A Metamodel-Driven MDA Process and its Tools

Gabor Karsai, Aditya Agarwal, and Akos Ledeczi
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

A domain-specific refinement of MDA, called DS-MDA is introduced, and a practical manifestation of it called MIC (for Model-Integrated Computing) is described. MIC extends MDA in the direction of domain-specific modeling languages. The MIC tools are metaprogrammable, i.e. are tailored for specific domains using meta-models. Meta-models capture the domain's and the target platform's general properties, as well as the transformation between the two. The paper introduces the tools and process that supports single domains, and proposes an extension towards multi-model processes.

Broadening MDA

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, but perhaps this is the first time when mainstream software developers 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 sometimes only for documentation) and source code (spread across thousands of files).

We need models to "do" MDA, of course, and multiple, different kinds of models. Models can capture our expectations ("requirements"), how the software is actually constructed ("design"), what kind of infrastructure the software will run on ("platform"), and many other 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) sometimes models can be computed from each other via a model transformation processes. Among the advocates of MDA an agreement seems to be forming that says 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.

We believe that we can 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 are – presumably – platform-specific). 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, but, 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[1].

But where are models coming from and what do they exactly capture? We argue that models should capture domain-specific knowledge. Domain-specificity in development is being widely recognized as a potential way of increasing productivity in software engineering. The idea

[1] Some UML tools provide this service today: code is derived from the models, and model changes are propagated to the generated source code.

is simple: instead of having a programmer to constantly translate domain-specific knowledge into low-level code, let us build first a "language"[2] for the domain, next build a translator that maps the language into executable artifacts and then build create the software by building domain-specific artifacts (and translate them into executable form). That is, raise the level of abstraction where the programming happens closer to the domain, away from the implementation. 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, and where models are sentences of domain-specific modeling languages (DSML-s). MIC captures the invariants of the domain in 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 transformation engine that interprets its "sentences"), and then use it to construct the end-product. Therefore the questions related to MIC are focusing on: (1) how to define new languages, and (2) how define transformation tools for those languages. In the next section we briefly introduce a suite of tools that help solve this problem.

Tools for Domain-Specific Model-Driven Architecture: MIC

First, we need tools 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)[13]:

$$L = < C, A, S, M_S, M_C>$$

The *C concrete syntax* defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The *A abstract syntax* 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 *S semantic domain* is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The M_C: $A \rightarrow C$ mapping assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The M_S: $A \rightarrow S$ semantic mapping relates syntactic concepts 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).

We have built a meta-programmable visual modeling environment, GME (see [2] for details), to solve the problem of defining the abstract syntax and the concrete syntax. For the abstract syntax, GME provides a UML class diagram [4] editor that allows capture of the abstract syntax in the form of a diagram, and of the static semantics in the form of OCL [10] expressions. Note that the UML class diagram acts like a "grammar" whose sentences are the "object graphs" that comply with it. For concrete syntax, GME uses idioms (patterns of classes) and stereotypes: specific idioms and stereotypes have specific meaning for the GME visualization and editing engine. As GME has a finite set of visualization tools, this will probably need to be extended in the future such that arbitrary visualization (and manipulation) techniques could be coupled to entities in the abstract syntax. Once the abstract and concrete syntax are defined, i.e. the meta-model of the language is built, GME can "interpret" this metamodel and morph itself into a domain-specific GME that supports that (and only that) language. This GME instance strictly enforces the language "rules": only models that comply with the abstract syntax and the static semantics can be built.

For the mapping into the semantic domain we have chosen a pragmatic approach: we assume that there is always a "target platform" whose semantics is well-known. This approach has been used in the past for formally specifying semantics[8]. Furthermore, the target platform also has an abstract syntax (with static semantics), and the transformation between the domain-

[2] Textual, or visual, yet precisely defined.

specific models and target platform models establishes the semantics of the DSM-s in terms of the target models. In other words, the semantics is captured in the form of (precisely specified) transformations. Thus, in MIC, transformations also play a crucial role: they specify the (dynamic) semantics of domain-specific models.

We have created a meta-programmable tool for implementing model transformations [5]: GRE (for Graph Rewriting Engine). GRE is programmed through a high-level language, called GReAT (Graph Rewriting And Transformations) captures metamodels of transformations as explicitly sequenced graph rewriting rules (see [6][7] for details). A graph rewriting rule consists of a graph pattern (which is matched against an input graph), a guard condition (which is evaluated over the matched subgraph), a consequence pattern (which captures what objects must be created in the target graph), and a set of attribute mapping actions (which describe how to compute the attributes of target objects from the attributes of input objects). The rewriting rules are transforming the input models (i.e. the input "graph") into the target models (i.e. the "output graph"). They explicitly reference elements of the input and the target metamodels, that is the pattern and consequence nodes of the transformation steps are strictly typed. Also, the input (target) subgraphs mentioned in the rules must be compliant with the input (target) metamodels. For efficiency reasons rewriting rules accept "pivot" points: initial bindings for pattern variables. This reduces the search in pattern matching process (effectively reducing it into matching on 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. GreAT works as an interpreter: it executes transformation programs (which are expressed in the form of transformation meta-models) on domain-specific models to generate target models. Obviously it runs slowly, although can provide great help in debugging transformation programs. We have also created a code generator that compiles the transformation specifications into executable code.

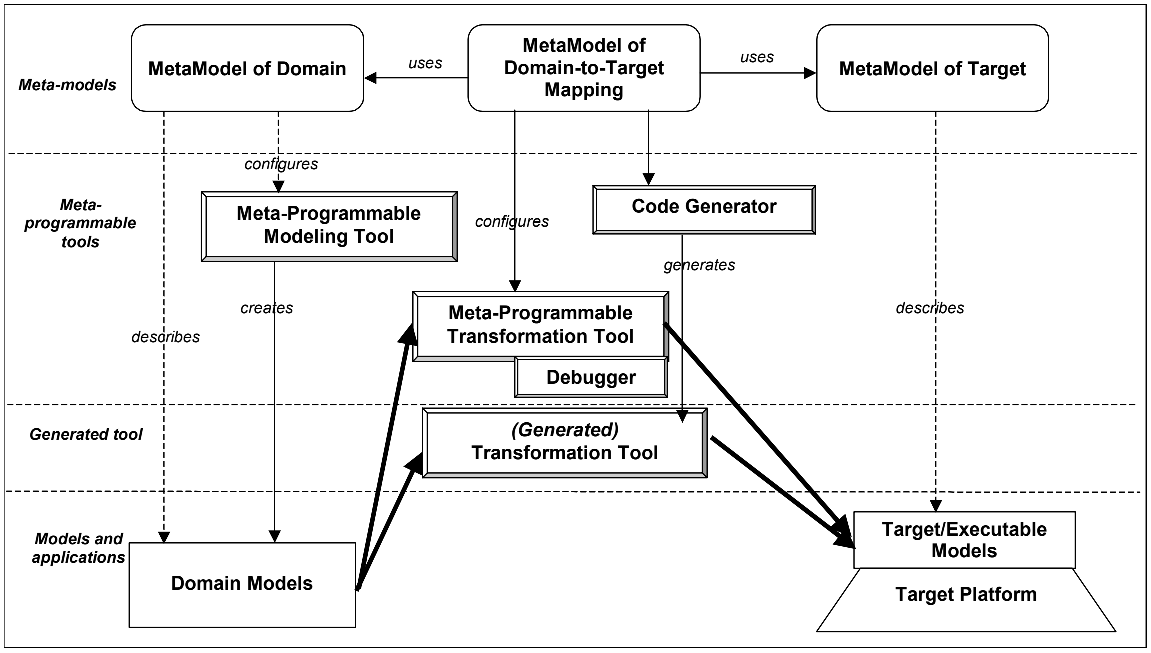
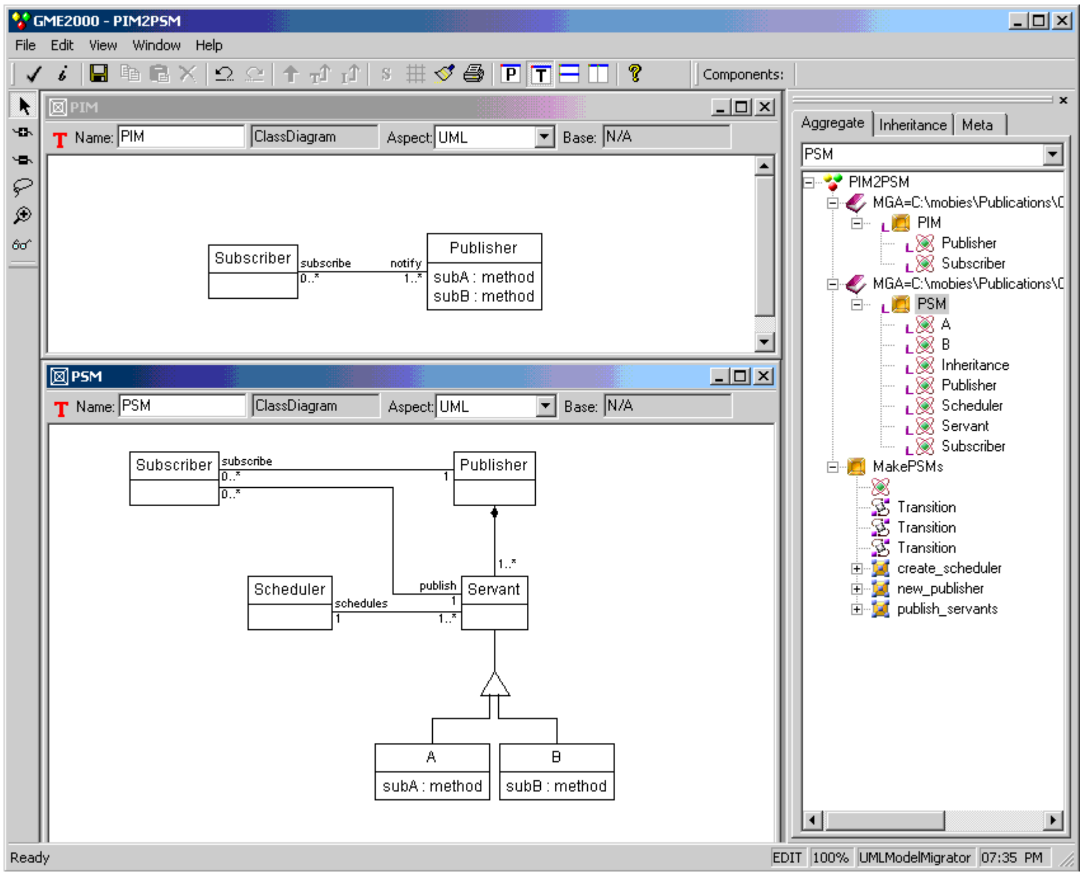


Figure 1: Tools of Domain-Specific MDA: MIC

Figure 1 shows the tools of MIC, and how they rely on meta-models. In order to set up a specific MIC process one has to create metamodels for the (input) domain, the (output) target, and for the transformations. One can then use the meta-programmable modeling tool (GME) to create and edit domain models, and the meta-programmable transformation tool (GReAT) to transform the models into target models. If the speed of the transformation is a concern, a code generator can be used to translate the transformation metamodels into executable code.

We believe a crucial ingredient in the above scheme is the meta-programmable transformation tool 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, also reasoning about the properties of the transformations.



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

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

Ready EDIT 100% UMLModelMigrator 07:35 PM

Figure 2: The Platform-Independent Model (top) and the Platform-Specific Model (bottom)

Example

The tools described above can also be used for implementing PIM/PSM transformations of the MDA as illustrated through the following example. The example shows the transformation of domain models from a more abstract, generic model to a model with more specialized components. On the top, Figure 2 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 2, 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.

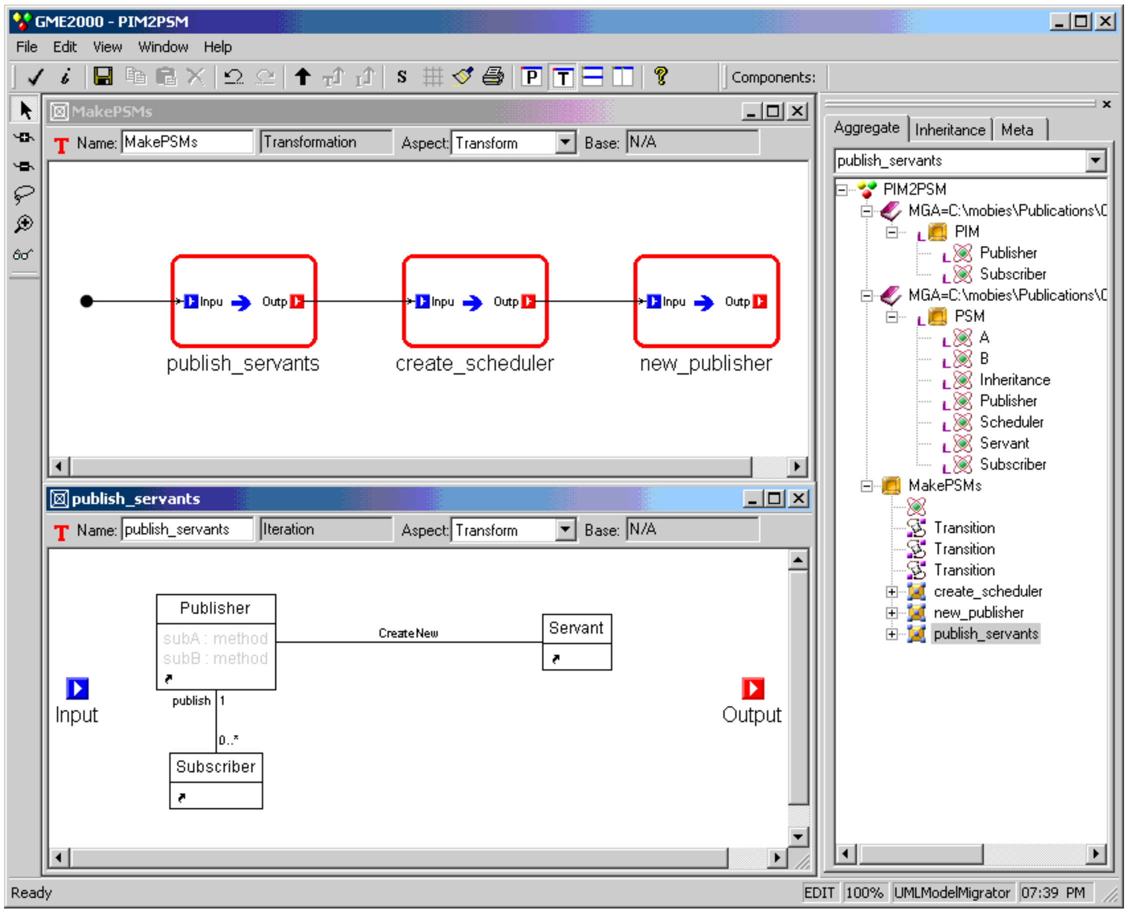


Figure 3: The rule execution order (above) along with the rewriting definition to change all Publishers to Servants (below)

Figure 3 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.

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 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 the consistent and recursive application of MDA and through the use of transformations the software engineering processes will be able to achieve the kind of reliability what the VLSI design processes could achieve.

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 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[3], 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.

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[14]. 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 for this 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.

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. We he briefly introduced a domain-specific model-driven process and its supporting tools, and summarized the major concepts behind it. For further details, please the papers in the reference.

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.

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

[3] We have actually experimented with specifying (and the bootstrapping) the code generator from a formal model introduced on Figure 1.

Jonathan Sprinkle have contributed to the discussions and work that lead to GreAT, and Feng Shi has written the first implementation of GreAT-E.

References

[1] J. Sztipanovits, and G. Karsai, “Model-Integrated Computing”, Computer, Apr. 1997, pp. 110-112

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

[3] http://www.omg.org/mda/

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

[5] 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.

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

[7] Blostein D., Schürr A., ”Computing with Graphs and Graph Rewriting”, Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.

[8] A. Maggiolo-Schettini, A. Peron, “A Graph Rewriting Framework for Statecharts Semantics”, Proc.\ 5th Int.\ Workshop on Graph Grammars and their Application to Computer Science, 1996.

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

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

[12] 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.

[13] 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

[14] Keith Duddy: UML2 must enable a family of languages. CACM 45(11): 73-75 (2002).