Defining Model Driven Engineering Processes

Frédéric Fondement and Raul Silaghi
Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland
{Frederic.Fondement, Raul.Silaghi}@epfl.ch

Abstract

Software engineering techniques made it possible for developers to build larger, and more accurate, reliable, and maintainable software-intensive systems. This was essentially possible by introducing techniques for raising the level of abstraction for describing the problem and its solution, and by clearly establishing a methodology to define both the problem and how to move to its solution. Model Driven Engineering (MDE) targets precisely at organizing such levels of abstraction and methodologies. It encourages developers to use models to describe both the problem and its solution at different levels of abstraction, and provides a framework for methodologists to define what model to use at a given moment (i.e., at a given level of abstraction), and how to lower the level of abstraction by defining the relationship between the participating models. Such an MDE process is supposed to be defined by means of assets, and methodologists have the duty to provide such assets. However, it is not yet clear what exactly these assets are, despite the fact that techniques to express them have already been widely studied. This position paper addresses this issue by identifying some of the MDE assets that have to be provided, and shows how they should be defined in order to enable them to participate in different MDE process definitions.

Keywords. *Model Driven Engineering, MDE, Model Driven Architecture, MDA, Component-Oriented Programming.*

1. Introduction

Software engineering has enabled developers to build more important and reliable systems over the years. The number of lines of code that implement software systems has increased significantly over the last forty years, moving from ten thousand to ten million lines of code nowadays [1]. Three main categories of techniques have allowed developers to manage the increasing complexity of software-intensive systems. First, methodologies, such as SADT, Catalysis, B, (Enterprise) Fondue, or Extreme Programming, defined clearly each step of a development process. Second, mechanisms for raising the level of abstraction, such as functional programming, object-orientation, middleware, or aspects, allowed developers to better encapsulate complexity, and thus, to produce more modular, reusable, and extensible programs. Third, software verification and testing helped to enforce the quality of the final system.

Model Driven Engineering (MDE) [2] attempts to organize new efforts in these directions, and offers the possibility (1) to clearly define methodologies, (2) to develop systems at any level of abstraction, and (3) to organize and automate the testing and validation activities. Moreover, this technique states that any specification should be expressed by *models*, which are both human and machine understandable. Models, depending on what they represent, can reside at any level of abstraction, and can be restricted to address only certain aspects of the system. Since they are all machine understandable, several collaborative tools can automate (at least partially) a certain number of tasks, such as model refactorings, model refinements, or model to code generation. As a consequence, the process of developing systems becomes iterative, refining abstract models to more concrete ones, and in the end, automatically generating and deploying the complete code.

Several Computer Aided Software Engineering (CASE) tools were developed since the late eighties with the same purpose of facilitating the development and maintenance processes, most of the time to support a software development method. Unfortunately, many problems that appeared back then are still present nowadays. It is hard to choose the right tools for answering the different needs of the different stakeholders [3], with the right levels of abstraction [4], supporting the right platforms and methodologies, and providing several features, such as prototyping, test and validation, version control, traceability, reusability, or even synchronization between models and their implementation. Moreover, most of the time it is impossible to make these different CASE tools interoperate. In the end, only a few

tools that impose a very specific method and well-defined notations will be able to accompany a system during its complete lifetime. The problem is that the lifetime of a system can be longer than thirty years! Relying on a single or a small set of CASE tools, which might not be supported in the long term, is often not acceptable. A solution is to develop agreed-upon *standards* that all CASE tools should use during the entire life cycle of software systems.

In order to address some of these problems in the context of MDE, the Object Management Group (OMG) [5] has launched the Model Driven Architecture (MDA) initiative [6], challenging to gather and define all specifications necessary to support the MDA approach to software development. While some of these specifications have already been delivered, others are on their way. Their declared intent is to define precisely what language should be used to express models [7][8], how to specify model transformations [9], how to exchange models [10], how to store and make models evolve [11][12], and, more recently, how to generate code [13]. However, since MDA is still in its early phases, and since there are not many industrial applications using MDA, some of these specifications lack precision, while others are still missing. In order to overcome technical problems, such as interoperability, versioning, or transformations, standardization seems to be the right solution. However, because of the freedom MDE gives to methodologists, standardization will not solve problems like complexity and accuracy, which are inherent to the provided methodology and its associated notations. Moreover, managerial issues like voluntariness and support, main responsibles for the success of a method according to [14], will certainly not be addressed either. One should remember that "structured mess is still mess!".

MDE will significantly help software engineers of any domain (e.g., cellular phones, automotive embedded systems, nuclear plant control) by clearly specifying a *process* and by constraining all tools that intent to support it to comply with a well-defined set of rules. However, such an MDE process will not solve all problems since there will always be systems for which the MDE process will simply not be adaptable to support their development. Dismissing the study of this weakness that survived the state of the art before MDE, the goal of this position paper is to briefly present (in section 2) how the activities of MDE methodologists and developers can be performed, and to introduce (in section 3) the idea of reusable MDE components for defining MDE processes. Concluding remarks and future work directions are presented in section 4.

2. Model Driven Activities

Model driven activities are the set of actions to be taken by methodologists, on one hand, and by system developers, on the other hand, in order to *define* a clear MDE process, and to *apply* it, respectively. After presenting briefly what does it mean to apply an MDE process from the point of view of developers, we deduce in the second part what are the exact assets to be provided in order to clearly define an MDE process from the point of view of methodologists.

2.1. Applying an MDE Process

The idea promoted by MDE is to use models at different levels of abstraction for developing systems. Thus, the main activity of MDE developers is to design models, just like they used to develop code, but led by a methodology this time. The advantage of having an MDE process is that it should clearly define each step to be taken, forcing the developers to follow the defined methodology in this way. It should specify the sequence of models to be developed, and how to derive a model from another one at the abstraction level immediately above it. By providing developers with such a methodology, they are supposed to know at any moment during the development life cycle what is to be done next and how to achieve it.

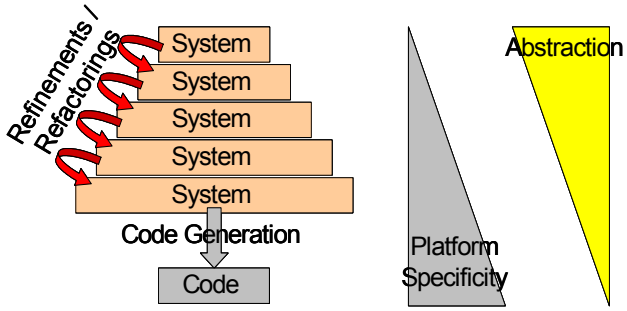

Figure 1. Model Driven Refinements

Applying an MDE process is depicted in Figure 1. The system under development is first described by a model at a very high level of abstraction, i.e., ignoring any kind of platform-related dependencies. Such a model is referred to as a *Computational Independent Model (CIM)* in the MDA terminology. This kind of model is intended to capture only the system requirements, without specifying how to achieve them; it is the description of the problem. Good candidates to play the CIM role are use cases [15] and feature-oriented diagrams [16]. In this paper, we consider the CIM as a unique model, since its platform-independence remains absolute even though several languages and refinements are involved for making its details more accurate. A series of interactive refinements may then be performed that have the responsibility to make the system more platform-specific at each refinement step. For instance, the system may be expressed once again, but more precisely this time, by class diagrams and state diagrams [17] to show business behavior. Further on, additional information may be added to in-

tegrate middleware-specific concerns, such as the distribution concern. Another refinement could further enhance the resulted distribution-aware model with information specific to the concrete CORBA middleware technology, before generating code for a specific CORBA middleware platform (e.g., OpenORB), as explained in [18]. In this paper, a refined model will be called a *Platform Specific Model* (*PSM*), and a model at the abstraction level immediately above, which was the source for the corresponding refinement step, will be called a *Platform Independent Model* (*PIM*). These terminologies have already been introduced by MDA, however, we *relativize* them in this paper, i.e., each PIM is relative to a PSM, and vice-versa. As a consequence, models that are not at the highest or lowest level of abstraction play the role of PSMs first, and PIMs afterwards.

At each step of the MDE process, information related to quality management could be integrated as well, such as verification, validation, and test case generation. A verification might be the action that checks whether a PSM does not break the specification promoted by its PIM, or vice-versa in the case of reverse engineering. A validation step may allow system developers (or even clients) to instantiate prototypes out of intermediate models in order to test their functionalities before the system is fully implemented. Automatic test case generation may produce as outcome scenarios, i.e., sets of messages that are supposed to be sent and received by the working system, allowing in this way to test its actual implementation.

One of the most important advantages of using an MDE process is its adaptability to changes. When a change occurs, be it at the highest level of abstraction (e.g., a change in the requirements of the system) or at a lower level of abstraction (e.g., moving to another platform, such as moving from PostgreSQL to MySQL), its impact is well localized and the parts that are not touched by the change are immediately reusable. However, the refinements have to be performed once again in order to "update" the changing parts. It becomes more problematic when the modeling language changes because such *re-refinements* are not directly possible.

In order to apply MDE-inspired processes in large projects, which typically involve many developers and tools, several issues have to be addressed, such as model interchange, diagram interchange, model versioning, concurrent management, and so on. Several MDA specifications are targeting some of these issues [10][11][12], but these are more tool-related issues than methodology issues. Nevertheless, they remain problems that will have to be addressed sooner or later [19].

2.2. Defining an MDE Process

As one can deduce from the previous section, an MDE process should define:

1. how many levels of abstraction are there, and what platforms have to be integrated;
2. what are the modeling notations and the abstract syntax to be used at each level of abstraction;
3. how refinements are performed, and what platform and additional information they integrate into the lower level of abstraction;
4. how code is generated for the modeling language used at the lowest level of abstraction, and perhaps even how to deploy that code;
5. how can a model be verified against the upper level model, how can it be validated, and how can it generate test cases for the system under development.

The first important technique is *metamodeling* [20], which allows methodologists to define precisely a class of models. Metamodeling clearly defines a modeling language by specifying its abstract syntax, eventually along with its semantics. We also believe it is of paramount importance to define its possible concrete syntaxes in order to allow conforming models to be viewed and modified by different stakeholders using the different modeling notations that are available in a given view. If we have a closer look, one level of abstraction is already defined by the modeling language to be used. Therefore, defining the corresponding metamodels solves already the first two duties of methodologists (points 1 and 2 presented above). Moreover, if the semantics is clearly defined, it is also possible to perform the validation part of point 5. In terms of standards, MDA proposes the Meta Object Facility [8], a specification that has proven its accuracy in defining the abstract syntaxes of several modeling languages, and that was implemented by several tools. Unfortunately, it does not provide any means do define the semantics nor the concrete syntaxes.

The second technique is *model transformation* [21]. This technique allows methodologists to clearly define relationships between models. Model transformations depend only on the metamodels of the related models. Methodologists may use this technique to clearly specify the refinements from PIMs to PSMs. MDA is about to deliver the MOF Query/View/Transformations specification [9] to provide methodologists with a means to realize the model transformation technique. Moreover, the MOF QVT specification promotes both forward and reverse transformations, which will allow to propagate changes to models at lower, respectively upper, levels of abstraction, enabling the possibility of automatic synchronizations and re-refinements. Moreover, it gives the possibility to verify the PSM model against the PIM model, and vice-versa, at any moment dur-

ing the development life cycle. When tools implementing the MOF QVT specification will be available, point 3 and the verification part of point 5 will be solved as well.

For solving point 4, *code generators* are needed, which would map a model to some textual or binary files. In order to address this issue, OMG has recently launched the MOF2Text request for proposal [13].

Referring to point 5, an important problem is the current lack of a tool-independent solution and of appropriate modeling notations for completely specifying how to generate test cases out of models, how to perform validation and deployment, how to depict such models, and so on. Moreover, one should be aware that the proposed list of assets to be delivered by methodologists is not at all complete, and new points will probably have to be added as MDA moves along.

3. MDE Components

Defining an MDE methodology is a hard and long task, which involves specialists in domains, platforms, testing, verification, validation, etc. In this section, we investigate how methodologists could provide the assets discussed in section 2.2 in a modular way that would facilitate their reuse by other methodologists. Moreover, solutions are proposed for the unresolved points of section 2.2. We present first some already existing mechanisms that solve the reuse issue up to a certain point, how to use such mechanisms, and what are their limitations and weaknesses. In order to solve the identified problems, we propose in the second part some light techniques that would allow methodologists to take advantage of existing results that already solve some MDE-related issues.

3.1. Current MDE Techniques and Their Limitations

The currently existing techniques that address the reusability of assets provided by methodologists are defined either at metamodeling or model transformation level. However, these techniques are not universal enough yet. Similar ideas are discussed in [22].

One such technique is the basic *package dependency*. This relationship enables methodologists to reuse concepts defined by other metamodels when defining new metamodels. For instance, if we have a closer look at the way UML 2.0 [17] and MOF 2.0 [8] are built, one will immediately notice that they both depend on the same UML 2.0 Infrastructure [7] using such a package dependency technique. Moreover, UML 2.0 introduces the «merge» mechanism, which allows methodologists to add new properties to the metaclasses of the imported packages. Additional OCL constraints [23] may be used as well in order to better tailor the imported packages to the exact needs. However, one should notice that the package dependency technique requires the existence of packages of such reusable metamodels. Fortunately, the UML metamodel presents a very modular package architecture, allowing some packages to be reused without the others. With respect to concrete syntax definitions, the issue of merging such definitions is not at all taken into account by this technique.

Another technique is *profiling*, which allows an external asset, also referred to as a *profile*, to extend a given metamodel for storing new information in the conforming models. The profiling technique is mainly based on the principles of branding (i.e., `stereotypes`) and associated key/value pairs (i.e., `tags`). It provides methodologists with the possibility to enhance metamodels independently from the methodology. For instance, a refinement step may simply apply a profile through a model transformation. Following the profiling technique, platform specialists have the opportunity to formalize their knowledge in reusable data structures (i.e., profiles), on one hand, and model transformations to apply such profiles to concrete models, on the other hand. Several platforms have been already described this way, e.g., the UML profiles for EJB [24] or CORBA [25]. Moreover, profiles may extend each other using the «merge» mechanism, allowing methodologists to describe abstract platforms and to provide a sequence of refinements to more concrete platforms (following exactly the MDE scheme), provided that the metamodel does not change, like it is the case in [18]. Unfortunately, profiles can be used only if the metamodel embeds support for a profiling technique. Moreover, a given profile can only be applied to one, and only one, modeling language. With respect to concrete syntax definitions, the issue of extending such definitions is not at all addressed by profiles.

Model transformations are supposed to be reusable thanks to the *interoperability* that everybody hopes *MOF QVT* [9] will provide. Since all transformation languages are supposed to have a common minimal core at abstract syntax level, a transformation may invoke, or even extend, another transformation. However, as it stands today, the QVT specification is neither finalized nor supported by any tool. Moreover, the belief that a unique standard for the complete universe of model transformations will exist one day is somehow an utopia, just like it was believed that all systems will be described by UML, and only UML, or metamodels by MOF, and only MOF.

As suggested in [26], another important technique to be taken into account is *semantically rich metamodels*, i.e., metamodels that come along with specific support to address an issue that has been neglected till then. An appropriate example is the Action Semantics [27] extension to UML, which is now part of the UML specification since

version 1.5. It is interesting to notice that tools already exist to interpret (or, in other words, validate), and generate code out of such action-semantics-enabled models. Unfortunately, the UML action semantics are defined in the context of a UML-only approach, paying no attention to other metamodels or UML profiles that already exist out there. As a consequence, platform specialists that have already provided platform descriptions through profiles, or through reusable metamodel packages, are expected now to provide model transformations to these semantically rich metamodels, for instance to perform validation in the case of action semantics. A standard metamodel for generating test cases can be imagined as well. Moreover, one should notice that the action semantics experience has inspired the latest OCL definition [23], which defines now a metamodel.

In the same spirit, several *semantically rich models* have been defined. For instance, when developing a UML model, which is supposed to be further on translated to the C language, access to the standard library and the data types of the C language should be provided as well. This is supported by additional UML model elements that are integrated into the models of the system under development, but only at the level of abstraction where the C language has been chosen as the programming language platform.

3.2. Component-Oriented Metamodeling

As presented in section 3.1, some techniques already exist for methodologists to reuse what others have previously defined. However, these techniques are not mature enough yet to provide methodologists with the same reusability power that component-oriented programming [28] has introduced to the software engineering world. We explore now how components can be used in defining MDE processes.

The most important characteristic of components is their interfaces, namely the `required` and `provided` interfaces. `Required` interfaces declare the contract of what the environment is supposed to make available to components. If another system part of the environment provides the realization of the `required` interfaces of a component, it enables the possibility to use that component, which is then able to realize the contract defined by its `provided` interfaces. As a consequence, any client system, including other components, providing realizations for the `required` interfaces of a component can use that component afterwards according to its `provided` interfaces.

Profiles have been called MDA components as well [29]. This seems true at a first impression, but the resemblance does not go that far. The `provided` interfaces can certainly be compared to the stereotypes and tags defined by profiles. `Required` interfaces can be compared to the extended metamodel. Component behavior can be compared to the endomorphic model transformation that applies a profile to concrete models. Unfortunately, such a modularity is not always achievable. For instance, the `required` interfaces, i.e., the extended metamodel, cannot always be realized as required by the different client models that will use the profile afterwards. An immediate consequence of this is that profiles developed for a given metamodel cannot work with another one, even if the two metamodels are comparable. For instance, all the numerous profiles defined for UML 1.4 will simply become obsolete once the UML 2.0 is delivered. One can argue that model transformations may solve this problem, but they cannot because the "profiled" model, i.e., the model resulted by applying the profile to a concrete model, will no longer conform to the new metamodel, which is certainly richer than the old metamodel that the profile actually extends. Stating this in another way, if one wants to benefit from the new facilities offered by UML 2.0, s/he cannot work with profiles extending the UML 1.4.

The adapter structural pattern [30] together with the MOF QVT may provide some help to solve the previously presented problem. Both the pattern and the QVT promote automatic data conversion wherever needed. In the case of profiles and metamodels, profiles should not extend well defined metamodels, but *virtual metamodels*, also referred to as *views*, that will be adapted later on to concrete metamodels. An example that suits very well the problem we are discussing is OCL 2.0 [23], which defines its own metamodel by extending (based on the package dependency mechanism) the UML 1.4 metamodel. The issue is now how to make OCL 2.0 work both with UML 2.0 and MOF 2.0, i.e., how to use OCL 2.0 both for specifying constraints in UML 2.0 models and for specifying "well-formedness" rules for MOF 2.0 defined metamodels (such as the UML 2.0 metamodel). Instead, the OCL metamodel could be viewed as a *generic profile*, or an MDA component, extending a *generic view* that could be adapted later on to all interesting metamodels, including the domain specific languages that methodologists might consider more suitable than UML or MOF for modeling different concerns. Following such an approach, an OCL interpreter, or code generator, could be applied to any model conforming to such adapted metamodels since it would depend only on the OCL generic profile. Moreover, another profile could extend or reuse the presumed OCL profile, by adapting its own extended view to the extended view of the OCL profile.

The same approach can be reused for model transformations as well, since model transformations depend on the metamodels of the models manipulated as input and output. A view approach, making transformations depend on views instead of metamodels, may be much easier to reuse by only adapting the views to concrete metamodels.

4. Conclusions and Future Work

Provided that accurate standards (as promised by MDA) and tools will be available, Model Driven Engineering will reduce significantly the work of system developers and maintainers. Nevertheless, it will be a challenge for methodologists to correctly and clearly define an MDE process. Moreover, the component-oriented modularization of the assets that methodologists should provide as part of an MDE process definition will play an important role in the reusability of (parts of) MDE processes by other methodologists. Each such MDE component should be easily comprehensible in order for the final MDE process to be understandable and then usable by system developers. This would allow a methodology to be tailored, or even built entirely from scratch, by simply assembling MDE components, with the final purpose of guiding the development and realization of a specific project.

In this position paper, we presented a class of possible MDE processes, and we identified some of the most important assets to be delivered by methodologists for fully specifying such MDE processes, taking the classical form of metamodels and model transformations. These assets are:

- the sequence of levels of abstraction,
- the modeling language to be used at each level of abstraction, including their possible (partial) representations in order to be understood by the different stakeholders of an MDE project,
- refinements that map these levels of abstraction between different modeling languages by relying on model transformations,
- mappings to already existing semantically rich modeling languages, for instance for validation or test cases generation.

Moreover, we proposed to slightly change the way platforms are described, taking advantage of the adapter GoF pattern. Platforms may still be described by profiles, but a hierarchy of profiles that adapt no longer a metamodel but a *view* of a metamodel is more recommended. The changes that such profiles inflict on the concrete syntaxes should be provided as well. Moreover, in order to be easily integrated in an MDE process, these profiles should be delivered with interactive transformations that act on the extended views, on one hand, and facilitate the process of applying the profiles to concrete models, on the other hand. The intent of having such transformations is to hide from developers the details of the different profiles. Developers are not required to know both the concrete and abstract syntaxes of a modeling language. Further more, they should not be required to know the content of a profile if a concrete syntax could solve the problem.

The last proposition is to take advantage of semantically rich components for MDE assets, using generic profiles and model transformations. Such a component declares a *view* that it can manipulate (i.e., `required` interfaces), perhaps a metamodel it is able to deliver based on this view (i.e., `provided` interfaces), and a set of behaviors it is able to offer. An example was considered, namely an OCL interpreter component, which declares a view and provides the OCL metamodel related to this view along with several features, such as to parse textual OCL constraints, provide type checking, interpret expressions, or find violating model elements. To become concrete, the declared view must be further adapted to UML 1.x, UML 2.0, MOF 1.4, MOF 2.0, etc. Moreover, just like normal components, MDE components can rely on and use other MDE components.

Acknowledgments

This work was partially supported by the Hasler Stiftung under grant number DICS-1850 and the INTERREG III TestUML project under grant number 14/AJ/9.1/1.

References

[1] J.-M. Jézéquel, “Model-driven engineering: Basic principles and challenges”, Invited Presentation at Formal Methods for Components and Objects (FMCO'03), Leiden, Netherlands, November 2003.

[2] S. Kent, “Model Driven Engineering”, Integrated Formal Methods: Third International Conference, IFM 2002, Turku, Finland, May 15-17, 2002. LNCS Vol. 2335, Springer-Verlag, 2003, pp. 286 – 298.

[3] R. Le Delliou, C. Mersier, “How to choose a software development method: a French company EDF’s experience report”, Proceedings of the Conference on Software Engineering Environments, Noordwijkerhout Netherlands, 5-7 April 1995.

[4] A. Seffah, J. Rilling, “Investigating the relationship between usability and conceptual gaps for human-centric CASE tools”, Proceedings of the Human-Centric Computing Languages and Environments Symposia, 5-7 September 2001.

[5] Object Management Group, `http://www.omg.org/`, September 2004.

[6] Object Management Group, *MDA Guide*, v1.0.1, omg/03-06-01, June 2003.

[7] Object Management Group, *UML 2.0 Infrastructure Final Adopted Specification*, ptc/03-09-15, September 2003.

[8] Object Management Group, *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*, ptc/03-10-04, October 2003.

[9] Object Management Group, *MOF 2.0 Query/Views/Transformations RFP*, ad/02-04-10, April 2002.

[10] Object Management Group, *XML Metadata Interchange (XMI)*, v2.0, formal/03-05-02, May 2003.

[11] Object Management Group, *MOF 2.0 Versioning and Development Lifecycle RFP*, ad/02-06-23, June 2002.

[12] Object Management Group, *MOF 2.0 Facility and Object Lifecycle RFP*, ad/03-01-35, January 2003.

[13] Object Management Group, *MOF Model to Text Transformation Language RFP*, ad/04-04-07, April 2004.

[14] J. Iivari, "Why are CASE Tools Not Used?", Communications of the ACM, Vol. 39, No. 10, October 1996.

[15] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, June 1992.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study" Technical report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, Carnegie Mellon University, November 1990.

[17] Object Management Group, *UML 2.0 Superstructure Final Adopted specification*, ptc/03-08-02, August 2003.

[18] R. Silaghi, F. Fondement, A. Strohmeier, "Towards an MDA-Oriented UML Profile for Distribution", Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Monterey, CA, USA, September 20-24, 2004. IEEE Computer Society, 2004, pp. 227 – 239. Also available as Technical Report, Nº IC/2004/49, Swiss Federal Institute of Technology in Lausanne, Switzerland, May 2004.

[19] M. Alanen, J. Lilius, I. Porres, D. Truscan, "Realizing a Model Driven Engineering Process", Technical Report No. 565, Turku Centre for Computer Science, November 2003.

[20] C. Atkinson, T. Kühne, "The Role of Metamodeling in MDA", International Workshop in Software Model Engineering (in conjunction with UML'02), Dresden, Germany, October 2002.

[21] S. Sendall, W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development", IEEE Software, Vol. 20, Issue 5, September 2003, pp. 42 – 45.

[22] J.-P. Almeida, R. Dijkman, M. van Sinderen, L. Ferreira Pires, "On the Notion of Abstract Platform in MDA Development", Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Monterey, CA, USA, September 20-24, 2004. IEEE Computer Society, 2004, pp. 253 – 263.

[23] Object Management Group, *UML 2.0 OCL Final Adopted specification*, ptc/03-10-14, October 2003.

[24] Rational Software Corporation, *UML Profile for EJB, JSR-000026*, Public Draft, May 2001.

[25] Object Management Group, *UML Profile for CORBA Specification*, v1.0, April 2002.

[26] A. Kleppe, J. Warmer, "Do MDA Transformations Preserve Meaning? An investigation into preserving semantics", MDA Workshop, York, UK, November 2003.

[27] S. Mellor, S. Tockey, R. Arthaud, P. Leblanc, *An Action Language for UML: Proposal for a Precise Execution Semantics*, UML 98, LNCS Vol. 1618, Springer-Verlag, 1998, pp. 307 – 318.

[28] C. Szyperski, *Component Software*, Addison-Wesley, December 1997.

[29] P. Desfray, "MDA – When a major software industry trend meets our toolset, implemented since 1994.", v1.2, Softeam whitepaper, May 2003.

[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*, Addison-Wesley, January 1995.