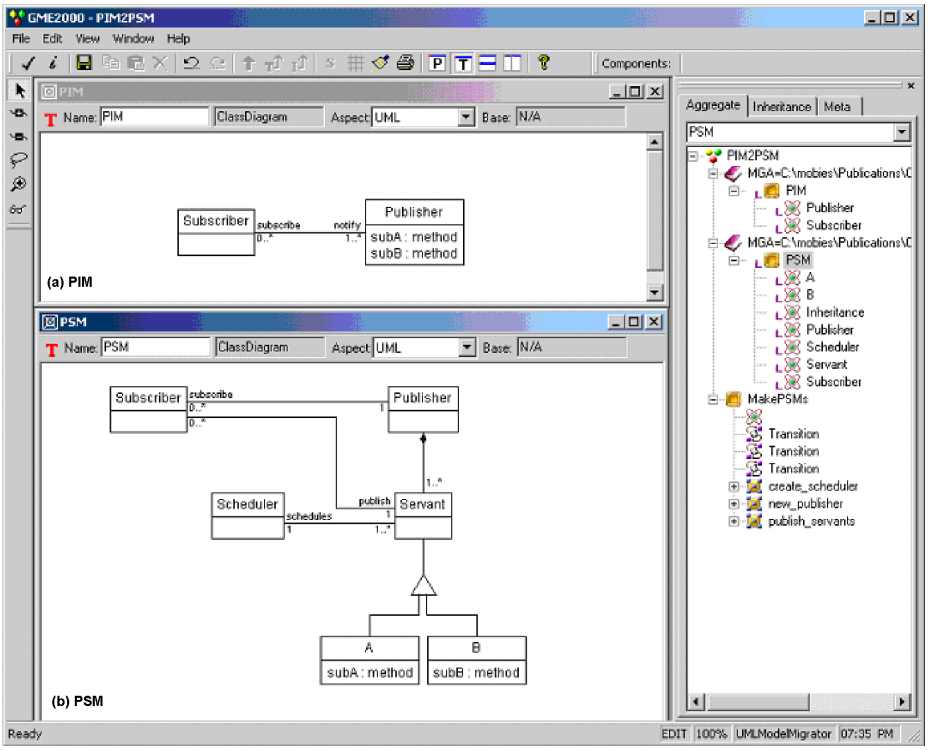

Fig. 4. Platform-Independent Model and Platform-Specific Model

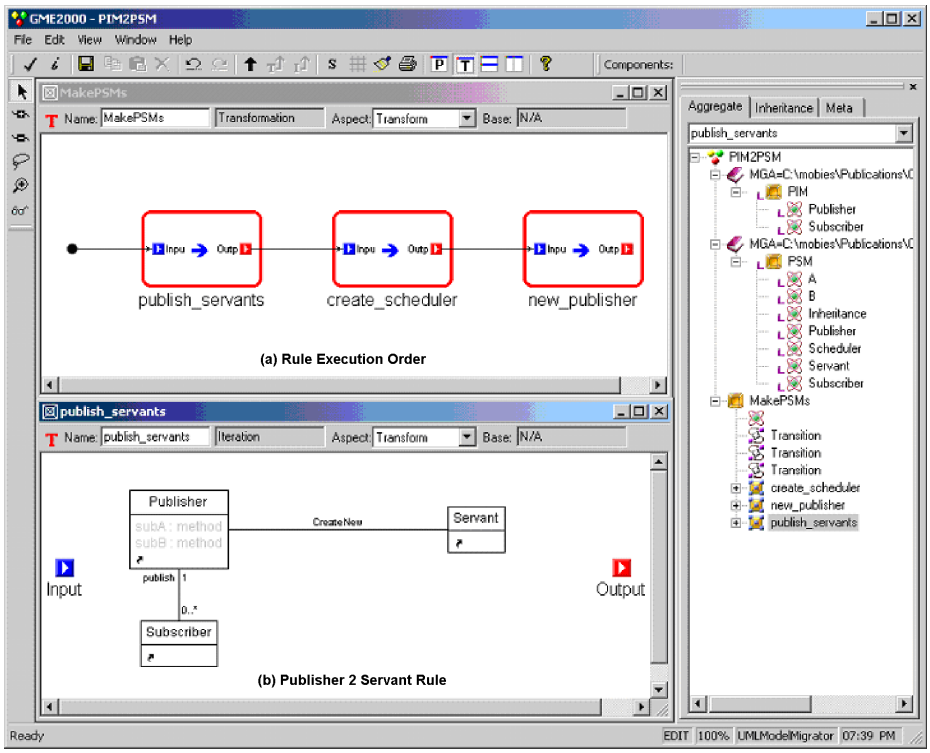

Fig. 5. Transformation rules to convert publisher subscriber PIM to PSM

to be matched, and on the right side the *Servant* (PSM) denotes the new object to be created.

6 Extending MIC towards a multi-model process

In the previous discussions on MIC we have focused on a single DSML and a single target; similar to the simple PIM/PSM mapping. In a large-scale application of MIC, however, a multitude of metamodels and transformations are needed. As discussed in the introduction, models can capture requirements, designs, platforms (and many other subjects), as well as the transformations between them. We envision that the next generations of software development tools are going to support this multi-model development.

It is interesting to draw the parallel here with the design and development of very large-scale integrated circuits. VLSI circuits are design using a number of languages (VHDL being only one of them), and using transformation tools (datapath generators, etc.) that "link" the various design artifacts [9]. It usually takes a number of steps to go from a high-level representation of a design to the level of masks, and consistency and correctness must be maintained across the levels. We believe that with consistent and recursive application of MDA through the use of transformations, the software engineering processes will be able to achieve the kind of reliability VLSI design processes have achieved.

We envision that the developers who apply MIC develop a number of domain-specific modeling languages. Modeling languages are for capturing requirements, designs, but also platforms. It is conceivable that the engineer wants to maintain a hierarchy of design models that represent the system on different levels of abstractions. Lower-level design models can be computed from higher-level ones through a transformational process. Maintaining consistency across models is of utmost importance, but if a formal specification of the transformation is available then presumably this can be automated. Note that model transformations are also models. Thus they can be computed from higher-level models by yet another transformation, and, in fact, transformation specifications can be derived also through a transformation process.

Naturally, the toolset introduced above need to be extended to allow this multi-model MIC. Presumably models should be kept in a model database (or "warehouse"), with which the developers interact. Transformations can be applied automatically, or by the user. Transformation results may be retained, and/or directly manipulated by the developer. We believe that an environment that supports this process should allow multiple formalisms for visualizing and manipulating artifacts ("models"), and transformation objects should ensure the coherency across the objects.

7 Relationship to OMG's QVT

On can argue that the approach described above *is* MDA, instead of being an extension of MDA. Indeed, the approach described above has all the ingredients of the MDA vision as described in [3], and recent papers pointed towards using UML as a mechanism for defining a family of languages [21]. We fully agree with this notion, however we can

envision applications where the full flexibility and power of MIC is not worth the cost. For example, “throw-away”, “use-it-once” applications may not require all the features of MIC.

On the other hand, the MIC tools discussed above provide a way of implementing the MDA concepts, including QVT, although they do not support everything, at least not immediately. The reason is that we wanted to focus on the DSML-s and drive the entire development process using domain-specific models, instead of the modeling language(s) defined by UML. We were also concentrating on building configurable, meta-programmable tools that support arbitrary modeling approaches.

8 Challenges and opportunities for Graph Transformations in the MDA

Graph Transformation (GT) is powerful technology that may lead to a solid, formal foundation for MDA. It offers a number of challenges and opportunities for further research, as listed below.

1. GT as the bridge between the programmer’s intentions and their implementation. The vision of Intentional Programming (from Charles Simonyi) is that software should be built in the form of “intentions”: high-level abstractions specify what the software needs to do, and explicit transformations that convert these intentions into executable code. We believe GT is the perfect vehicle to realize this vision.
2. “Provably correct” software via provably correct GT-s. In many critical applications software reliability is of utmost importance, yet reliability is assured using extensive testing and, to a very limited extent, through formal verification. Formal specification of the transformations can lead to formal verification of the software artifacts and may be able to provide some confidence in the generated software.
3. Non-hierarchical (de)composition supported by GT-s. Requirement analysis often leads to a non-hierarchical decomposition of the problem, and implementation could also lead to non-hierarchical composition of the system. Perhaps the best example for the latter is aspect-oriented programming. We argue that GT-s offer a uniform framework for describing, representing, implementing and analyzing these orthogonal (de)compositions.
4. Assurance of para-functional properties of the final SW. Especially in the field of embedded systems, it is often necessary to calculate para-functional properties (like schedulability, timeliness, performance, lack of deadlocks, etc.) of the system at design time. We conjecture that some of these properties can be formally defined and calculated using GT techniques.
5. Efficient implementations of GT. The usability of GT tools will determine success of the GT technology. Efficient implementation algorithms need to be developed such that the performance of GT based transformations is at acceptable levels and is comparable to that of the equivalent, hand-written code.
6. GT tools and support as an integral part of the software development process. A large number of IDE-s are used in software development today, some with extremely well-defined processes. These IDE-s (typically) do not handle GT-s (yet).

Again, the industrial success of GT-s will depend on how well GT tools are integrated with existing tools and processes.

7. Design tool integration via GT. Many development processes require a large number of (possibly heterogeneous) tools. This implies a need for generic tool integration solutions [16]. We claim that GT-s offer an opportunity for implementing these solutions and provide a highly effective technology for the rapid creation of integrated tools.
8. Teaching software engineers about GT as a programming paradigm (design-time and run-time). Currently, GT technology is not a mainstream software engineering technology. One potential cause of this is the lack of trained software engineers who *can* use GT as an engineering tool. There is a need for courses and tutorials on this topic, and the training should cover the use of GT-s for both design-time (i.e. transformation on the software design artifacts), and run-time (i.e. GT on domain-specific data structures, to implement some application functionality).

9 Summary

We claim that a sophisticated model-driven software development process requires multiple, domain-specific models. Building transformation tools that link these models, by mapping them into each other (including mapping into executable) form a crucial tool component in the process, and graph transformations offer a fundamental technology for these transformations. We here briefly introduced a domain-specific model-driven process and its supporting tools, some of them based on graph transformations, and summarized the major concepts behind it.

The described toolset exists in a prototype implementation today and has been used in small-scale examples. However, it needs to be extended towards a multi-model process, where a wide variety of models (and modeling languages) can be used. This extension is the subject of ongoing research.

10 Acknowledgements

The NSF ITR on “Foundations of Hybrid and Embedded Software Systems” has supported, in part, the activities described in this paper. The effort was also sponsored by DARPA, AFRL, USAF, under agreement number F30602-00-1-0580.The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon. The views and conclusions contained therein are those of authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of the DARPA, the AFRL or the US Government. Tihamer Levendovszky and Jonathan Sprinkle have contributed to the discussions and work that lead to GReAT, and Feng Shi has written the first implementation of GReAT-E. A shortened, preliminary version of this paper has appeared in the WISME workshop at the UML 2003 conference.

References

1. J. Sztipanovits, and G. Karsai, “Model-Integrated Computing”, IEEE Computer, Apr. 1997, pp. 110-112
2. A. Ledeczi, et al., “Composing Domain-Specific Design Environments”, IEEE Computer, Nov. 2001, pp. 44-51.
3. “The Model-Driven Architecture”, http://www.omg.org/mda/, OMG, Needham, MA, 2002.
4. J. Rumbaugh, I. Jacobson, and G. Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1998.
5. A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, G. Karsai, “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.
6. Rozenberg G. (ed.), “Handbook on Graph Grammars and Computing by Graph Transformation: Foundations”; Vol.1-2. World Scientific, Singapore, 1997.
7. Blostein D., Schürr A., “Computing with Graphs and Graph Transformations”, Software - Practice and Experience 29(3): 197-217, 1999.
8. Maggiolo-Schettini A., Peron A., “Semantics of Full Statecharts Based on Graph Rewriting”, Springer LNCS 776, 1994, pp. 265–279.
9. 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.
10. A. Radermacher, “Support for Design Patterns through Graph Transformation Tools”, Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, September 1999.
11. Object Management Group, “Object Constraint Language Specification”, OMG Document formal/01-9-77. September 2001.
12. XSL Transformations, www.w3.org/TR/xslt.
13. Karsai G., “Tool Support for Design Patterns”, NDIST 4 Workshop, December, 2001. (Available from: www.isis.vanderbilt.edu).
14. 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.
15. J. Gray, G. Karsai, “An Examination of DSLs for Concisely Representing Model Traversals and Transformations”, 36th Annual Hawaii International Conference on System Sciences (HICSS’03) - Track 9, p. 325a, January 06 - 09, 2003.
16. Karsai G., Lang A., Neema S., “Tool Integration Patterns, Workshop on Tool Integration in System Developement”, ESEC/FSE , pp 33-38., Helsinki, Finland, September, 2003.
17. Uwe Assmann and A. Ludwig, “Aspect Weaving by Graph Rewriting”, In U. Eisenecker and K. Czarnecki (ed.), Generative Component-based Software Engineering. Springer, 2000.
18. T. Clark, A. Evans, S. Kent, P. Sammut, “The MMF Approach to Engineering Object-Oriented Design Languages”, Workshop on Language Descriptions, Tools and Applications (LDTA2001), April, 2001
19. Karsai G., “Why is XML not suitable for Semantic Translation”, Research Note, ISIS, Nashville, TN, April, 2000. (Available from: www.isis.vanderbilt.edu)
20. 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.
21. Keith Duddy, “UML2 must enable a family of languages”, CACM 45(11), p, 73-75, 2002.
22. H. Fahmy, B. Blostein, “A Graph Grammar for Recognition of Music Notation”, Machine Vision and Applications, Vol. 6, No. 2 (1993), 83-99.

23. G. Engels, H. Ehrig, G. Rozenberg (eds.), “Special Issue on Graph Transformation Systems”, Fundamenta Informaticae, Vol. 26, No. 3/4 (1996), No. 1/2, IOS Press (1995).