Model-Driven Engineering and its introduction with metamodeling tools

Tomaž Lukman, Marjan Mernik

Abstract— **This article discusses the opportunities that are offered by the paradigm shift form code-centric software engineering to Model-Driven Engineering (MDE) and one of the problems that hinder this shift. To enable MDE in practice sophisticated software tools have to be developed and employed. This paper presents and compares the available ways for developing such software tools and argues that the most reasonable of them is the development with metamodeling tools. Based on this finding we identified a pretentious question that is relevant to practitioners: "Which metamodeling tool should be procured?" The main contribution of this paper is the answer to this question that is given for a real-life project, which dealt with the procurement of a metamodeling tool.**

I. Introduction

SOFTWARE systems have proven their usefulness and efficiency across a myriad of application domains in which they have been adopted. Based on the success of software systems in these application domains, they are applied and are going to be applied in more and more additional application domains. Currently the markets demand for new software is not fully addressed by the software development industry. The *main challenges that the software engineering discipline* has to face are [1, 2]: How to sustainably increase the productivity and how to shorten the time-to-market periods for new software?

Unfortunately the *mainstream software development methodologies* used today do not stand up to these challenges. Their common denominator is that they still embody a code-centric paradigm i.e., they are largely leveraged by third generation programming languages (e.g., Java, C# and C++). The constructs of these general-purpose programming languages abstract the solution space - the domain of computing technologies. The problem or task that software has to solve is actually located in the problem space – the application domain in which the software will operate (e.g., healthcare, industrial process control, and insurance). There is a big semantic gap between these two spaces, which has to be bridged by a software system (the solution). Third generation languages are designed for the implementation of software systems that are dormant in a large set of various application domains, therefore their constructs are very general and full of implementation details, which are not relevant to the user but to the machine on which the software is supposed to run. This bares *two problems*, which are the *reasons why mainstream software development cannot stand up to the identified challenge*:

1. In order to describe a concept of a specific application domain with the general concepts of a third generation programming language more space is needed, than to do the same with concepts that are closer to that domain (are more domain-specific). Consequently this demands more effort form the developers and therefore causes a lower productivity rate.
2. The solution is cluttered with implementation details, in other words the solution is stated on a low level of abstraction. Therefore the developer does not only have to cope with the *essential complexities* (originating form the problem/task itself) but also with *accidental complexities* (originating from the implementation technologies) of the software development activity.

The statement that the present mainstream development methodologies are still code-centric often causes a lot disagreement. The arguments that supports this disagreement are: models have been used in software engineering for many decades; today they are adopted into the software lifecycle relatively often, due to the popularization of modeling by the de facto standard modeling language Unified Modeling Language (UML) [3]. Both of these statements are true but they do not disproof our claim. In these methodologies models are very often considered as second class development assets in particular as "mere" documentation [4], which is more like a necessary evil than a valuable development asset. Although (UML) models are constructed at the development of a software system they are frequently abandoned after a certain time. As software has to evolve, due to inevitable requirement changes, which happen over time, developers commonly change the code instead of the model. This way the model does not describe the actual software system, which makes the model invalid and consequently useless. In order to make the model valid again, labor costly updates of models have to be carried out. Although this synchronization issue is mitigated through the usage of automatic code generation, reverse engineering or even round-trip tools [5], they do not offer a solution for it. The source of this issue is the fact that only a small part of the code (e.g., class skeletons for UML models) can be generated from the models, because they are constructed with general-purpose modeling languages (e.g., UML). The additional part of the implementation has to be added

Tomaž Lukman is with the Department of systems and control at the Jožef Stefan Institute, Ljubljana, Slovenia (`tomaz.lukman@ijs.si`).

Marjan Mernik is with the Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia (`marjan.mernik@uni-mb.si`).

manually in into the generated code. Thus the introduction of models through UML and other general-purpose modeling languages has not raised the productivity to a sufficient level. The reason for this is that they do not address the first reason (problem) of insufficient productivity (i.e., they lack of domain-specific concepts).

II. Model-Driven Engineering

Model-driven engineering[1] (MDE) is a software development approach that has the potential to address the identified challenges of software engineering. It offers an environment that ensures the systematic and disciplined use of models throughout the development process of software systems. The essential idea of MDE is to shift the attention form program code to models. This way models become the primary development artifacts [4] that are used in a formal and precise way. The two main components that enable MDE are Domain Specific Modeling Languages (DSMLs) and model transformations.

A. *Domain-Specific Modeling Languages*

One of the ways to increase productivity is through reuse. Experiences [6, 7] have shown that closeness to the application domain has been the most effective vehicle for the reuse of knowledge and other software development assets.

Usually an average software development organization is specialized for the development of software within one or only a few application domains, so it has valuable expert knowledge about those domains. This domain-specific knowledge is often the organization's prime intellectually property [1] and can also be the source of its competitive advantage. In order to shield the organization against personnel fluctuation this knowledge has to be made explicit and one of the best ways to do so is to codify it into a DSML. A DSML formalizes the application structure, behavior, and requirements within a particular application domain [8]. Because of that we actually reuse this formalized knowledge every time we create a model with that particular DSML. A properly designed DSML enables only the modeling of meaningful (legal) applications within the domain it abstracts. Formally a DSML is a 5-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S), semantic mapping (M_S), and syntactic mapping (M_C):

$$L = \langle C, A, S, M_S, M_C \rangle.$$

The concrete syntax C defines the front end of the DSML - the notation of the language with which the user will model. This notation can be either textual or visual. We will focus on visual notations, because they make the best use of human visual perception [9]. Wisely chosen graphic symbols are more expressive and intuitively related to the application domain they abstract. Additionally such symbols help to flatten learning curves and also simplify the communication with domain experts (users of the software system) [8].

The abstract syntax A defines the concepts, relationships, and integrity constraints of the DSML [10]. The most frequent way to define the abstract syntax is through a metamodel (it can also be defined with graph grammars).

The semantic domain S is a domain, which is able to define the meaning of the models (this is usually the domain of computing or a formal/mathematical domain).

The semantic mapping M_S defines the meaning for each element of the abstract syntax with concepts from the semantic domain. The most common way to do this is with a model interpreter that interprets models and gives them meaning with the help of the underlying programming language in which the interpreter was written. Various types of interpreters can be provided in order to support different development tasks (e.g., code generators, model checkers).

The syntactic mapping M_C binds each element of the abstract syntax with a representation (elements from the concrete syntax).

B. *Model transformations*

MDE ensures that models are formally defined and precise, thus a partial automation of the software development process can be achieved. It is commonly accepted that automation is by far the most effective technological means for boosting productivity and reliability [1].

The automated parts of the development process are achieved through model transformations. A model transformation is the automatic generation of one or more target models from one or more source models, according to transformation definition(s). A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language [11]. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language [11]. MDE aims to automate many of the complex but routine development tasks which still have to be done manually today [9] with model transformations. Some of the tasks that are automatable with them are [12]:

- Generating lower-level models, and eventually code, from higher-level models.
- Mapping and synchronizing among models at the same level or different levels of abstraction.
- Creating query-based views of a system.
- Model evolution tasks such as model refactoring.
- Reverse engineering of higher-level models from lower-level models or code.

Czarnecki and Helsen [12] have proposed a separation between model-to-model transformations and model-to-code transformations, which are more often referred to as code generation. This separation is particularly important for practitioners, because most of them are first of all interested into automatic code generation.

[1] Also known as Model-driven development (MDD) or Model-driven software development (MDSD)

III. Importance of software tools

Sophisticated software tools are needed to build up an infrastructure that enables MDE and are therefore a vital element for the achievement of the advantages that MDE promises in practice. Stahl and Völter [13] go even a step further and claim that MDE does not make sense without tool support. The minimal features that software tools must assure in order to enable MDE with a particular DSML are:

- *Modeling environment* for the chosen DSML, which enables the creation and editing of visual models. This environment must also include a way of defining and enforcing constraints on the build models.
- *Artifacts generator* (*model-to-code transformation engine*), which enables the generation of source code, documentation and other development artifacts based on the given models.

These two features are necessary, but not even close to sufficient, for an efficient, effective and competitive development with the selected DSML. Essential features that developers have grown accustomed to are missing, therefore it is very likely that developers will reject MDE and rather stick to the development with third-generation programming languages, which offer high quality Integrated Development Environments (IDEs) that possess a variety of features for the simplification and acceleration of software development (e.g., Eclipse, Microsoft Visual Studio.NET).

Based on a literature survey and on our own experiences some of the additional features that are useful are:

- *Model debugger* – the development of today's complex and extensive software is hardly imaginable without debugging capabilities. Debugging capabilities should also be available on the modeling level.
- *Model validation* – models are validated with the constraints that are present in the domain they belong to.
- *Model-to-model transformation engines* – to enable advanced development tasks on the available models a mode-to-model transformation engine is needed. Examples of such tasks are: model refactoring [14], and exploration of design alternatives [15].
- *Test suite* – enables testing on the modeling level.
- *Model analysis tools* – enable analysis of the constructed models in various ways e.g., assessing the quality of models (this is done with model metrics).
- *Model simulators* – in some domains (e.g., embedded software) the execution on the real platform is not rational (e.g., the upload and execution of the program take a long time) or not possible, therefore simulation capabilities on the modeling level are much desired.

The more of these features are available the bigger are the chances that developers will accept MDE.

IV. Development of software tools

The task to develop a tool set that enables all the presented features is very pretentious and even more so when these features have to be implemented into one IDE, which is preferred by most of the users. Due to the inevitable evolution of the DSML or the domain it abstracts [16], the implementations of all of features have to be modified in a consistent way, which is another major challenge for the developers. As if this was not enough a software development organization can have a set of DSMLs, therefore this development task has to be carried out several times – for each of them.

It is necessary to emphasize that there are *different paths (approaches) to reach the goal of developing an IDE for a particular DSML.* We will classify these paths into three categories, which are depicted in Figure 1.

The *development from scratch* (X on Figure 1) is the development path without any reuse. More precisely: no assets (libraries or frameworks) that are external to the implementation language are reused. Because of that this kind of development demands the advanced skills of building interpreters/parsers, diagramming user interfaces and other nontrivial components. On the other hand the developers are not constrained by the capabilities of the assets they reuse; therefore the resulting IDE can be very specific. Unfortunately most of the IDEs developed this way have a hardcoded DSML definition, which makes them difficult to change in order to follow the DSML or domain evolution.

The *development with reuse* (Y on Figure 1) is the path that makes reuse of one or more assets (libraries or frameworks). Most commonly the reused asset is a diagramming library. Popular assets to be reused are also template engines, which can be used as artifacts generators. Because of the mentioned reuse less of the DSML definition is hardcoded. The main challenges of this development path are: customization of the reused assets, which have to be very well understood, and the integration of custom code and the reused assets.

The *development with metamodeling tools* (Z on Figure 1) is the path that enables the development of an IDE for the selected DSML based on the formal definition of that DSML. A metamodeling tool is an IDE that allows the definition of an arbitrary DSML and consequently the generation of a model-driven IDE for that DSML [17]. One of the aims of metamodeling tools is to reduce the amount of coding that has to be done with general-purpose programming languages in the discussed context.

The main question is: *Which of these paths is the most suitable in the majority of situations?* The answer to this question is provided by the following reasoning, which is done with the help of Figure 1 (the development cost in this context is defined as a composite of the invested money and time).

Some initial definitions to clarify the scheme in Figure 1:

$$\boldsymbol{D} = \{\boldsymbol{I}, \boldsymbol{II}, \boldsymbol{III}, \dots\} \text{ – \textit{Application domains}}$$
$$\boldsymbol{\mathcal{L}} = \{\boldsymbol{d_1}, \boldsymbol{d_2}, \boldsymbol{d_3} \dots\} \mid \forall \boldsymbol{d} \in \boldsymbol{D} \text{ – \textit{DSMLs}}$$
$$\boldsymbol{\Omega} = \{\boldsymbol{\omega_1}, \boldsymbol{\omega_2}, \boldsymbol{\omega_3} \dots\} \text{ – \textit{Identified metamodeling tools}}$$
$$\boldsymbol{\Psi} = \{\boldsymbol{\psi_1}, \boldsymbol{\psi_2}, \boldsymbol{\psi_3} \dots\} \text{ – \textit{Identified reusable assets}}$$

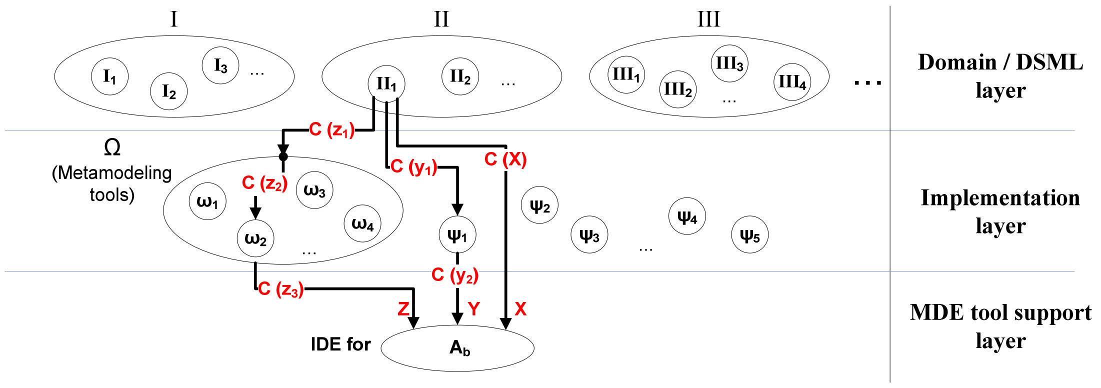


Figure 1: **The path from an arbitrary DSML to an IDE for it.**

The following definitions are true for *an arbitrary DSML out of any application domain*:

$\boldsymbol{C\ (X)}$ – *Cost of X*

$\boldsymbol{C\ (y_1) + C\ (y_2), \forall\ \psi_n, n}\ \in \mathbb{N}$ – *Cost of Y*
$\boldsymbol{C\ (y_1)}$ **–** *Cost of finding & procuring a reusable asset*
$\boldsymbol{C\ (y_2)}$ **–** *Cost of the development with the selected asset*

$\boldsymbol{C\ (z_1)\ +\ C\ (z_2)\ +\ C\ (z_3), \forall\ \omega_m, m}\ \in \mathbb{N}$ – Cost of *Z*
$\boldsymbol{C\ (z_1)}$ **–** *Cost of finding the available metamodeling tools*
$\boldsymbol{C\ (z_2)}$ **–** *Cost of selecting a metamodeling tool*
$\boldsymbol{C\ (z_3)}$ **–** *Cost of developing with the selected metamodeling tool*

The *development from scratch* can be expensive and time-consuming. The needed development effort according to [18] is at least one man-year of work. Although this number and claims are not supported by any quantitative research, the existing literature disseminates and accepts the dogma that the *development with metamodeling tools* is much cheaper than the *development from scratch*. Our first deduction (1) is based on the explicit expression of this opinion in [13, 18, 19]:

$$\boldsymbol{C\ (X) \gg C\ (z_1)\ +\ C\ (z_2)\ +\ C\ (z_3)} \qquad (1)$$

According to the sources [18, 19] the *development with metamodels* is also much cheaper than *the development with reuse*. We agree with this assumption, above all because of our own experience with the development of a model-driven IDE for the ProcGraph language [20], which are documented in [21] and [22]. Based on this we deduct (2):

$$\boldsymbol{C\ (y_1) + C\ (y_2) \gg C\ (z_1)\ +\ C\ (z_2)\ +\ C\ (z_3)} \qquad (2)$$

Considering (1) and (2) we infer that *metamodeling tools have to be employed in order to reduce the cost of MDE introduction.*

An exception to the “develop with metamodeling tools” rule is the tool support for a DSML, which is abstracting a domain that has a sufficiently large (potential) user base and/or market. In this case the investment into development from scratch is reasonable [19]. Well known examples of such tools are LabVIEW - a graphical programming environment and Simulink - a hierarchical block-diagram design and simulation tool.

Currently there are around 10 metamodeling tools available on the market, which have very different capabilities. Therefore practitioners are facing the pretentious question: **“***Which metamodeling tool should we procure?* The procurement of a sub-optimal or even an improper metamodeling tool will have profound negative consequences: unjustified expenses (tool purchase, developer work); losing ground (several months) with the competition; the risk of project failure.

Unfortunately this question has been neglected in the literature (no obvious answer to this question can be found), even though it could importantly contribute to the minimization of MDE introduction costs.

Although a considerable amount of articles about metamodeling tools can be found, the majority of them are biased, because they describe the metamodeling tool that was developed by the article authors. We have noticed only a few articles [23, 24] that are comparing different metamodeling tools, but they are not sufficiently helpful for practitioners, because they review too few of them.

This paper describes a disciplined process of procuring the fittest metamodeling tool for a real-life project and thus may be of interest for MDE practitioners and even more the ones who intend to become such practitioners. This project is the development of an IDE for the ProcGraph language [20].

V. The Process of a Metamodeling Tool Procurement

The general process of procuring the fittest metamodeling tool as we see it is presented in Figure 2. The initiation of this process should be the decision of stakeholders to introduce MDE into their organization. An important input into this process is the definition of the DSML for which the IDE will be developed. The highlighted phases in Figure 2 represent the most important activities of the process. The following subsections documented their execution for the project we have undertaken. The presented process includes also an exception, which occurs if none of the metamodeling tools is suitable; the workflow in that case is modeled in Figure 2 and should be self-explanatory. The process can end when the stakeholders decide (based on the selection report) if the selected metamodeling tool should be procured.

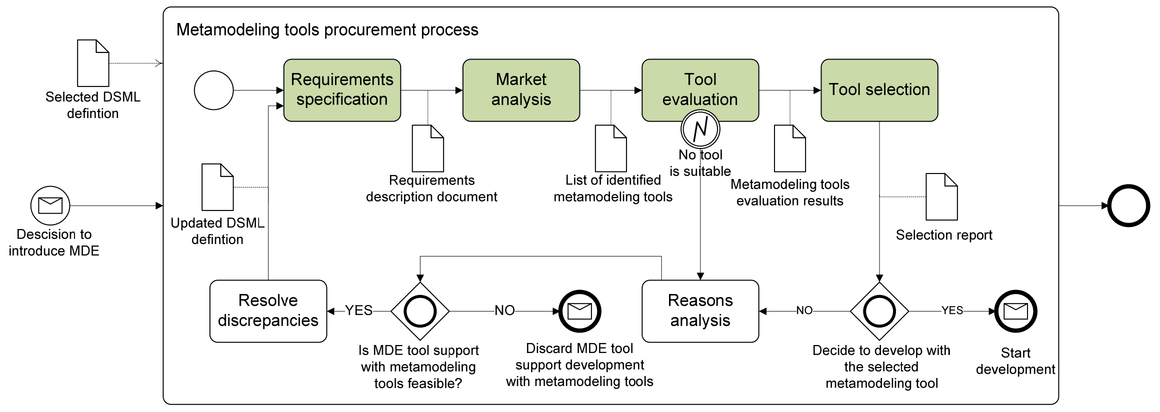

Metamodeling tools procurement process
Selected DSML defintion
Descision to introduce MDE
Updated DSML defintion
Requirements specification
Market analysis
Tool evaluation
Tool selection
Requirements description document
List of identified metamodeling tools
No tool is suitable
Metamodeling tools evaluation results
Selection report
Resolve discrepancies
YES
NO
Is MDE tool support with metamodeling tools feasible?
Discard MDE tool support development with metamodeling tools
Reasons analysis
NO
YES
Decide to develop with the selected metamodeling tool
Start development

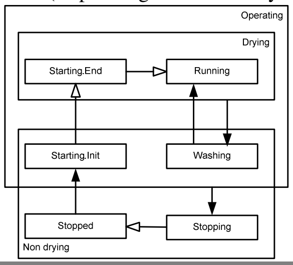
Figure 2: **The metamodeling tool procurement process as we see it.**

A. *Requirements specification*

The requirements specification phase is one of the most important development phases as it defines what the software system should do. The mistakes and misconceptions made here are the ones which demand the most work and expenses for their correction.

The requirements in our project had to be stated in a way that would enable the evaluation and selection of metamodeling tools. The requirements actually became the evaluation criteria. We developed a template for each requirement, which can be seen in Table I. A measurement scale from 1 to 5 was defined in order to denote the importance of each requirement. The stakeholders define the importance of each requirement according to this scale. The number 5 was chosen to denote that a requirement is mandatory. After this phase we produced a requirements description document with 26 requirements.

TABLE I: REQUIREMENT EXAMPLE

Id & short name:	*R2.4 - Superstates can overlap*
Importance:	3 - Important
Preface:	ProcGraph notation does allow the existence of overlapping superstates, which can be beneficial in some situations.
Description:	*It should be possible to create superstates that can overlap.*
Example:	A state transition diagram with overlapping superstates ("Operating" and "Non drying"):  Operating Drying Starting.End Running Starting.Init Washing Stopped Stopping Non drying

B. *Market Analysis*

This phase includes scanning the market for currently available metamodeling tools and the identification of the ones who could be potentially useful for our project.

Our search, which was also committed on the databases of scientific publications, identified 5 candidates (see Table II).

TABLE II: IDENTIFIED METAMODELING TOOLS

Tool name	*Vendor/Author*	*Execution environment*
GME	ISIS @ Vanderbilt University	stand-alone
DSL Tools	Microsoft	VS.NET 2005
MetaEdit+	MetaCase	stand-alone
GMF	open-source & industry partners	Eclipse
GEMS	ISIS @ V. U. & industry partners	Eclipse

C. *Tool evaluation*

The essence of this time consuming and labor-intensive phase is to evaluate the capability of the identified candidate tools to meet every requirement that was specified.

Each fulfillment of a requirement was evaluated: with 1 if the requirement was fully fulfilled, with ½ if the requirement was fulfilled satisfactory, but not optimal and with 0 if the requirement was unfulfilled or fulfilled unsatisfactory.

In the first step we evaluated all the candidates only with the mandatory requirements, this way a quick elimination of the least fit candidates was possible. In our project we dismissed GME, MetaEdit+ and GEMS.

The second step was to evaluate the remaining candidates in-detail according to all the non-mandatory requirements. The result of this can be viewed in Table III (Eval. column).

D. *Selection*

In this phase we decide which tool is the fittest. To make a decision we used the weighted score method (WSM) [25]. The weighted score for each requirement evaluation is calculated based on the equation (Table III, Sel. column):

TABLE III: THE SECOND EVALUATION STEP & SELECTION

Req. Id	*Req. Importance*	*DSL Tools*		*GMF*	
		Eval.	*Sel.*	*Eval.*	*Sel.*
R1	5	1	5	1	5
R1.1	4	0	0	1	4
R1.2	4	1	4	1	4
R1.3	5	1	5	1	5
R1.4	5	1	5	1	5
R1.5	3	1	3	½	1,5
…					
R11	2	½	1	1	2
Final score			89,5		**93,5**

$$score_{weighted} = requrement_{importance} * score_{evaluated}$$

The final score, which gives a numeric indication of the fitness of a candidate, is calculated (where n is the number of all the specified requirements):

$$score_{final} = \sum_{j=1}^{n} score_{weigthed\ j}$$

The fittest metamodeling tool for our project was GMF. It reached the highest score, which can be seen in Table III.

VI. Related work

The literature provides only general procurement processes that are dealing with the procurement of "Commercial Off The Shelf" (COTS) software. COTS is a software product that already exists, that is supplied by a vendor and that has specific functionality [26]. Metamodeling tools can be classified as COTS software, therefore COTS procurement processes could be used at the procurement of metamodeling tools. The crucial question is: "Should they?"

Currently there is a great variety of COTS procurement processes available that are documented in the literature. Some of the most important are [27]: the OTSO (Off-The-Shelf Option) approach, the PORE approach, the CAP (COTS acquisition process) approach, the CARE (COTS-Aware Requirements Engineering) approach and the MiHOS (Mismatch-Handling aware COTS Selection) approach. Although intensive research efforts have been spend on the development of these methods, none of them can be considered as the silver-bullet to solving the COTS selection problem [27]. Because of that each practitioner stands in front of the challenge: "Which of these methods should be used for our project?" Even this decision turns out to be difficult and requires considerable effort. In [28] R. Glass stated: "What help do practitioners need? We need some better advice on how and when to use methodologies." One of the most burning research challenges in the area of COTS procurement is how to adopt an arbitrary COTS selection process into different specific contexts [27].

Based on these open questions it is our opinion that COTS selection processes are too generic to aid MDE practitioners at the procurement of metamodeling tools.

VII. Conclusion

This article documented the project of procuring the fittest metamodeling tool for the implementation of an IDE for the ProcGraph language. This example may be useful for MDE practitioners, which are facing a similar challenge and is even more important as an initial step towards the development of guidelines and eventually a method for the procurement of the fittest metamodeling tool (in the given situation). These guidelines and method would provide a more stable context for the introduction of MDE, because they would minimize the procurement cost and risk.

VIII. References

[1] B. Selic, "The pragmatics of model-driven development," IEEE Software, vol. 20, no. 5, pp. 19-25, 2003.

[2] S. Sendall, and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," IEEE Software, vol. 20, no. 5, pp. 42-45, 2003.

[3] The Object Management Group, Unified Modeling Language: Superstructure, Version 2.0, 2004.

[4] E. Seidewitz, "What models mean," IEEE Software, vol. 20, no. 5, pp. 26-32, 2003.

[5] S. Demeyer, S. Ducasse, and E. Tichelaar, "Why unified is not universal: UML shortcomings for coping with round-trip engineering," LNCS vol.1723, pp. 630-645, 1999.

[6] T. J. Biggerstaff, "A perspective of generative reuse," Annals of Software Engineering, vol. 5, pp. 169-226, 1998.

[7] W. B. Frakes, and K. Kyo, "Software reuse research: status and future," IEEE Transactions on Software Engineering, vol. 31, no. 7, pp. 529-536, 2005.

[8] D. C. Schmidt, "Model-Driven Engineering," IEEE Computer, vol. 39, no. 2, pp. 25-31, 2006.

[9] C. Atkinson, and T. Kuhne, "Model-driven development: a metamodeling foundation," IEEE Software, vol. 20, no. 5, pp. 36-41, 2003.

[10] K. Chen, J. Sztipanovits, and S. Neema, "Toward a semantic anchoring infrastructure for domain-specific modeling languages," in 5th International Conference on Embedded Software, 2005, pp. 35-43.

[11] A. G. Kleppe, J. Warmer, and W. Bast, MDA Explained: The Model Driven Architecture: Practice and Promise: Addison-Wesley 2003.

[12] K. Czarnecki, and S. Helsen, "Feature-based survey of model transformation approaches," IBM Systems Journal, vol. 45, no. 3, pp. 621-646, 2006.

[13] T. Stahl, and M. Volter, Model-Driven Software Development: John Wiley & Sons, 2006.

[14] R. France, S. Ghosh, E. Song et al., "A Metamodeling Approach to Pattern-Based Model Refactoring," IEEE Software, vol. 20, no. 5, pp. 52-58, 2003.

[15] J. Gray, Y. Lin, and J. Zhang, "Automating change evolution in model-driven engineering," IEEE Computer, vol. 39, no. 2, pp. 51-58, 2006.

[16] R. France, and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in FOSE'07, 2007, pp. 37-54.

[17] N. Zhu, J. Grundy, J. Hosking et al., "Pounamu: A meta-tool for exploratory domain-specific visual language tool development," Journal of Systems and Software, vol. 80, no. 8, pp. 1390-1407, 2007.

[18] J.-P. Tolvanen, "MetaEdit+: domain-specific modeling for full code generation demonstrated," in 19th Annual OOPSLA Conference, 2004, pp. 39-40.

[19] A. Ledeczi, A. Bakay, M. Maroti et al., "Composing domain-specific design environments," IEEE Computer, vol. 34, no. 11, pp. 44-51, 2001.

[20] G. Godena, "ProcGraph: a procedure-oriented graphical notation for process-control software specification," Control Engineering Practice, vol. 12, no. 1, pp. 99-111, 2004.

[21] G. Kandare, "Računalniško podprto načrtovanje programske opreme za postopkovno vodenje s programirljivimi logičnimi krmilniki," Doctoral Dissertation, Faculty of Electrical Engineering, 2004.

[22] T. Lukman, "Model driven engineering in the domain of industrial control systems," Undergraduate Thesis, University of Maribor, Faculty of Electrical Engineering and Computer Science, 2007.

[23] B. Carsten, "Model-Driven HMI Development: Can Meta-CASE Tools do the Job?," in 40th Annual Hawaii International Conference on System Sciences, 2007, pp. 1530-1605.

[24] S. Kelly, "Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM," 19th Annual OOPSLA Conference, 2004.

[25] J. Kontio, S. F. Chen, K. Limperos et al., "A COTS Selection Method and Experiences of Its Use," in 20th Annual Software Engineering Workshop, 1995.

[26] N. A. Maiden, and C. Ncube, "Acquiring COTS software selection requirements," IEEE Software, vol. 15, no. 2, pp. 46-56, 1998.

[27] A. Mohamed, G. Ruhe, and A. Eberlein, "COTS Selection: Past, Present, and Future," ECBS '07, 2007, pp. 103-114.

[28] R. L. Glass, "Matching methodology to problem domain," Communications of the ACM, vol. 47, no. 5, pp. 19-21, 2004.