Workshop Proceedings

ChaMDE 2008

First International Workshop on Challenges in Model Driven Software Engineering

September 28th, 2008, Toulouse, France

Organized in conjunction with MoDELS'08
11th International Conference on Model Driven Engineering Languages and Systems

Edited by:
Stefan Van Baelen (K.U.Leuven, Belgium)
Ragnhild Van Der Straeten (VUB, Belgium)
Tom Mens (UMH, Belgium)

Table of Contents

Foreword

MoDELS'08 is already the eleventh conference on UML and Model-Driven Engineering, Languages and Systems. After more than a decade, research in MDE has significantly evolved.

Model-Driven Engineering is about creating, transforming, generating, interpreting, weaving models using modelling languages, tools, etc. After more than a decade of research in MDE, still a lot of fundamental and practical issues remain. Therefore this workshop addresses the question of how to proceed next and aims at identifying future challenges in MDE.

The objective of the workshop is to provide a forum for people from academia as well as industry people to:

- identify obstacles to MDE research and practice;
- facilitate transfer of research ideas to industry;
- propose "revolutionary" novel ideas; and
- proclaim important challenges that are either fundamental or pragmatic.

We received 15 submissions from 11 different countries, of which 11 papers were accepted. From these accepted papers, 7 papers have been selected to give a workshop presentation.

We hope that input from both research and practice will help to identify the future "grand challenges" in model-driven software engineering and discuss revolutionary ideas or new, original ways of thinking about MDE.

The ChaMDE 2008 organising committee,

Ragnhild Van Der Straeten,
Tom Mens,
Stefan Van Baelen,

September 2008.

Acknowledgments

The Organising Committee of ChaMDE 2008 would like to thank the workshop Program Committee for their helpful reviews.

Academic members:

- Jean Bézivin, Université de Nantes, France
- Xavier Blanc, LIP6, France
- Dirk Deridder, Vrije Universiteit Brussel, Belgium
- Gregor Engels, University of Paderborn, Germany
- Vincent Englebert, Facultés Universitaires de Notre Dame de la Paix, Belgium
- Robert France, Colorado State University, USA
- Dragan Gasevic, Athabasca University, Canada
- Sébastien Gérard, CEA-LIST, France
- Wouter Joosen, K.U.Leuven, Belgium
- Richard Paige, University of York, UK
- Ivan Porres, Åbo Akademi University, Finland
- Bernhard Rumpe, University of Braunschweig, Germany
- Hans Vangheluwe, McGill University, Canada

Industry Members:

- Anneke Kleppe, Capgemini Financial Service, The Netherlands
- Jochen Küster, IBM Zurich Research Laboratory, Switzerland
- Laurent Rioux, THALES Research and Technology, France

This workshop is organised as an event in the context of

- the MoVES IAP-Phase VI Interuniversity Attraction Poles Programme funded by the Belgian State, Belgian Science Policy
- the research project "Model-Driven Software Evolution", an "Action de Recherche Concertée" financed by the Ministère de la Communauté française – Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique
- the research project EUREKA-ITEA EVOLVE, partially funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders)

MDE Adoption in Industry: Challenges and Success Criteria

Parastoo Mohagheghi[1], Miguel A. Fernandez[2], Juan A. Martell[2], Mathias Fritzsche[3] and Wasif Gilani[3]

[1] SINTEF, P.O.Box 124-Blindern, N-0314 Oslo, Norway
parastoo.mohagheghi@sintef.no
[2] Telefónica Research & Development, Valladolid, Spain
mafg@tid.es, jamartell@gfi-info.com
[3] SAP Research CEC Belfast, United Kingdom
{mathias.fritzsche, wasif.gilani}@sap.com

Abstract. Model-Driven Engineering has been promoted for some time as the solution for the main problem software industry is facing, i.e. complexity of software development, by raising the abstraction level and introducing more automation in the process. The promises are many; among them improved software quality by increased traceability between artifacts, early defect detection, reducing manual and error-prone work and including knowledge in generators. However, in our opinion MDE is still in the early adoption phase and to be successfully adopted by industry, it must prove its superiority over other development paradigms and be supported by a rich ecosystem of stable, compatible and standardized tools. It should also not introduce more complexity than it removes. The subject of this paper is the challenges in MDE adoption from our experience of using MDE in real and research projects, where MDE has potential for success and what the key success criteria are.

Keywords: Model-driven engineering, challenges, domain-specific modeling, performance engineering, traceability.

1 Introduction

Today's software systems are complex in nature; the size has been growing because of the increased functionality, heterogeneity is also becoming a bigger concern as systems are built from several systems or include legacy code, systems are distributed over multiple sites and there are new requirements such as dynamicity and autonomy (self-* properties, for example self-healing). Handling each of these challenges requires specific approaches which often include domain-specific knowledge and solutions. However, based on the experience gained from multiple domains and projects, some solutions may be identified as beneficial to complex software development in general.

Model-Driven Engineering (MDE) is an approach built upon many of the successful techniques applied in software engineering: It can be characterized by: a) raising the abstraction level by hiding platform-specific details ; b) taking advantage of models in all the phases of software development to improve understanding; c) developing specific languages and frameworks to achieve domain appropriateness; and d) taking advantage of transformations to automate repetitive work and improve software quality [1]. These are all techniques useful for complex system development and therefore one may expect rapid adoption of the paradigm by industry. So far, we cannot see such wide adoption, as also confirmed by a review of industrial experiences presented in [2]. To be accepted by the majority, the industry must gain confidence on the promises of MDE and have access to proper tools and experts.

The European research projects MODELWARE[1] and its continuation MODELPLEX[2] have focused on MDE approaches and tools with the goal of making them suitable for complex system development. Some of the companies involved in these projects have experience from applying MDE in real projects while others think that MDE is not yet mature enough to be taken from research projects to industry production. This paper therefore elaborates on where we can expect added value from MDE and what the barriers are from experiences gained in the context of these projects. In the remainder of this paper we discuss industry expectations and experience in Sections 2 and 3 and conclude our discussion in Section 4.

2 SAP Experience

SAP has already started working towards applying MDE concepts, and currently employs models in various stages of business application development. The Composition Environment is one example where MDE concepts are applied for efficient development of Composite Applications. Composite Applications are self-contained applications that combine loosely coupled services (including third party services) with their own business logic, and thereby provide user centric front-end processes that transcend functional boundaries, and are completely independent from the underlying architecture, implementation and software lifecycle. With Composition Environment even the non-technical users, such as business domain experts, consultants, etc., having no programming skills, are able to model and deploy customized applications suited to their specific business requirements.

Based on our experiences with the currently employed tools for MDE of business processes, such as the Composition Environment, we identified the general need of supporting non-technical users with regards to non-functional requirements, such as the impact of their design decisions on performance, etc. Within the context of performance engineering, for instance, such a support means guidance towards better design / configuration that actually meets the timelines, and optimized resource mapping against each activity in the business process.

We implemented such performance related decision support as an extension of MDE. By implementing this extension, named Model-Driven Performance Engineering (MDPE), we realized the need for supporting requirements with respect to non-functional aspects, especially performance. The implementation of MDPE heavily uses the MDE concepts such as meta-modeling, transformations, model

[1] http://www.modelware-ist.org/
[2] http://www.modelplex-ist.org/

weaving and mega-modeling. For instance, ten different meta-modeling languages are employed in order to make the process usable for a number of domain-specific modeling languages. During the implementation of MDPE, we recognized that the application of MDE concepts enabled us to focus on the creative tasks of development rather than repetitive coding. For instance, code generation for our meta-models saved us significant development effort. The only place where a significant amount of coding effort was required was for the integration of MDPE into the existing tool infrastructure.

Meta-model extension is the generally employed technique for model annotations, such as done with profiles in the case of UML [4]. However, this is not applicable while dealing with the proprietary models. The application of model weaving enabled us a high degree of flexibility as we are able to annotate any kind of proprietary model with the help of a generic editor [4]. Higher-order transformations are used to enable traceability in our approach [5]. Additionally, mega-modeling enables us to locate our model artifacts, such as the tracing models related to the models in our transformation chain [6].

As for the challenges, we experienced that MDE concepts are on the one hand very systematic and efficient, but on the other hand also difficult to understand for developers as they require quite a high level of abstraction and training. Also, the MDE tool support is sometimes not mature enough. Especially the available tooling to define model transformation chains lacks capabilities of modern IDEs, which could decrease the development time for model transformations significantly.

Concluding, based on the experiences gained with the development of MDPE, we are optimistic regarding the capabilities of MDE in case the tool support improves, and the MDE community meets the challenges associated with the MDE process, such as providing support for dealing with non-functional aspects of system development.

3 Telefónica Experience

In [3], we have discussed the experience of Telefónica in moving from a code-centric to a model-centric software development. Earlier efforts in modeling failed due to the complexity of UML, the lack of proper tools and the inability to maintain models in synch with code, among other issues. Due to the above problems with UML, we decided to develop our own programming tools and frameworks addressing the problem domain. But without any industry standards to rely on, this approach had no future in the long term and was also difficult to use for non-technical stuff, such as telecom domain experts, as it did not have the required abstraction level.

This was an experience from eight years ago, but not so many things seem to have fundamentally changed. What we look for is a domain-specific modeling (DSM) language integrated in a development environment that will permit the modeling of our basic domain concepts, such as interfaces, devices, networks, protocols and services. We also emphasize adhering to current industry standards in the domain, since we now look for a domain-specific solution, not a company-wide solution. Other requirements are: a) the ability to model in multiple abstraction levels, hiding details as desired; b) the integration of model verification tools based on OCL or other constraint languages and c) the composition / weaving of the models at run time to reflect the changes in the network's operational status.

In the road toward these objectives we foresee numerous challenges. First of all, the UML standard has evolved but, with this evolution, the syntax has become even more complex and the necessary supporting mechanisms and tools for dealing with this added complexity are not yet available. Even something as conceptually simple as exporting a UML diagram from one tool to another has not been accomplished yet with ease. On the other hand, developing a DSM solution requires high skills related to meta-modeling and tool development. Also a big concern with Domain-Specific Languages (DSLs) is getting the people in that domain to agree upon a standard syntax. Another challenge is having that DSL interact properly with anything outside of its domain, having a different underlying syntax to that of other languages.

Model synchronization (for example applying multiple profiles to a source model) and roundtrip engineering are yet to be addressed successfully and mechanisms for dealing with very large and complex models, such as hierarchical models, traceability and model management in general are also in an inception phase right now, at least regarding to the aspect of tool support. All these features are important to make a full fledged MDE process work in complex, real-life projects.

Another challenge for organizations wanting to get started in MDE, closely related with the previous idea of managing all these artifacts, is that they may end up dealing with more complexity than anticipated at first. The underlying problem here is: are the techniques for handling complexity in danger of making the software engineering process itself too complex? To adequately address complexity we have to substitute it for something simpler not for something different but equally complex.

It is our opinion also that there are some basic milestones a new technology has to go through for it to be considered mainstream. To start with, we need a proper context for it to flourish and be nurtured in. The fabric of this context is made of the proper professionals with the proper knowledge and expertise and supporting material which helps in turn to create these professionals. This has to be accompanied by the development of high-quality literature, tutorials and proper material to draw new professionals in.

The main question that an organization has to ask itself is "do I really need MDE?" The second question relates with its ability to adapt its processes to the ones needed from an MDE point of view (partially discussed also in [3]), adapt their staff to new ways of looking at problems and create new layers of software development supporting all the aspects MDE has to offer. Companies may be reluctant to change either their structure or part of it. Apart from software factories for product line engineering (PLE) we have not identified very good candidates for MDE to be applied to.

4 Conclusions

MDE is a long-term investment and needs customization of environment, tools and processes, and training. For companies that have a product line, MDE can pay off since this cost is amortized over several projects. For one-of–a-kind projects this will not pay in most cases. Despite differences in domain and the type of systems developed in the two companies, there are common challenges as described here. The most important one is the complexity of developing an MDE environment tailored to the company needs. This environment requires:

- Developing proper languages for communication between technical and non-technical experts and for modeling various aspects. The major challenge here is to have the required language engineering expertise since creating own profiles or meta-models are difficult and for complex systems we probably need several languages. Hence more domain-specific meta-models and profiles are needed that are supported by tools and may be reused. The current tools for developing meta-models and editors are not user friendly, the learning curve is steep and the documentation and support is not satisfactory.
- Several tools are required for modeling, model-to-model and model-to-text transformation, verification and simulation, and other tools to store, reuse and compose models. There is no tool chain at the moment and companies must integrate several tools and perform adaptation themselves.

Both of the above requirements put a high burden on companies that traditionally used third-party tools for modeling and performed programming by hand. Training is another major challenge here. We see advantages in gradual introduction and support by management, as well as in the creation of teams of experts that can give support and create the necessary tools for MDE adoption in the whole company.

Acknowledgments. Part of the ideas presented in this paper are based on conclusions obtained in the MODELPLEX project (IST-FP6-2006 Contract No. 34081), co-funded by the European Commission as part of the 6th Framework Program.

References

1. Mohagheghi, P.: Evaluating Software Development Methodologies based on their Practices and Promises. Accepted at the 7th Int'l Conference on Software Methodologies, Tools and Techniques (Somet'08)
2. Mohagheghi, P., Dehlen, V.: Where is the Proof? A Review of Experiences from Applying MDE in Industry. In ECMDA-FA 2008, LNCS 5095, Springer, pp. 432—443 (2008)
3. Fernandez, M.: From Code to Models: Past, Present and Future of MDE Adoption in Telefónica. In: 3rd Europen Workshop From Code Centric to Model Centric Software Engineering: Practices, Implications and Return on Investment (C2M), co-located with ECMDA 2008, pp. 41—51 (2008)
4. Fritzsche M., Johannes J., et al: Systematic Usage of Embedded Modelling Languages in Model Transformation Chains. Accepted at the Software Language Engineering Conference (SLE'08)
5. Fritzsche, M., Johannes, J., Zschaler, S., Zherebtsov, A., Terekhov, A.: Application of Tracing Techniques in Model-Driven Performance Engineering. In: ECMDA-FA 4th Workshop on Traceability (2008)
6. Barbero, F. Jouault, J. Bezivin: Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In: 15th ECBS'08, IEEE Press, pp. 277--286 (2008)

Scalability: The Holy Grail of Model Driven Engineering

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
(dkolovos,paige,fiona)@cs.york.ac.uk

Abstract. Scalability is a desirable property in Model-Driven Engineering (MDE). The current focus of research in MDE is on declarative languages for model management, and scalable mechanisms for persisting models (e.g., using databases). In this paper we claim that, instead, modularity and encapsulation in modelling languages should be the main focus. We justify this claim by demonstrating how those two principles apply to a related domain – code development – where the issue of scalability has been addressed to a much greater extent than in MDE.

1 Introduction

The adoption of MDE technologies in an industrial context involves significant benefits but also substantial risks. Benefits in terms of (eventual) productivity, quality and reuse are today foreseeable. On the other hand, the most important concerns raised of MDE are those of scalability [1], the cost of introducing MDE technologies to the development process (training, learning curve) and longevity of MDE tools and languages. To our perception, the latter two concerns (cost of induction and longevity) are not preventive for the adoption of MDE; however scalability is what is holding back a number of potential adopters.

2 Scalability in MDE

Large companies typically develop complex systems that require proportionally large and complex models that form the basis of representation and reasoning. Moreover, development is typically carried out in a distributed context and involves many developers with different roles and responsibilities. In this context, typical exploratory questions from industrial parties interested in adopting MDE include the following:

1. *In our company we have huge models, of the order of tens of thousands of model elements. Can your tool/language support such models?*
2. *I would like to use model transformation. However, when I make a small change in my (huge) source model, it is important that only this change to be propagated to the target model; I don't want the entire target model to be regenerated.*

3. (similarly) *I would like to use code generation. However, when I make a small change in my (huge) model I don't want all the code to be regenerated.*
4. *In my company we have many developers and each manages only a specific part of the model. I would like each developer to be able to check out only a part of the model, edit it locally and then merge the changes into the master copy. The system should also let the developers know if their changes are in conflict with the rest of the model or with changes done by other developers.*

Instead of attempting to answer such questions directly, we find it useful to consider analogies with a proven and widely used environment that addresses those problems in a different – but highly relevant – domain. The domain is code development and the environment is the well known and widely used Eclipse Java Development Tools (JDT).

As a brief overview, JDT provide an environment in which developers can manage huge code-bases consisting of (tens of) thousands of Java source code files (concern 1). JDT supports incremental consistency checking and compilation (concerns 2,3) in the sense that when a developer changes the source code of a particular Java class, only that class and any other classes affected by the change – as opposed to all the classes in the project or the workspace – are re-checked and re-compiled. Finally, JDT is orthogonal with version control, collaborative development (concern 4), and multi-tasking tools such as CVS and SVN and Mylyn.

3 Managing Volume Increase

As models grow, tools that manage them, such as editors and transformation engines, must scale proportionally. A common concern often raised is that modelling frameworks such as EMF and widely-used model management languages do not scale beyond a few tens of thousands of model elements per model. While this is a valid concern, it is also worth mentioning that the Java compiler does not allow source-code files that exceed 64 KB, but in the code-development domain this is rarely a problem.

The reason for this asymmetry in perception is that in code development, including all the code of an application in a single file is considered – at least – bad practice. By contrast, in modelling it is deemed reasonable to store a model that contains thousands of elements in a single file. Also, it is reasonable that any part of the model can be hard-linked with an ID-based reference to any other part of the model.

To deal with the growing size of models and their applications, modelling frameworks such as EMF [2] support lazy loading and there are even approaches, such as Teneo [3], for persisting models in databases. Although useful in practice, such approaches appear to be temporary workarounds that attempt to compensate for the lack of encapsulation and modularity constructs in modelling languages. In our view, the issue to be addressed in the long run is not how to manage large monolithic models but how to separate them into smaller modular and reusable models according to the well understood principles defined almost 40 years ago in [4], and similarly to the practices followed in code development.

4 Incrementality

In the MDE research community, incrementality in model management is sought mainly by means of purely declarative model transformation approaches [5, 6]. The hypothesis is that a purely declarative transformation can be analysed automatically to determine the effects of a change in the source model to the target model. Experience has demonstrated that incremental transformations are indeed possible but their application is limited to scenarios where the source and target languages are similar, and the transformation does not involve complex calculations.

JDT achieves incrementality without using a declarative language for compiling Java source to bytecode; instead it uses Java which is an imperative language. The reason JDT can achieve incremental transformation lies mainly within Java itself. Unlike the majority of modelling languages, Java has a set of well-defined modularity and encapsulation rules that, in most cases, prevent changes from introducing extensive ripple effects.

But how does JDT know what is the scope of each change? The answer is simple: it is hard-coded. However, due to the modular design of the language, those cases are relatively few and the benefits delivered justify the choice to hard-code them. Also it is worth noting that the scope of the effect caused by a change is not related only to the change and the language but also to the intention of the transformation. For example, if instead of compiling the Java source code to bytecode we needed to generate a single HTML page that contained the current names of all the classes we would unavoidably need to re-visit all the classes (or use cached values obtained earlier).

5 Collaborative Development

As discussed in Section 2, a requirement is to enable collaborative development of models. A common requirement is that each developer should be able to check out a part of the model, modify it and then commit the changes back to the master copy/repository. Again, the formulation of this requirement is driven by the current status which typically involves constructing and working with large monolithic models. With enhanced modularity and encapsulation, big models can be separated into smaller models which can then be managed using robust existing collaborative development tools such as CVS and SVN, augmented with model-specific version comparison and merging utilities such as EMF Compare [7]. Given the criticality of version control systems in the business context, business users are particularly reluctant to switching to a new version control system[1]. Therefore, our view is that radically different solutions, such as dedicated model repositories, that do not build on an existing robust and proven basis are highly unlikely to be used in practice.

6 Modularity in Modelling Languages

All the above clearly demonstrate the importance of modularity and encapsulation for achieving scalability in MDE. There are two aspects related to modularity in modelling: the design of the modelling language(s) used and the capabilities offered by the underlying modelling framework. In this section we briefly discuss how each of those aspects affect modularity and envision desirable capabilities of modelling frameworks towards this direction.

6.1 Language Design

With the advent of technologies such as EMF and GMF, implementing a new domain-specific modelling language and supporting graphical editors is a straightforward process and many individuals and organizations have started defining custom modelling languages to harvest the advantages of the context-specific focus of DSLs. When designing a new modelling language, modularity must be a principal concern. The designers of the language must ensure that large models captured using the DSL can be separated into smaller models by providing appropriate model element *packaging* constructs. Such constructs may not be part of the domain and therefore they are not easily foreseeable. For example, when designing a DSL for modelling relational databases, it is quite common to neglect *packaging*, because relational databases are typically a flat list of tables. However, when using the language to design a database with hundreds of tables, being able to group them in conceptually coherent packages is highly important to the manageability and understandability of the model.

6.2 Modelling Framework Capabilities

In contemporary modelling frameworks there are three ways to capture relationships between two elements in a model: containment, hard references and soft references. Containment is the natural relationship of one element being a composite part of another, a hard reference is a unique-ID-based reference that can be resolved automatically by the modelling framework and a soft reference is a reference that needs an explicit resolution algorithm to navigate [8].

To enable users to split models over multiple physical files, contemporary modelling frameworks support cross-model containment (i.e., the ability of a model element to contain another despite being stored in different physical files).

[1] Evidence of this is that CVS which was introduced in the 1980s is still the most popular version control system despite its obvious limitations compared to newer systems such as SVN

With regard to hard and soft non-containment references, hard references are typically proffered because they can be automatically resolved by the modelling framework and thus, they enable smooth navigation over the elements of the model with languages such as OCL and Java. Nevertheless, in our view hard references are particularly harmful for modularity as they increase coupling between different parts of the model and prevent users from working independently on different parts. On the other hand, soft references enable clean separation of model fragments but require custom resolution algorithms which have to be implemented from scratch each time.

To address this problem, we envision extensions of contemporary modelling frameworks that will be able to integrate resolution algorithms so that soft references can be used, and the efficient and concise navigation achievable with languages such as OCL can still be performed.

7 Conclusions

In this paper we have demonstrated the importance of modularity and encapsulation for achieving scalability in MDE. We have identified two main problems: neglect of modularity constructs during the design of modelling languages and extensive use of ID-based references that lead to high coupling between different parts of the model. With regard to the first issue we have been working on preparing a set of guidelines for the design of scalable and modular DSLs and expect to report on this soon. The second issue is quite more complex and we plan to elaborate and prototype a solution based on EMF in the near future.

References

1. Jos Warmer, Anneke Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *Proc. 6th OOPSLA Workshop on Domain-Specific Modeling*, Portland, Oregon, USA, October 2006.
2. Eclipse Foundation. Eclipse Modelling Framework. http://www.eclipse.org/emf.
3. Eclipse Foundation. Teneo, 2008. http://www.eclipse.org/modeling/emft/?project=teneo.
4. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of ACM*, 15(12):1053–1058, 1972.
5. David Hearnden, Michael Lawley, Kerry Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *Proc. Model Driven Engineering Languages and Systems*, pages 321–335.
6. Holger Giese, Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, pages 1619–1374, March 2008.
7. Eclipse Foundation. EMF Compare, 2008. http://www.eclipse.org/modeling/emft/?project=compare.
8. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. Detecting and Repairing Inconsistencies Across Heterogeneous Models. In *Proc. 1st IEEE International Conference on Software Testing, Verification and Validation*, pages 356–364, Lillehammer, Norway, April 2008.

A foundation for MDE

Ernesto Posse[1] and Juergen Dingel[1]

[1]Applied Formal Methods Group
Software Technology Lab
School of Computing
Queen's University
Kingston, Ontario, Canada
{dingel,eposse}@cs.queensu.ca

Abstract. The central idea of Model-Driven Engineering (MDE) is to use models as the main artifact in the process of software development. One of the "grand visions" of MDE is to leverage models at every stage of the development process, from requirements to design, implementation and maintenance. While great advances have been made, MDE is still lacking adoption by developers. We believe that for MDE to live up to its full potential it must rest on a solid foundation, and therefore, one of the main challenges facing MDE today is the establishment of such foundation. In order to illustrate what can be achieved, what is missing and what kind of issues must be addressed by a successful approach to MDE, we take a look at UML-RT.

1 Introduction

Software Engineering has as purpose making the process of software development an engineering discipline. All engineering fields rely on the disciplined use of models to reach their objectives. From sketches to blueprints and detailed mathematical equations, models play a central role in capturing requirements and design. They allow engineers to reduce risks and costs by dealing with fundamental decisions in the early stages of development. They serve as guides for implementation. Furthermore, they also serve as a useful mechanism for communication between clients, engineers, implementers, etc.

The benefits of models are apparent throughout the engineering spectrum. Taking a cue from other engineering disciplines, Model-Driven Software Engineering proposes the use of models in software development. Nevertheless, this vision of the use of models throughout the software development process is not yet a reality, in spite of advances in the field. MDE does not yet enjoy widespread adoption. Why is this the case? There are many different reasons concerning both "cultural" and technical issues. Amongst many software developers there is still a culture of coding: to them code, rather than models, are the real object of development. Furthermore, there is lack of training and education in MDE methods both in academia and industry. On the more technical side, the applications of MDE sometimes seem *ad hoc*, brittle and unscalable to the current and future demands on software.

How can we deal with these issues? There is no obvious "silver bullet," and therefore a wise strategy would be to attack the problem from all possible sides. There is a need for more publicly documented success stories of MDE. There is a need for more training programs in MDE, for relevant textbooks with realistic examples and tools. There is a need for improvement in modelling languages. There is a need for better tools that help automate the use of models, including modelling editors, code generators, simulators and analysis tools such as model checkers. But for these tools and development processes to be more generally beneficial and less *ad hoc*, they need to be cemented on a solid foundation, and therefore we need a better, deeper understanding of the underlying theory of modelling. Hence, the challenge is to find a proper foundation for MDE that supports the process of software development in general, and to support the analysis of models and the creation of tools in particular.

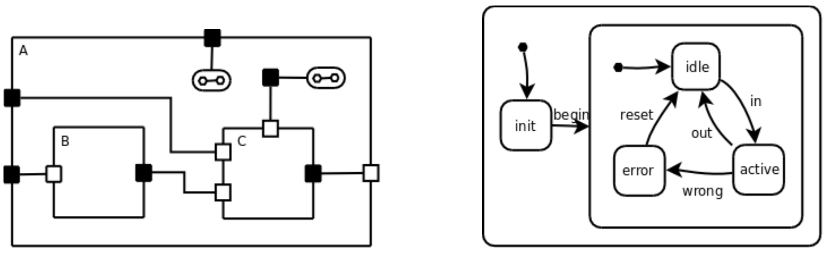
Modelling tools depend, of course, on the modelling languages being used. To be able to do any non-trivial manipulation or analysis on models, their modelling languages need to have a rich and expressive, yet simple and manageable set of linguistic features and constructs (*abstract syntax*) with a usable notation (*concrete syntax*) and well-defined meaning (*semantics*) which can guarantee strong properties that aid in the analysis of models. Hence the problems of identifying effective syntax and semantics and finding suitable ways to describe them, are at the very core of the foundations of modelling. Furthermore, it is not enough to define the meaning of a language's constructs to be able to use it effectively. It is also fundamental to have an understanding of which language patterns are better or desirable (for maintainability, performance, etc.,) as well as an understanding of best practices. Therefore, in addition to syntax and semantics, a foundation for MDE should also cover *pragmatics*, much in the same way as in the theory of programming languages and linguistics in general. Syntax, semantics and pragmatics provide a basis for reasoning about models.

In order to illustrate the need for research in foundations of MDE, and to suggest some approaches to this challenge we look at one of the more successful realizations of MDE: UML-RT/Rational RoseRT.

2 An example: UML-RT/RoseRT

UML-RT [9] is a UML profile intended for the domain of distributed, embedded real-time systems that evolved from the ROOM modeling language and methodology [8]. Rational Rose Technical Developer (formerly RoseRT) is a development environment for UML-RT developed by Rational Software Corporation (now part of IBM.) A model in UML-RT consists of a hierarchical specification of components called *capsules* which contain behavioural specifications in the form of State Machines, and which can interact between them through *connectors*[1]. Figure 1 shows an example of a UML-RT model. Figure 1 (a) shows a structural view of the model consisting of a capsule A with capsules B and C

[1] The term "connector" as used here can be seen as an example of "connector" as it is sometimes used in software architecture [10].



(a) Structure: a hierarchical capsule. (b) Behaviour: a State Machine.

Fig. 1. An example of a UML-RT model.

contained in A, and linked through connectors. Some ports have associated State Machines, a simplified form of Statecharts [3], as shown in Figure 1 (b).

UML-RT provides a set of features which is rich enough, and expressive enough to model large systems and manage their complexity. For example, it enforces strong encapsulation by making interaction through ports and connectors the only mechanism to change a capsule's state[2]. Such strong encapsulation enables compositional designs and therefore it helps tackling complexity. Other examples include simplified State Machines without orthogonal regions, which are a source of many complications, and a simple run-to-completion semantics, whereby the system reaches a stable state before processing the next event. These features give models strong properties which enable a clear conceptual understanding necessary for reasoning about systems. Furthermore, they seem to provide effective control of complexities of development.

Aside from a strong set of linguistic features, UML-RT enjoys a reasonably simple notation and syntax which makes diagrams understandable despite the complexity of models (*e.g.*, hierarchical capsules with ports, hierarchical states and group transitions, etc.)

Furthermore, it enjoys good tool support as provided by RoseRT, a tool that provides code generation, some support for analysis of models (*e.g.,* supporting a conservative approach to static deadlock analysis,) and respects the needs of developers (*e.g.*, by allowing incomplete or non-wellformed models during active development.)

In summary, UML-RT seems to be a well-designed language with a set of syntactic and semantic features as well as reasoning support that make the use of models worth-while, this is, the benefits (*e.g.*, substantially increased productivity) outweigh the costs. All these characteristics seem to explain, at least in part, the success of UML-RT in its domain.

Nevertheless, not everything about UML-RT is perfect, and there is much room for improvement, in terms of features, tool support and formal semantics. For example, with respect to features, there is a pressing need for an action language [5] to better align modelling with the execution environment (*e.g.*, by

[2] cf. [10] urges to make connectors first-class in software architecture.

preventing harmful use of actions in State Machines, and ensure they conform with the semantics.) There is also a need for behavioural interface specifications (*e.g.*, as given by Protocol State Machines,) to improve support for modular design to deal with the intricacies of software composition by imposing constraints on component interactions (*e.g.*, specifying ordering of events.) Another example is the lack of direct support, in spite of its name, for real-time specifications and schedulability analysis. These are just a few examples to highlight the need for research of features and capabilities that could further enhance the effectiveness of UML-RT and RoseRT, ultimately improving the development process.

On the tooling side there is also need for much research. For instance, the realization of certain language features by simulation or code generation introduces complex issues (*e.g.*, how to guarantee run-to-completion semantics in the presence of multi-threading, or how to implement real-time.) There is also need for better support for execution and analysis, for example in the form of debugging, model animation and execution traces.

With respect to formal semantics of UML-RT there has been some promising research (*e.g.*, [2,4,7,11],) but there does not seem to be a generally accepted formal foundation yet. There is a need for a better understanding of the role of theories that have been proposed, and therefore we need to compare and relate them, and we need to explore their applicability, to see how they aid us in developing and reasoning about systems.

3 Conclusions

A well-designed and effective modelling language(s) is a necessary prerequisite for an MDE approach to be successful. The current state of the art does not yet provide the kind of understanding required. Just as the applications of MDE appear *ad hoc*, the design of modelling languages often does too. Hence, more research into modeling language design is needed, covering syntax, semantics, pragmatics, and reasoning. In particular, when designing modelling languages, or examining existing ones, we need to:

– identify core features central to modelling in the intended domain: consider the case of UML 2, a language with many features, often criticized for suffering a language bloat problem; there is limited understanding of how all such features interact with each other and as a consequence, only a subset is used in practice (*e.g.*, associations in class diagrams [1] or actions in activity diagrams); this highlights the importance of efforts such as the fUML proposal which attempts to find a manageable, executable subset of the UML [6],
– clarify their semantics and possible properties and trade-offs: consider again the case of UML 2 and in particular, activity diagrams whose token-offer semantics involve intricate deadlock avoidance rules; like the rest of the UML, there is no formal semantics yet for this, but it needs to be clarified as activities play a central role in UML 2,
– clarify the impact of model transformations on relevant properties of models: as models go through a complex life cycle and are manipulated by transfor-

mations, it becomes essential that these transformations preserve the model's semantics and any property of interest (*e.g.*, safety, liveness, security, etc,)
- study language patterns: how are constructs better applied and combined? language patterns might yield insight on the language's features and provide guidance for design.

We believe that process algebras could help solving these issues and provide an adequate foundation. Nevertheless, much work needs to be done, in particular to close the gap between the theory of process algebras and the practice of MDE.

Other challenges to discuss at the workshop
1. Multi-formalism modelling: the semantics and pragmatics of models with different components or views described in different Domain Specific Languages.
2. Optimized code generation: what kind of optimizations can be applied to models and code generated from them.
3. Training, teaching and dissemination of MDE methods.

Acknowledgments We would like to thank Bran Selic and Major Ron Smith for their insights into UML-RT, Rational RoseRT and MDE in general.

References

1. Z. Diskin, S. Easterbrook, and Juergen Dingel. Engineering associations: from models to code through semantics. In *TOOLS-EUROPE 2008, 46th International Conference Objects, Models, Components, Patterns*, Lecture Notes in Business and Information Processing. Springer, 2008.
2. R. Grosu, M. Broy, B. Selic, and G. Stefanescu. *Behavioral Specifications of Businesses and Systems*, chapter 6: What is behind UML-RT?, pages 73–88. Kluwer Academic Publishers, 1999.
3. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.
4. S. Leue, A. Stefanescu, and W. Wei. An AsmL Semantics for Dynamic Structures and Run-Tim Schedulability in UML-RT. In *Proceedings of the 46th International Conference on Objects, Models, Components, Patterns (TOOLS'08)*, Lecture Notes in Business and Information Processing, 2008. To appear.
5. Object Management Group. Concrete Syntax for a UML Action Language. http://www.omg.org/docs/ad/07-08-02.pdf, 2007.
6. Object Management Group. Semantics of a Foundational Subset for Executable UML Models. http://www.omg.org/docs/ad/08-05-02.pdf, 2008.
7. S. Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams.* Ph.d. thesis, Universität Ulm, 2006.
8. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object Oriented Modeling.* Wiley & Sons, 1994.
9. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. http://www.objectime.com/uml, 1998.
10. M. Shaw and Garlan. D. *Software Architecture: Perspectives on an emerging discipline.* Prentice Hall, 1996.
11. M. von der Beeck. A formal semantics of UML-RT. In *Proceedings of Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006*, pages 768–782, 2006.

Behavior, Time and Viewpoint Consistency: Three Challenges for MDE

José Eduardo Rivera[1], José Raul Romero[2], and Antonio Vallecillo[1]

[1]Universidad de Málaga (Spain)
[2]Universidad de Córdoba (Spain)
{rivera,av}@lcc.uma.es, jrromero@uco.es

Abstract. Although Model Driven Software Development (MDSD) is achieving significant progress, it is still far from becoming a real Engineering discipline. In fact, many of the difficult problems of the engineering of complex software systems are still unresolved, or simplistically addressed by many of the current MDSD approaches. In this position paper we outline three of the outstanding problems that we think MDSD should tackle in order to be useful in industrial environments.

1 Introduction

Although both MDSD and MDA have experienced significant advances during the past 8 years, some of the key difficult issues still remain unresolved. In fact, the number of engineering practices and tools that have been developed for the industrial design, implementation and maintenance of large-scale, enterprise-wide software systems is still low — i.e. there are very few real *Model-Driven Engineering* (MDE) practices and tools. Firstly, many of the MDSD processes, notations and tools fall apart when dealing with large-scale systems composed of hundred of thousands of highly interconnected elements; secondly, MDE should go beyond conceptual modeling and generative programming: it should count on mature tool-support for automating the design, development and analysis of systems, as well as on measurable engineering processes and methodologies to drive the effective use of all these artifacts towards the predictable construction of software systems.In particular, engineering activities such as simulation, analysis, validation, quality evaluation, etc., should be fully supported.

We are currently in a situation where the industry is interested in MDE, but we can easily fail again if we do not deliver (promptly) anything really useful to them. There are still many challenges ahead, which we should soon address in order not to lose the current momentum of MDE.

In this position paper we focus on three of these challenges. Firstly, the specification of the behavioral semantics of metamodels (beyond their basic structure), so that different kinds of analysis can be conducted, e.g., simulation, validation and model checking. A second challenge is the support of the notion of time in these behavioral descriptions, another key issue to allow industrial systems to be realistically simulated and properly analyzed — to be able to conduct, e.g., performance and reliability analysis. Finally, we need not only to tackle the *accidental* complexity involved building software systems, but we should also try to deal with their *essential* complexity. In this sense, the effective use of independent but complementary viewpoints to model large-scale systems, and the specification of correspondences between them to reason about the consistency of the global specifications, is the third of our identified challenges.

2 Adding Behavioral Semantics to DSLs

Domain Specific Languages (DSLs) are usually defined only by their abstract and concrete syntaxes. The abstract syntax of a DSL is normally specified by a metamodel, which describes the concepts of the language, the relationships among them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules.

The concrete syntax of a DSL provides a realization of the abstract syntax of a metamodel as a mapping between the metamodel concepts and their textual or graphical representation (see Fig. 1). A language can have several concrete syntaxes. For visual languages, it is necessary to establish links between these concepts and the visual symbols that represent them — as done, e.g, with GMF. Similarly, with textual languages links are required between metamodel elements and the syntactic structures of the textual DSL.

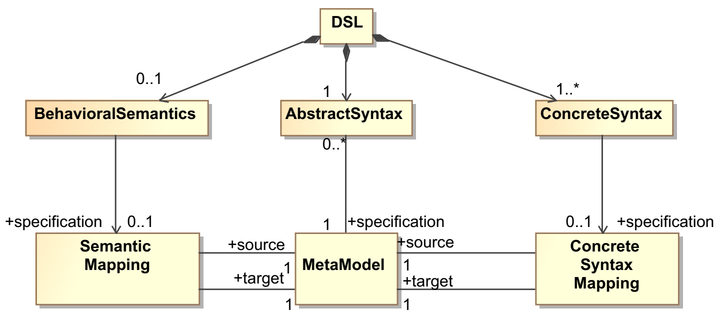


Fig. 1. Specification of a Domain Specific Language

Current DSM approaches have mainly focused on the structural aspects of DSLs. Explicit and formal specification of a model semantics has not received much attention by the DSM community until recently, despite the fact that this creates a possibility for semantic mismatch between design models and modeling languages of analysis tools [1]. While this problem exists in virtually every domain where DSLs are used, it is more common in domains in which behavior needs to be explicitly represented, as it happens in most industrial applications of a certain complexity. This issue is particularly important in safety-critical real-time and embedded system domains, where precision is required and where semantic ambiguities may produce conflicting results across different tools. Furthermore, the lack of explicit behavioral semantics strongly hampers the development of formal analysis and simulation tools, relegating models to their current common role of simple illustrations.

The definition of the semantics of a language can be accomplished through the definition of a mapping between the language itself and another language with well-defined semantics (see Fig. 1). These *semantic mappings* [2] are very useful not only to provide precise semantics to DSLs, but also to be able to simulate, analyze or reason about them using the logical and semantical framework available in the target domain. In our opinion, in MDE these mappings can be defined in terms of model transformations.

Describing Dynamic Behavior. There are several ways for specifying the dynamic behavior of a DSL, from textual to graphical. We can find approaches that make use of, e.g., UML diagrams, rewrite logic, action languages or Abstract State Machines [3] for this aim. One particular way is by describing the evolution of the state of the modeled artifacts along some time model. In MDE, model transformation languages that support in-place update [4] can be perfect candidates for the job. These languages are composed of rules that prescribe the preconditions of the actions to be triggered and the effects of such actions. Furthermore, if these transformations use concrete syntax of the DSL, they allow designers to work with domain specific concepts [5], raising the level of abstraction and making behavioral specifications intuitive to specify and understand.

Model Simulation and Analysis. Once we have specified the behavior of a DSL, the following step is to perform simulation and analysis over the produced specifications. Defining the model behavior as a model will allow us to transform them into different semantic domains. In general, each semantic domain is more appropriate to represent and reason about certain properties, and to conduct certain kinds of analysis [3]. Of course, not all the transformations can always be accomplished: it depends on the expressiveness of the semantic approach.

3 Adding Time to Behavioral Specifications

Formal analysis and simulation are critical issues in complex and error-prone applications such as safety-critical real-time and embedded systems. In such kind of systems, timeouts, timing constraints and delays are predominant concepts [6], and thus the notion of time should be explicitly included in the specification of their behavior.

Most simulation tools that allow the modeling of time require specialized knowledge and expertise, something that may hinder its usability by the average DSL designer. On the other hand, current in-place transformation techniques do not allow to model the notion of time in a quantitative way, or allow it by adding some kind of clocks to the DSL metamodel. This latter approach forces designers to modify the DSL metamodel to include time aspects, and allows them to easily design rules that lead the system to time-inconsistent states (cf. [6]). Furthermore, standard approaches only allow rule patterns (such as LHS) to be composed of system states, making many useful action properties inexpressible without unnatural changes to a system's specification [7].

One way to avoid this problem is by extending behavioral rules with their duration, i.e., by assigning to each action the time it needs to be performed. Analysis of these timed rules cannot easily done using the common theoretical results and tools defined for graph transformations. However, other semantic domains are better suited. We are now working on the definition of a semantic mapping to real-time Maude's rewrite logic [8]. This mapping brings several advantages: (1) it allows to perform simulation, reachability and model-checking analysis on the specified real-time systems; (2) it permits decoupling time information from the structural aspects of the DSL (i.e., its metamodel); and (3) it allows to state properties over both model states and actions, easing designers in the modeling of complex systems.

4 Viewpoint Integration and Consistency

Large-scale heterogeneous distributed systems are inherently much more complex to design, specify, develop and maintain than classical, homogeneous, centralized systems. Thus, their complete specifications are so extensive that fully comprehending all their aspects is a difficult task. One way to cope with such complexity is by dividing the design activity according to several areas of concerns, or *viewpoints*, each one focusing on a specific aspect of the system, as described in IEEE Std. 1471.

Following this standard, current architectural practices for designing open distributed systems define several distinct viewpoints. Examples include the viewpoints described by the growing plethora of Enterprise Architectural Frameworks (EAF): the Zachman's framework, ArchiMate, DoDAF, TOGAF, FEAF or the RM-ODP. Each viewpoint addresses a particular concern and uses its own specific (viewpoint) *language*, which is defined in terms of the set of concepts specific that concern, their relationships and their well-formed rules.

Although separately specified, developed and maintained to simplify reasoning about the complete system specifications, viewpoints are not completely independent: elements in each viewpoint need to be related to elements in the other viewpoints in order to ensure the *consistency* and *completeness* of the global specifications. The questions are: how can it be assured that indeed *one* system is specified? And, how can it be assured that no views impose contradictory requirements? The first problem concerns the conceptual *integration* of viewpoints, while the second one concerns their *consistency*. There are many approaches that try tackle the problem of consistency between viewpoints, many of them coming from ADL community (see, e.g., [3] for a list of such works). However, many of the current viewpoint modeling approaches to system specification used in industry (including the IEEE Std. 1471 itself and the majority of the existing EAFs) do not address these problems (c.f. [9]).

The most general approach to viewpoint consistency is based on the definition of *correspondences* between viewpoint elements. Correspondences do not form part of any of the viewpoints, but provide statements that relate the various different viewpoint specifications—expressing their semantic relationships. The problem is that current proposals and EAFs not consider correspondences between viewpoints, or assume they are trivially based on name equality between correspondent elements and are implicitly defined. Furthermore, the majority of approaches that deal with viewpoint inconsistencies assume that we can build an underlying metamodel containing all the views, which is not normally true. For instance, should such a metamodel consist of the intersection or of the union of all viewpoints elements? Besides, the granularity and level of abstraction of the viewpoints can be arbitrarily different, and they may have very different semantics, which greatly complicates the definition of the common metamodel.

Our efforts are currently focused on the development of a generic framework and a set of tools to represent viewpoints, views and correspondences, which are able to manage and maintain viewpoint synchronization in evolution scenarios, as reported in [10], and that can be used with the most popular existing EAFs.

References

1. Kleppe, A.G.: A language description is more than a metamodel. In: Proc. of the Fourth International Workshop on Software Language Engineering (ATEM 2007), Nashville, USA (2007) `http://megaplanet.org/atem2007/ATEM2007-18.pdf`.
2. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? Computer **37**(10) (2004) 64–72
3. Vallecillo, A.: A Journey through the Secret Life of Models. Position paper at the Dagstuhl seminar on Model Engineering of Complex Systems (MECS) (2008) `http://www.lcc.uma.es/~av/Publicaciones/08/TheSecretJourneyOfModels.pdf`.
4. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)
5. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Proc. of FASE 2008. Volume 4961 of LNCS., Springer (2008) 77–92
6. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: Causality and logical clocks. In: ICGT. (2002) 120–134
7. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Concurrency, Graphs and Models. (2008) 354–382
8. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196
9. Romero, J.R., Vallecillo, A.: Well-formed rules for viewpoint correspondences specification. In: Proc. of WODPEC 2008, Munich, Germany (2008)
10. Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A.: Change management in multi-viewpoint systems using ASP. In: Proc. of WODPEC 2008, Munich, Germany (2008)

Challenges in bootstrapping a model-driven way of software development

Dennis Wagelaar*

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be

Abstract. Current MDE technologies are often demonstrated using well-known scenarios that consider the MDE infrastructure to already be in place. When starting up your own model-driven development infrastructure, because existing boxed-in tools are insufficient, for example, you will come across a number of challenges. Generally, you cannot just sit down and implement all your model transformations and other MDE infrastructure, because it simply takes too long before you get usable results. An incremental approach to putting model-driven development into place gives you the necessary "break-points", but poses extra challenges with regard to the MDE technologies used. This paper discusses some of these challenges, such as bootstrapping a step-wise refinement chain of model transformations, bootstrapping the (usage of the) modelling language, the position of round-trip engineering and useful properties for a model transformation tool.

1 Introduction

Current MDE technologies are often demonstrated from the point of view where either the MDE infrastructure is already in place, or the MDE infrastructure is part of a ready-to-run solution that only requires you to provide some models and off you go. That leaves out the scenario where you'll have to provide your own MDE infrastructure, such as meta-models, model transformations, configuration tools and build processes. A common reason for this scenario is that the existing ready-to-run MDE tools don't provide (exactly) what you want and require you to do some "post-customisation" of the tool's output. That "post-customisation" is a typical model transformation scenario, which means that writing your own model transformation definitions is suddenly within the scope of your software development process. Moving model transformation definition within the scope of your development process poses a number of challenges.

In this paper, I will elaborate on some of these challenges and relate them to each other by means of a central case study. This case study involves the model-driven development of an instant messenger application[1].

* The author's work is part of the VariBru project, which is funded by the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB)

[1] http://ssel.vub.ac.be/ssel/research:mdd:casestudies:im

2 Bootstrapping model transformations and language abstractions

The instant messenger case study started out with a UML model and a simple Java code generator. At this point, one could identify several recurring patterns in the source model: getter and setter methods, explicit observer pattern implementations, explicit abstract factory pattern implementations, etc. There were also parts in the source model that hampered platform independence, such as explicit references to Java API (collection types, applet, AWT). These recurring patterns as well as the platform-specific parts were all candidates for abstraction and model transformation. I've decided to start with automatically generating the getters and setters.

When you define a model transformation, you already know which recurring pattern you want to generate in your output model: getters and setters in this case. Now you need to decide what language abstractions to use in your source model to represent the combination of an attribute and its getters and setters. UML uses the profile mechanism for this, where you can extend the semantics of existing language constructs with stereotypes. Let's say you define an <<EncapsulatedAttribute>> stereotype on top of the attribute language construct. You can then use a model transformation to generate getter and setter methods for each encapsulated attribute. Listing 1.1 shows what such a transformation definition could look like in ATL.

```
module Accessors;
create OUT : UML2 from IN : UML2;
...
rule PublicPropertySingle {
  from s : UML2!"uml::Property" (
      UML2!"Accessors::EncapsulatedAttribute".allInstances()
      ->select(e|e.base_Property=s)->notEmpty())
  using { baseNameS : String = s.accessorBaseNameS; }
  to t : UML2!"uml::Property" (...),
    getOp : UML2!"uml::Operation" (name <- 'get'+baseNameS,
                                   class <- s.class,
                                   ownedParameter <- Sequence{getPar}),
    getPar : UML2!"uml::Parameter" (name <- 'return',
                                    type <- s.type,
                                    direction <- #return),
    getDep : UML2!"uml::Dependency" (name <- 'Get'+baseNameS,
                                     client <- getOp,
                                     supplier <- s),
    getDepST : UML2!"Accessors::accessor" (base_Dependency <- getDep), ...
}
```

Listing 1.1. Accessors ATL transformation module

The interesting part of this transformation definition lies in the complexity of working with stereotypes. Whereas the UML modelling tools provide a user-friendly way of working with stereotypes, model transformation tools fail to hide the underlying complexity of stereotypes. Stereotypes have a meta-class representation that allows them to be instantiated at the model level: stereotype applications. The example transformation definition in Listing 1.1 applies the <<accessor>> stereotype by instantiating the `UML2!"Accessors::accessor"` meta-class. In order to find stereotyped elements, the model transformation has to look up each `UML2!"uml::Property"` instance, as well as each `UML2!-"Accessors::EncapsulatedAttribute"` instance, and match them against each other. This inefficient procedure is a direct result of the fact that the UML2 meta-model has no knowledge of any stereotype definitions.

Another way of introducing new language abstractions is by extending the meta-model. In the case study, that would be the UML2 meta-model. Listing 1.2 shows what the Accessors transformation definitions looks like when you use a meta-model extension[2]. The input pattern is simpler, as it only needs to find instances of `UML2!"accessors::EncapsulatedProperty"`, and the dependency between getters/setters and their attribute no longer requires a separate stereotype instance that links to it.

```
module Accessors2;
create OUT : UML2 from IN : UML2;
...
rule PublicPropertySingle {
  from s : UML2!"accessors::EncapsulatedProperty"
  using { baseNameS : String = s.accessorBaseNameS; }
  to t : UML2!"accessors::EncapsulatedProperty" (...),
    getOp : UML2!"uml::Operation" (name <- 'get'+baseNameS,
                                   class <- s.class,
                                   ownedParameter <- Sequence{getPar}),
    getPar : UML2!"uml::Parameter" (name <- 'return',
                                    type <- s.type,
                                    direction <- #return),
    getDep : UML2!"accessors::AccessorDependency" (name <- 'Get'+baseNameS,
                                                   client <- getOp,
                                                   supplier <- s), ...
}
```

Listing 1.2. Accessors2 ATL transformation module

The previous transformation definition examples illustrate a conflict that has existed in the MDE community for a long time: should you use UML profiles or meta-models for language extension? In the domain of program transformation, people have used modular grammars, as is demonstrated by the Stratego/XT approach to implementing language extensions and transformation definitions [5]. Considering this and the added complexity of stereotypes in the domain of model transformation, direct extension using modular meta-models seems the way to go.

However, it is not likely that UML tools are going to support unlimited meta-model extension any time soon. This is demonstrated by the fact that UML has been defined by a meta-model for over 10 years, while UML extensions are still defined as profiles today. The technical reason behind this is that in graphical languages, syntax extension is more complicated than in textual languages. Extension by stereotypes is easy from a concrete syntax point of view: just add UML keywords to the original graphical representation of the stereotyped model element. Tools for defining graphical concrete syntax, such as the Eclipse

[2] I still use the name UML2 to refer to the extended meta-model: UML2 is only a symbolic name that can be bound to any concrete meta-model.

Graphical Modeling Framework, are not nearly as easy to use as the tools for meta-modelling.

I believe that this paradox, where easy language extension causes complex model transformation definitions, can be mitigated by providing an automated translation from UML models with profiles to models based on "pure" meta-models. A simple transformation can generate the meta-model representation of a UML profile. A higher-order transformation can generate a transformation definition that translates stereotype instances applied to regular model elements to special model elements. The result would be a situation where you can create UML models with profiles, while you can also write the kind of transformation definitions shown in Listing 1.2.

3 Evolving a step-wise refinement chain

The previous section started out from the initial state of the instant messenger case study, which exhibited several candidates for language abstraction and model transformation: getter and setter methods, explicit observer pattern implementations, explicit abstract factory pattern implementations, explicit references to Java API (collection types, applet, AWT), etc. Now that a language abstraction and a model transformation definition are in place for getter and setter methods, a start can be made with the other abstractions/transformation definitions.

Keeping the transformation of each language abstraction you've introduced nicely seperated in its own transformation definition reduces local complexity. You can concentrate on solving a local transformation problem. Global complexity does not reduce, however, and generally requires managing. Take the model transformation for the observer pattern, for example. This model transformation adapts the setters of each observed attribute such that the `update()` method of the observers is triggered. That means that the setter methods must already exist in the model.

Batory et al. have already pointed out that you need to manage dependencies between the different refinement steps in [1]. There even are a number of model transformation languages that support *critical pair analysis* [6][7], which provides an automated analysis of any dependencies between model transformation definitions. Unfortunately, critical pair analysis is not an easy computing task and doesn't scale well. It is also not applicable to all model transformation languages.

Normally, transformation definitions in a step-wise refinement chain are designed to work on the output of the previous model transformation and the developer is conscious of the dependencies. When adding *alternatives* for a specific model transformation definition, or when *changing* one of the model transformation definitions in the refinement chain, the situation becomes different. Will the alternative/changed model transformation definition still provide what is required by the next transformation steps? Techniques such as critical pair analysis may be able to tell whether the following transformation steps will still trigger, but can't say much about changes in the semantics of the outcome.

I rather believe that the dependencies between transformation steps must be concentrated in semantically rich meta-classes. For example, when the observer pattern transformation definition requires the presence of setter methods in order to adapt them, the meta-model should provide an explicit notion of setter methods. By converging the dependencies between transformation steps in semantically rich meta-classes, automated analysis of such dependencies becomes much easier.

4 To round-trip or not to round-trip

The instant messenger case study uses an incomplete code generator, that does not understand any of the UML behaviour diagrams. Instead, it uses UML's support for adding native method bodies in Java, C++, etc. to operations. Obviously, Java editing support in a UML case tool is nowhere near as advanced as what Eclipse JDT can provide. As a result, all method bodies for the instant messenger are written in Eclipse JDT, after the skeleton code is generated. These method bodies are then manually added back to the UML model, such that the next code generation cycle picks them up. This approach is a terrible maintenance nightmare and cannot be used in real situations. Merging-style code generators, such as EMF's JET and Acceleo, look nice in the beginning. They provide a way to propagate any changes in the model back to the code while merging with the existing, hand-written code. They do not provide a way to propagate code changes back to the model, however. That means that model and code can still grow apart from each other and at some point the model no longer reflects what is in the code.

A full round-trip engineering (RTE) approach can be used to solve this problem. RTE is useful in situations where the model does not provide a complete view of the software. In the case of the instant messenger, the model provides a complete specification, but still an incomplete view: the Eclipse JDT view of the method bodies is far more useful. The problem with RTE is that it is very hard to do in a general way, as is illustrated by Van Paesschen in [3] and by Antkiewicz et al. in [4]. First of all, the model transformations must be executable in both directions. There are bi-directional model transformation languages, such as QVT Relations [8] and Triple Graph Grammars [9], that support this. Writing a bi-directional transformation definition is more complex than writing a single direction transformation, however. Your output patterns have a double function as input patterns as well. As such, you must make sure that your output patterns function correctly as input patterns as well. Then, both Van Paesschen and Antkiewicz acknowledge that RTE requires more than just bi-directional transformations in the case that your model transformation definitions are not injective. Consider a simple model transformation that applies a profile to a model if and only if that profile was not yet applied. The reverse of this transformation is not possible without having the original source model available.

Recent work on RTE shows a different approach to the problem. In [10], Hettel et al. point to the possibility of doing RTE with only having a forward transformation definition and reference source and target models available. In [11], Xiong et al. actually go as far as claiming an initial implementation of such a system based on ATL. If this approach can be made to work in general, where the developer only needs to provide source model and forward model transformation definition, this removes any disadvantages that RTE has over simple forward engineering.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering **30** (2004) 355–371
2. Henriksson, A., Larsson, H.: A Definition of Round-trip Engineering. Technical report, Department of Computer and Information Science, Linköpings Universitet, Linköping, Sweden (2003)
3. Van Paesschen, E.: Advanced Round-Trip Engineering: An Agile Analysis-driven Approach for Dynamic Languages. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium (2006)
4. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. In: Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), Genova, Italy. Volume 4199 of Lecture Notes in Computer Science., Springer-Verlag (2006) 692–706
5. Bravenboer, M., Visser, E.: Designing Syntax Embeddings and Assimilations for Language Libraries. In: Models in Software Engineering. Workshops and Symposia at MoDELS 2007. Volume 5002 of Lecture Notes in Computer Science., Springer-Verlag (2008) 34–46
6. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci. **127** (2005) 113–128
7. Mens, T., Kniesel, G., Runge, O.: Langages et Modèles à Objets (LMO 2006). In: Proceedings of Langages et Modèles à Objets (LMO 2006), Nîmes, France. (2006) 167–183
8. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. (2005) Final Adopted Specification, ptc/05-11-01.
9. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Tinhofer, G., ed.: Proceedings of the WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science. Volume 903 of Lecture Notes in Computer Science., Springer-Verlag (1994) 151–163
10. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT 2008), Zürich, Switzerland. Volume 5063 of Lecture Notes in Computer Science., Springer-Verlag (2008) 31–45
11. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards Automatic Model Synchronization from Model Transformations. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07), Atlanta, Georgia, USA, ACM Press (2007) 164–173

UML/OCL Verification In Practice

Jordi Cabot and Robert Clarisó

Universitat Oberta de Catalunya
{jcabot, rclariso}@uoc.edu

Abstract. In the MDD approaches, models become the primary artifact of the development process and the basis for code generation. Identifying defects early, at the model-level, can help to reduce development costs and improve software quality. There is an emerging need for verification techniques usable in practice, i.e. able to find and notify defects in real-life models without requiring a strong verification background or extensive model annotations. Some promising approaches revolve around the *satisfiability* property of a model, i.e. deciding whether it is possible to create a well-formed instantiation of the model. We will discuss existing solutions to this problem in the UML/OCL context. Our claim is that this problem has not yet been satisfactorily addressed.

1 Introduction

Model-driven development (MDD) advocates for the use of models as development artifacts. In this context, code is no longer written from scratch but synthesized from models (semi-)automatically. Therefore, any defect in the model will propagate into defects in the code. In MDD model correctness becomes a key factor in the quality of the final software product.

Although the problem of ensuring software quality has attracted much attention and research, it is still considered to be a Grand Challenge [1]. In this sense, this paper argues that this grand challenge must be adapted and extended to cover the verification of modeling notations commonly used in MDD approaches. In this field, it is essential to provide a set of tools and methods that helps in the detection of defects at the model-level and smoothly integrates in existing MDD-based tool-chains without an excessive overhead. Characteristics of existing tools, such as required designer interaction or manual model annotations seriously impair its usability in practice.

We will discuss existing approaches and their limitations to motivate that this problem is still unsolved and remains an important challenge in MDE. Throughout the paper, we will focus on the UML as an example of a MDD modeling language. However the contents of the paper also apply to all kinds of domain-specific modeling languages (DSMLs). Some of the problems identified herein already arise when just considering the graphical elements of the models. However, complexity increases when designers use textual languages (e.g. OCL) to improve the precision/formalization of the models. Note that all existing model-level verification approaches proceed by translating the models to a more formal language (e.g. Alloy). Therefore, characteristics (and limitations) of correctness techniques for UML/OCL models largely depend on the properties of available verification techniques for those underlying formal languages.

2 A definition of "correctness"

One of the first problems when discussing correctness at the model-level is the large number of existing correctness notions, according to many different criteria: static vs dynamic, inter-diagram vs intra-diagram, . . .

A first degree of correctness can be that of *consistency* and *well-formedness*, checking that all uses of a model element (possibly in different diagrams) are consistent with its declaration and that the model as a whole can be expressed as a correct instantiation of its meta-model.

Even though this analysis provides an initial level of defect detection, it does not take into account the *semantic* correctness of the model being defined. By semantics, we consider the set of required conditions (i.e. integrity constraints) that should be satisfied by any correct instance of the model. These conditions may be implicit in the model notation (like the multiplicity constraints in UML associations) or explicitly defined using a constraint language like OCL. These semantics problems may affect either the static (structural) or dynamic (behavioral) view of the system. Examples of possible errors are the non-executability of pre and postconditions in an operation or the presence of contradictory invariants or association multiplicities.

Different modeling notations and constraint languages have varying degrees of expressivity. Therefore, each notation creates a different challenge in terms of decidability and efficiency, and suggests a different set of analysis techniques. Due to space limitations, we will focus our discussion on the study of UML static models. In UML, static models may be expressed as class diagrams complemented with a set of OCL constraints.

A fundamental semantic correctness notion in static models is that of *model satisfiability*. (Strong) Satisfiability consists in deciding whether it is possible to create a non-empty and finite instantiation of the model in such a way that all integrity constraints are satisfied. Clearly, an unsatisfiable model is useless since every time users try to create a new object, e.g. instantiating one class of the model, at least one of the integrity constraints will become violated.

The importance of satisfiability comes from the ability to define many other correctness properties, such as liveliness, constraint redundancy, subsumption and so forth, in terms of the satisfiability problem. For example, a designer can check if an integrity constraint C is redundant by formulating a satisfiability problem where $\neg C$ replaces C in the model. If that model is satisfiable, it means it is possible to satisfy the remaining integrity constraints while violating C, so C is not a redundant constraint.

In the next sections we describe current approaches for UML/OCL model satisfiability and possible further research directions to cope with this challenging problem as a basis for identifying semantic defects in UML/OCL models.

3 State of the Art

In order to succeed in a MDD context, we believe any method for model satisfiability should fulfill the following list of requirements:

- Understand the input notation used by the designer (e.g. UML/OCL), not a formal notation nor a subset of that notation. If an internal formal notation is used, it should be transparent to the designer.
- Analyze the designer's model *as is*, without requiring any type of manual annotation.
- Perform the analysis automatically and without requiring user interaction.
- Provide results in a format meaningful to the designer.
- Be efficient and scale up to support large real-life examples.
- Integrate seamlessly into the designer tool chain.

Existing solutions lack of one or more of the previous qualities, and that might justify the lack of adoption of model-level verification tools in current MDD projects. In what follows, we describe the weaknesses of existing methods in terms of the main challenges they have to face when trying to satisfy the previous properties.

1. **Decidability:** The complexity of satisfiability analysis mainly depends on the expressiveness of the logic used to define the model and its constraints. Allowing a notation such as OCL makes the problem undecidable. Three different strategies are used to confront this undecidability:
   - Relaxing automation: Methods based on theorem proving might require user assistance during proofs, e.g. HOL-OCL [2].
   - Constraining the logic: Some methods work on a restricted subset of OCL (e.g. [3]) and some others do not support OCL at all (e.g. [4]). There is a trade-off between expressiveness (e.g. "are numerical constraints supported?") and complexity.
   - Performing bounded verification: If a finite bound is defined, it is then possible to check a property for all possible instances up to that maximum size, e.g. [5, 6]. This type of analysis can be used to prove the satisfiability of a model, but the lack of counterexamples within a bounded search space cannot be used to prove its unsatisfiability.
2. **Efficiency:** Reasoning on UML class diagrams is EXP-complete [7] even without OCL constraints and thus, current tools do not scale-up well which makes efficiency a concern for most non-trivial models.
3. **Usability:** Verification tools are often disappointing from the point of view of a designer. One of the main reasons is that tools do not directly manipulate the UML/OCL model but first translate it into a formal language (Alloy [6], CP [5], HOL [2], DL [8]) where the verification process takes place. Therefore, a good knowledge of this underlying language may be required to operate effectively, e.g. while selecting adequate parameters, tuning the model for the analysis or interacting with the tool. For this same reason, the interpretation of the results of the analysis might be complex and should be expressed in terms of the original model.

4. **Expressiveness:** The richness of the modeling languages (specially of the UML and OCL standards) creates a challenge for tool developers, who must support a wide variety of modeling constructs. Furthermore, the varied ecosystems of design tools, development tools and IDEs creates additional difficulties in terms of integration and interoperability.

4 Research Agenda: Promising Research Directions

The aim of this section is to sketch possible future research directions we believe may help in overcoming the previous limitations.

– **Automatic selection of the most appropriate verification approach for a specific model.** Each approach presents a different trade-off regarding the verification process. Depending on the model one approach may be more suited than others. For instance, for UML models without integrity constraints (a decidable problem) it may be better to use complete approaches (as those based on Description Logics) instead of approaches based on bounded verification.
– **Model partition to improve performance.** In most cases, the verification of a model m can be defined in terms of the verification of the submodels $m_i, \ldots, m_n$. Techniques for slicing the model in a subset of independent submodels (with the subsets to be computed depending on the property to be verified) will definitely help in improving the efficiency of the process due to its exponential nature.
– **Establish public community benchmarks to compare different tools and approaches.** Benchmarks provide an excellent resource to measure progress and the significance of a contribution. The existence of widely accepted benchmarks for model verification can foster progress and allow existing approaches to mature and exchange ideas.
– **Search space reduction for bounded methods.** Bounded methods require a finite search space. A smaller search space improves the efficiency but impairs the completeness of the verification. A preliminary analysis of the model could provide some insight on the best bounds of the search space as a trade-off between the two properties.
– **Apply SAT Modulo Theories to model satisfiability.** SAT Modulo Theories (SMT) is a promising technique for checking the satisfiability of a complex formula which combines recent improvements in SAT tools with the power of a custom solver specialised in a given logic [9]. In the case of model satisfiability, the challenge is identifying a subset of the modeling language which is sufficiently expressive yet allows efficient decision procedures.
– **Feedback improvements for defect correction.** Tools should not only be able to answer whether the model is correct. If the answer is no, the tool should be able to explain where, why and how it can be corrected.
– **Incremental verification.** The specification of a model is an iterative process where the model is continuously refined by means of adding, changing or

deleting some of its elements. Clearly, once a first version has been verified, we should be able to prove the correctness of new model versions without verifying the whole model again. Instead, only the "updated" parts should be considered.
– **Model normalization.** Normalizing a model, i.e. rewriting complex modeling constructs in terms of more basic ones, prior to the verification process helps to reduce the complexity of the verification algorithms that now do not need to consider the full language expressiveness. For instance, see [10] for some rules for normalizing OCL constraints.

5 Conclusions

Model-level verification is a key step toward improving software correctness. Despite the amount of research efforts devoted to this problem, existing approaches for model verification exhibit shortcomings that limit their applicability and adoption in MDE projects. In this paper, we have identified these problems and a possible set of research directions that may help in the creation of a new generation of verification tools that offer effectiveness, efficiency and usability.

References

1. Jones, C., O'Hearn, P., Woodcock, T.J.: Verified software: A grand challenge. IEEE Computer **39**(4) (2006) 93–95
2. Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zurich (2006)
3. Queralt, A., Teniente, E.: Reasoning on UML class diagrams with OCL constraints. In Embley, D.W., Olivé, A., Ram, S., eds.: ER. Volume 4215 of Lecture Notes in Computer Science., Springer-Verlag (2006) 497–512
4. Baruzzo, A., Comini, M.: Static verification of UML model consistency. In Hearnden, D., S, J., Rapin, N., Baudry, B., eds.: 3rd Workshop on Model Design and Validation (MoDeV2a). (2006) 111–126
5. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: MoDeVVa 2008. ICST Workshop. (2008) available online: http://gres.uoc.edu/pubs/MODEVVA08.pdf
6. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007). Volume 4735 of Lecture Notes in Computer Science. (2007) 436–450
7. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence **168** (2005) 70–118
8. van der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Proceedings of 6th International Conference UML 2003 - The Unified Modeling Language. (October 2003) 326–340
9. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT an SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM **53**(6) (November 2006) 937–977
10. Cabot, J., Teniente, E.: Transformation techniques for OCL constraints. Science of Computer Programming. **68**(3) (2007) 179–195

Improving Requirements Specifications in Model-Driven Development Processes

Jordi Cabot and Eric Yu

Department of Computer Science, University of Toronto
{jcabot,eric}@cs.toronto.edu

Abstract: Understanding the organizational context and rationales (the "Whys") that lead up to system requirements (the "Whats") help us to analyze the stakeholders' interests and how they might be addressed, or compromised, by the different design alternatives. These aspects are very important for the ongoing success of the system but are not considered by current Model-Driven Engineering (MDE) methods. In this paper we argue for the necessity of extending MDE methods with improved requirements techniques based on goal-oriented techniques for the analysis and specification of the organization context and discuss the benefits and challenges of such integration.

1. The Challenge

Understanding the purpose, goals, and intentions of a software system is a necessary condition for its successful design and implementation. By understanding the goals, the needs of the organization can be better aligned with the functionality provided by the system. However, organizations are complex in nature and in general, when attempting to design a system, there are usually many alternatives, each with different implications for the many parties (stakeholders) that may have an interest in the system. To identify, evaluate, and select the best alternatives is a considerable challenge but a key aspect for the ongoing success of the system.

Unfortunately, such an organizational-based analysis of the system's requirements has not yet been adopted by current model-driven engineering (MDE) methods. In general, MDE methods limit the requirements specification to the elicitation of the functional requirements of the system (i.e. the behaviour/services/tasks the system-to-be must provide) but neither consider the organizational context of the system nor its non-functional requirements (NFRs) [10] (as usability, extensibility and so on).

We believe this restricted requirements analysis challenges the ability of current MDE methods to provide an accurate system representation at the end of the development process. As a result, the generated system may not satisfy the stakeholder's expectations because their goals have not been sufficiently considered.

Therefore, in our opinion, providing a better support for the requirements specification and analysis is a still a research challenge for MDE and one of the reasons that may explain the low adoption of MDE methods among software professionals.

In the next sections, we present the goal-oriented methods (Section 2) as a possible response to this challenge by means of integrating these methods within MDE approaches (Section 3). Finally, we discuss a list of open problems that must be solved to make this integration feasible (Section 4) and end up with the conclusions.

2. Goal-Oriented Requirements Engineering Methods

The requirements engineering community has largely addressed and recognized the leading role played by goals in the requirements engineering process. Nowadays, beliefs and goals (the "Whys" of the system) have been incorporated into most requirements acquisition frameworks, commonly known as goal-oriented requirements engineering (GORE) methods. A goal is an objective to be achieved by the system under consideration. Goals may refer to either functional concerns or NFRs.

It has been proven that an explicit goal modelling representation helps stakeholders in coming up with the initial requirements in the first place, always one of the most problematic tasks in any development process. By focusing on goals instead of specific requirements, analysts enable stakeholders to communicate using a language based on concepts (e.g. goals) with which they are both comfortable and familiar [2].

Other well-known benefits of using goal-based methods are their higher level view of the system requirements as compared with traditional requirements specifications; their link with business goals and decisions; the stability of goals compared with the requirements that implement them; the ability to consider alternative solutions and the potential to improve the traceability of the system by establishing clearer links between process design decisions and technical system alternatives [12], which also facilitates the system evolution and maintenance. Besides, GORE methods allow verifying and validating the requirements early in the development cycle thus avoiding propagating the errors and misunderstandings through the later phases of the process.

3. Integrating GORE and MDE methods

Given the characteristics and potential benefits of current GORE methods, we believe that they are a strong candidate to cover the gap of current MDE approaches when it comes to requirements specification and analysis.

In this sense, we think that an integrated software development process, where the typical phases of MDE methods (analysis, design and code-generation) are preceded by the early and late requirements phases of GORE methods (e.g. [7]), could overcome the limitations of MDE methods pointed out in the introduction. For specifying the two first phases we propose to use the specialized *i** modelling framework [16]. UML (or similar DSLs) are the logical option for the rest.

It is important to note that in order to maximize the benefits of such integration, goals should be taken into account in all phases of the development process: functional goals must be "realized" by the static and dynamic behaviour of analysis and design models while non-functional ones drive the selection of possible design alternatives when moving from one phase to the following. This is a distinction from other recent works that have tried to bridge the gap between GORE and MDE methods by (partly) generating excerpts of an initial UML-based specification from the goals information [4],[11],[9,14].

In what follows, we provide a short description of each phase:

- **Early requirements phase**. Intentions (goals) of the actors (i.e. stakeholders) in the organizational context are modelled and analyzed. In this phase we explore why the system is needed, what alternatives might exist to attain each goal and what are the implications of each alternative for the stakeholders (i.e. what is their effect in terms of its contribution to the desired NFRs).
- **Late requirements phase**. The system is added as a new actor that can be in charge of fulfilling some of the desired goals. An analysis of the different alternative designs is performed and a final decision on the set of responsibilities to delegate to the system is made. From this assignment we may derive the list of functional requirements (tasks) for the system to be.
- **Analysis phase.** The static (i.e. structural) and dynamic (i.e. behavioural) aspects of the domain required by the software system to perform its functions are specified. Goals must still be taken into account when specifying these models. Most aspects can be specified using different semantically-equivalent alternatives, each one with its trade-offs. The selection of the most suitable one for the specific system under development should be influenced by the functional and non-functional goals that the system must fulfil.
- **Design phase.** The design phase adapts the previous models to the technical features of the technological platform where the system is going to be implemented. This implies defining a sequence of transformations that refine the models until all elements have a direct correspondence with the programming primitives available in the production environment. In general, all transformations are non-deterministic, in each refinement we have a number of possible design alternatives implying different non-functional qualities. The goals specified in the GORE models, in particular the non-functional ones, can help us to drive the transformation process and select the best alternative according to the requested NFRs.
- **Code Generation. A**ll software elements of the system (code, data structures,…) are generated from the design models.

4. Open Research Problems

Despite the potential benefits of mixing GORE and MDE methods, there are a number of open problems that must be solved to make their integration feasible in practice. In particular, the main challenge is to ensure that the benefits outweigh the extra work implied by the explicit definition of the goals at the beginning of the MDE process.

Therefore, our list of research challenges goes in the direction of smoothing the application of our approach by automating and simplifying as much as possible the transition between the different phases. A partial list is the following:

- Development of a common (meta)modelling framework. To bridge the gap between GORE and MDE models and ease the definition of model-to-model transformations between them, both kinds of models should be specified using "compatible" modelling languages. Current efforts try to formalize the i* language as a MOF-compliant metamodel (see [15], [3] and [1]).
- Reusing the information in the goal models to partially generate the static and dynamic aspects of the system. Existing approaches focus on the generation of preliminary use case diagrams, activity diagrams and class diagrams (see [8] [4],[11],[9, 14] [11] for more information). This generation must also consider the representation of the non-functional requirements. See [5, 13] for examples.
- Linking and traceability techniques between the business models (and business processes) and the analysis and design models and the architectural decisions. For instance, we should be able to detect, for each system goal, the subset of the design models that ensures the fulfilment of that goal.
- Incremental model synchronization techniques between all kinds of models.
- Consistency analysis between the different models involved in the process. Low-level models must be a consistent refinement of the higher-level ones.
- Propose a systematic way of dealing with NFRs in MDE (and in UML).
- Reusing the information about the system NFRs to influence in the analysis phase as, for instance, during the selection of the right analysis pattern to model certain aspects of the domain (e.g. to represent the *price* of a product we could just specify an attribute of type *Real* or use more complex patterns that facilitate recording the price in different money units; if *internationalization* is one of the desired NFRs for the system we should stick to the second option).
- Defining NFR-based model transformations to automatically transition from the analysis to the design phase. This transformation is not deterministic. Each modelling construct in the analysis models could be expressed using different design alternatives. Each alternative results in a different contribution to the non-functional qualities of the system. In our integrated approach, we could automate the transformation process by selecting the design alternatives that better match the specified NFRs in the requirements phases. This way, we also ensure that the resulting system is better aligned with the stakeholder's interests.
- Establishing methods for adjusting MDE processes to the reality of the development team. An analysis of the knowledge and skills of team members and team structures will help to tune the development process so that we can maximize the team abilities and facilitate the process application.

5. Conclusions

We have identified the limitations of current MDE methods when it comes to understand the needs and goals of the organization and its stakeholders regarding the software system to be developed. This limited requirements analysis in most common MDE methods negatively impacts the quality of the generated system. We believe that improving this requirements support is still an important MDE research challenge.

As a possible solution we have proposed to extend MDE methods with goal-oriented requirements engineering techniques. We think this enriched development process may be better suited to produce software systems that satisfy the stakeholders' expectations. It may happen that using our approach only pays off for certain types of systems and/or domains and/or organizations. This needs to be empirically validated.

Acknowledgements

Work supported by the Spanish Ministry of Education and Science (project TIN2005-06053), the grant 2007 BP-A 00128 (Catalan Government) and the Natural Science and Engineering Research Council of Canada.

References

1. Amyot, D. URN metamodel proposal. Available: http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/DraftZ151Metamodel
2. Anton, A. I.: Goal Identification and Refinement in the Specification of Software-Based Information Systems. PhD Thesis. Georgia Institute of Technology (1997)
3. Ayala, C. P., Cares, C., Carvallo, J. P., Grau, G., Haya, M., Salazar, G., Franch, X., Mayol, E., Quer, C.: A Comparative Analysis of i*-Based Agent-Oriented Modeling Languages. In: Proc. SEKE'05 (2005) 43-50
4. Bertolini, D., Delpero, L., Mylopoulos, J., Novikau, A., Orler, A., Penserini, L., Perini, A., Susi, A., Tomasi, B.: A Tropos Model-Driven Development Environment. In: Proc. CAiSE Forum 2006, (2006)
5. Cysneiros, L. M., Leite, J. C. S. d. P.: Nonfunctional Requirements: From Elicitation to Conceptual Models IEEE Trans. Software Eng. 30 (2004) 328-350
6. Forward, A., Lethbridge, T. C.: Problems and Opportunities for Model-Centric Versus Code-Centric Development: a Survey of Software Professionals. In: Proc. MiSE´08 (ICSE Workshop), (2008)
7. Giorgini, P., Kolp, M., Mylopoulos, J., Pistore, M.: The Tropos Methodology: an overview. In: F. Bergenti, M.-P. Gleizes, and F. Zambonelli, (eds.): Methodologies And Software Engineering For Agent Systems. Kluwer Academic Publishing (2003)
8. Jiang, L., Topaloglou, T., Borgida, A., Mylopoulos, J.: Incorporating Goal Analysis in Database Design: A Case Study from Biological Data Management. In: Proc. RE'06, (2006) 196-204
9. Martínez, A., Pastor, O., Estrada, H.: Closing the Gap between Organizational Modeling and Information System Modeling. In: Proc. WER'03, (2003) 93-108
10. Mylopoulos, J., Chung, L., Nixon, B. A.: Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. IEEE Trans. Softw. Eng. 18 (1992) 483-497
11. Perini, A., Susi, A.: Automating Model Transformations in Agent-Oriented Modelling. In: Proc. AOSE'05, LNCS, 3950 (2005) 167-178
12. Regev, G., Wegmann, A.: Where do Goals Come from? The Underlying Principles of Goal-Oriented Requirements Engineering. In: Proc. RE'05, (2005) 339-349
13. Salazar-Zarate, G., Botella, P., Dahanayake, A.: Introducing non-functional requirements in UML. In: UML and the unified process. IGI Publishing (2003) 116-128
14. Santander, V. F. A., Castro, J.: Deriving Use Cases from Organizational Modeling. In: Proc. RE'02, (2002) 32-42
15. Susi, A., Perini, A., Mylopoulos, J., Giorgini, P.: The Tropos Metamodel and its Use. Informatica 29 (2005) 401-408
16. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In: Proc. RE'97, (1997) 226-235

Lessons of Experience in Model-Driven Engineering of Interactive Systems: Grand challenges for MDE?

Gaëlle Calvary[1], Anne-Marie Pinna[2]

[1] Laboratoire d'Informatique de Grenoble, Equipe IIHM, 385 Rue de la Bibliothèque, BP 53
38041 Grenoble Cedex 9, France
[2] Laboratoire I3S, Bâtiment ESSI, 650, Route des Colles, B.P. 145
06903 Sophia-Antipolis Cedex, France
Gaelle.Calvary@imag.fr, pinna@polytech.unice.fr

Abstract. Model-based approaches have been recognized as powerful for separating concerns when designing a User Interface (UI). However model-based generation of UIs has not met a wide acceptance in the past. Today, taking benefit from MDE advances, models are revisited in Human Computer Interaction (HCI) for engineering multi-contexts interactive systems. This paper relates some difficulties and requirements highlighted by experience. Among them is the need of support for sustaining the collaboration between models both at design time and runtime. Another requirement is about the capitalization of models, mappings and metamodels for saving the know how in HCI. Last but not least requirement is about evaluation. How shall we evaluate the strengths and weaknesses of MDE for HCI?

Keywords: Human Computer Interaction, Interactive system, User interface, Multi-models, Multi-actors, Multi-views, Capitalization, Evolution, Evaluation.

1 Introduction

Model-based approaches have been widely explored in Human-Computer Interaction (HCI) for long. The motivation was to force designers focus on user's task instead of blending several concerns including cosmetic considerations. Rapidly, task modeling has emerged as a wise starting point when engineering interactive systems. Automatic generators of User Interfaces (UIs) have appeared (e.g., ADEPT [2]) but the poor quality of the resulting UIs killed the approach for long. Later on, the increasing diversity of platforms (e.g., PC or PDA) and the rise of Platform Dependent versus Independent Models (PSM versus PIM) brought models back to life in HCI [4]. Now, we are at the point of blurring the distinction between design time and runtime for making it possible for the end-user to fashion his/her own interactive space according to his/her feelings and needs as well as to the arrival and departure of interaction resources. The purpose of this paper is to relate where we are in the tandem MDE-HCI and which issues need to be solved for going further in Meta-Design [1].

2 Where we are in MDE for HCI

The canonical functional decomposition of interactive systems makes the distinction between the Functional Core (FC) and the UI. From a methodological point of view, along a forward engineering process, the starting point is most of the time a task model which structures the user’s goal into sub-goals. Conversely, reverse engineering consists in analyzing legacy systems for recovering the models that may have driven the design (e.g., the task model) and/or implementation (e.g., the architecture model). Using chains of tools (e.g., the UsiXML arsenal of tools - see http://www.usixml.org/), it is possible to hybrid forward and reverse engineering [3].
In this section, we first focus on the UI design, and then address the implementation of the whole interactive system.

2.1 MDE for the design of UIs

Designing a UI means setting all the degrees of freedom (e.g., the structure of the UI, the choice of interactors) based on given requirements (e.g., the user’s task and preferences, the ergonomic criteria that have been elicited, the targeted platforms). Fig. 1 shows four functionally equivalent UIs: they support the same user’s task T that consists in accomplishing T1 and T2 in interleaving. The UIs differ from an ergonomic point of view. In a, the subtasks T1 and T2 are directly observable whilst they are browsable in b-c-d. Browsing has a human cost: at least one physical action for commuting between subtasks.
Along a forward engineering process, once the task is modeled, decisions have to be made about the structure and rendering of the UI. Decisions are mostly driven by ergonomics (e.g., minimal actions). At this point, the UI is classically a mockup either drawn on paper sheets or prototyped using languages (e.g., Smalltalk) and tools (e.g., JBuilder). Predictive and/or experimental evaluations must be done.

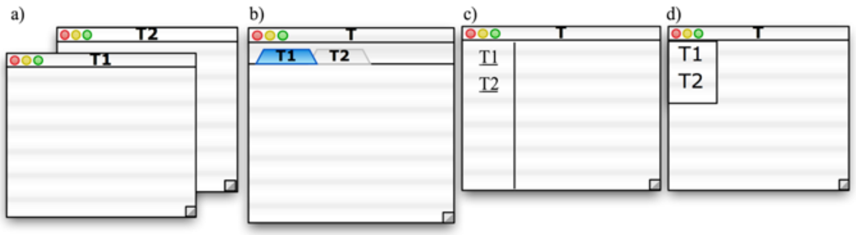


Fig. 1. Four functionally but not non functionally equivalent UIs.

Our lessons of experience are listed below:
MDE for capturing the know-how in HCI: the relevant models (e.g., task, structure, interactors) are identified giving rise to several notations (e.g., CTT [6]), metamodels and versions of metamodels. One open issue specific to HCI is the modeling of ergonomic properties.
MDE for platform independence: a plethora of Platform/Rendering Independent Languages has appeared (e.g., UsiXML, UIML (www.uiml.org), RIML (european CONSENSUS project), XIML (www.ximl.org), XAML (Microsoft), AUIML (IBM)). There is a need of reconciliation to clarify the state of the art.
Miscellaneous statements about MDE in practice: as the focus has mostly been set on

high level models so far (typically the user's task), these models are often overloaded with information typically related to transformation (e.g., the targeted platform). Most of the time, transformations are mono-technological (e.g., HTML). In addition, the rendering can be improved.

2.2 MDE for the implementation of interactive systems

Software architecture models (e.g., ARCH) improve software quality. ARCH refines the UI into sub functions among which is the Dialog Controller (DC). The DC is in charge of piloting interaction while ensuring consistency between the FC and the UI. The DC is a kind of implementation of the task model. General research in MDE deals with the FC only. Below are our lessons of experience about the whole system.

MDE for models collaboration: the global picture (Fig. 2) implies biaxial mappings and transformations: vertical for the engineering process, horizontal for complementary descriptions and/or code [5].

MDE for models at runtime: we are at the point of keeping the design models and design rationale (i.e., the biaxial lattice of Fig. 2) alive at runtime so that to enable the revision at runtime of any design decision.

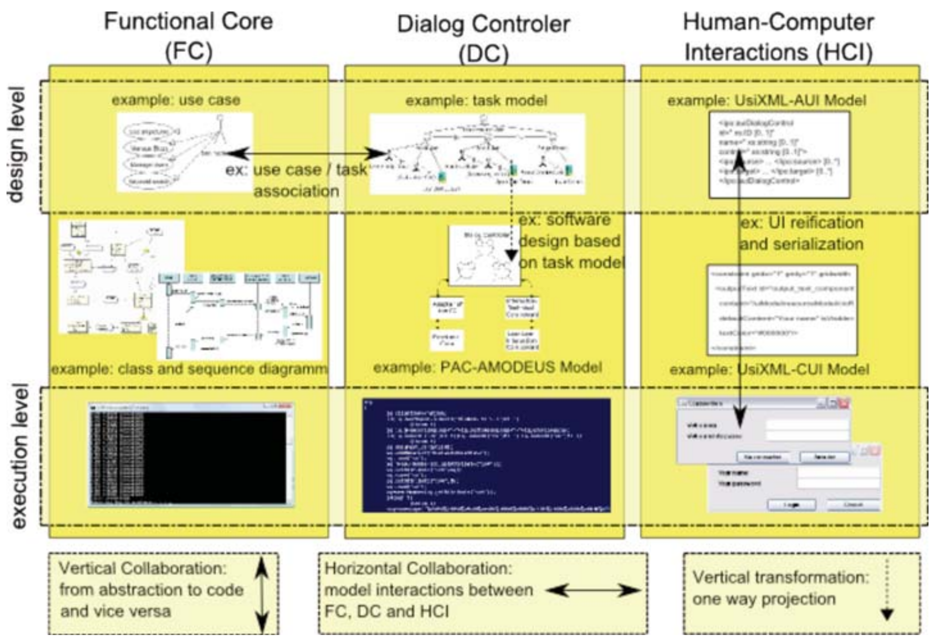

Functional Core (FC)
Dialog Controler (DC)
Human-Computer Interactions (HCI)
design level
execution level
example: use case
example: task model
example: UsiXML-AUI Model
ex: use case / task association
ex: software design based on task model
ex: UI reification and serialization
example: class and sequence diagramm
example: PAC-AMODEUS Model
example: UsiXML-CUI Model
Vertical Collaboration: from abstraction to code and vice versa
Horizontal Collaboration: model interactions between FC, DC and HCI
Vertical transformation: one way projection

Fig. 2. Biaxial and bidirectional transformations/mappings.

Miscellaneous statements about MDE in practice: automatic forward generators mostly produce "fast food" UIs in which models and transformations are lost. The transformations do not preserve collaborations at the code level, and horizontal collaborations between FC and UI are not yet well supported. In addition, consistency between models, transformations and code is not ensured at runtime along the models evolution. This calls for further research. Next section elicits three main requirements.

3 What we need for going further: three requirements to conclude

To our understanding, HCI is an interesting domain for both illustrating and inspiring research in MDE: on one hand, there is a long know-how in models and transformations in HCI. It provides a lot of knowledge to support experiments in MDE. On the other hand, keeping models at runtime raises new perspectives in both HCI and MDE. Whist the global picture has been demonstrated on simple case studies in HCI so far, we now aim at going beyond toy applications. This calls for mastering MDE for HCI, i.e., being able to reuse mature generic supports (methods and tools) from MDE so that to concentrate our efforts on the specific features of HCI (e.g., ergonomics). We identify three major requirements for going further: the support for multi-models and multi-actors, capitalization and evaluation of models.

#1. Support for multi-models and multi-actors

From now on, an interactive system is depicted as a graph of models that conveys both its design rationale and evolution over time. The models cover the internal state of the system (FC and UI) as well as its deployment on the interaction resources. We aim at formalizing the ergonomic properties so that to (1) characterize well-formed interactive systems, (2) be aware of the validity domain of interactive systems, (3) predict the effect of transformations, and (4) compute the conditions to satisfy and transformations to perform for ensuring a set of properties. Defining the actors in charge of such reasoning is interesting: who among designers and end-users might be in charge of observing and/or controlling graphs of models? Which views would be appropriate for whom and when? Fig. 3 and 4 provide two examples. In Fig. 3, the designer can delete interactors by placing a toolglass on either the graph of interactors or the UI itself. Consistency is ensured. In Fig. 4, the end-user controls the distribution of his/her web site among the two connected platforms. One step further, we can imagine putting metamodels under the human control as well. The global picture gives rise to the notion of Mega-UI [7].

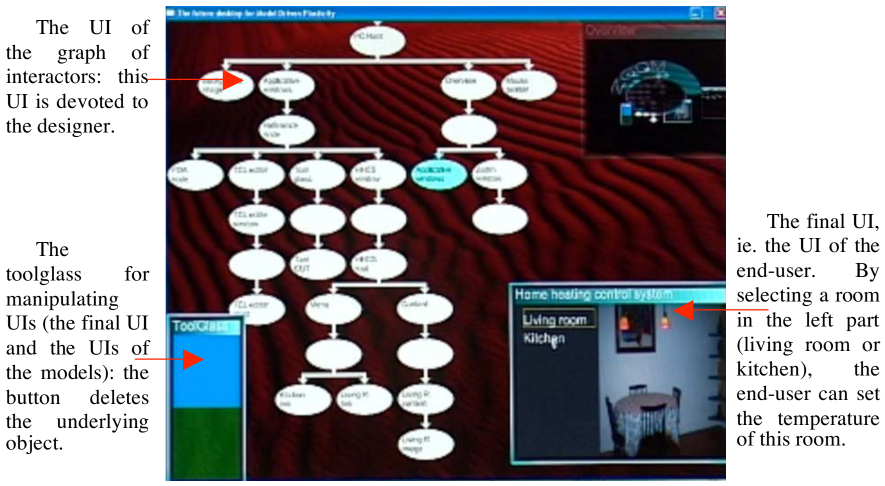
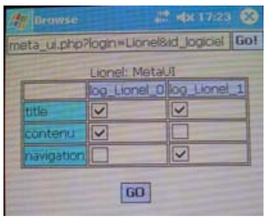

The UI of the graph of interactors: this UI is devoted to the designer.
The toolglass for manipulating UIs (the final UI and the UIs of the models): the button deletes the underlying object.
The final UI, ie. the UI of the end-user. By selecting a room in the left part (living room or kitchen), the end-user can set the temperature of this room.
Overview
ToolGlass
Home heating control system
Living room
Kitchen

Fig. 3. Additional UIs for the designer: the interactors model is made observable.



meta_ui.php?login=Lionel&id_logiciel

Lionel: MetaUI

	log_Lionel_0	log_Lionel_1
title	✓	✓
contenu	✓	☐
navigation	☐	✓

Fig. 4. Additional UIs for the end-user: the check boxes represent the mappings between the UI structure (the rows: title, content, navigation) and the connected platforms (the columns).

#2. Support for capitalization

As modeling and metamodeling takes time, capitalization is crucial for going beyond small and basic interactive systems. We need a broad capitalization of models (e.g., post-WIMP interactors such as round windows, platforms such as multi-touch tables), metamodels as well as gateways between languages as one will never fit all.

#3. Support for evaluation

In HCI, evaluation is necessary. If evaluating a novel interaction technique is feasible, how can we measure the effect of MDE in HCI. More than evaluate the performance of the approach (cost and benefit), it is important to integrate evaluation in the design process so that to step by step check whether transformations fulfills ergonomic properties and platforms constraints. This goes far beyond evaluating the performance of the generated code. To that end, we need methods and tools for benchmarking our proposals.

References

1. Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A.G., Mehandjiev, N.: Meta-design: a manifesto for end-user development. In: Communications of the ACM, Volume 47, Issue 9, pp 33-37, ACM Press, (2004)
2. Johnson, P., Wilson, S., Markopoulos, P., Pycock, J.: ADEPT-Advanced Design Environment for Prototyping with Task Models. In: Proceedings of InterCHI'93, p. 56, ACM Press, New York, Amsterdam, 24-29 April (1993)
3. Limbourg, Q.: Multi-path Development of User Interfaces, Ph.D. thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, November (2004)
4. Myers, B., Hudson, S.E., Pausch, R.: Past, Present, and future of user interface software tools, ACM Transactions on Computer-Human Interaction (TOCHI), Volume 7, Issue 1, pp 3-28, March (2000)
5. Occello, A., Casile, O., Pinna-Déry, A.-M., Riveill, M.: Making Domain-Specific Models Collaborate. In 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), pp 79-86, Montréal, Canada, 21-22 October (2007)
6. Paternò, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In: Proceedings of Interact'97, pp. 362-369, (1997)
7. Sottet, J.S., Calvary, G., Favre, J.M., Coutaz, J.: Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: Mega-UI. In: Human Centred Software Engineering: Software Engineering Architectures, Patterns and Models for Human Computer Interaction, Seffah, A., Vanderdonckt, J. and Desmarais, M. (Eds) (2008)

A Feature-Model for the UML Meta-Model Itself: Creating and Composing Lightweight Members of the UML Family

Ahmed Elkhodary[1]

[1] Department of Computer Science
George Mason University
4400 University Drive MSN 4A5
Fairfax, VA 22030 USA
aelkhoda@gmu.edu

Abstract. In this paper, we propose a new approach and suggest enhancements to the UML Profile mechanism to combine the strength of Domain Specific Modeling Languages (DSML) while avoiding their weaknesses. The approach suggests creating a feature model for the UML meta-model itself. The feature-model captures a finite list of syntactic variations in members of the UML family. In addition, the approach overcomes a very critical weakness in current DSML approaches when multiple DSMLs are used to model one system, which is inconsistency. The solution proposed enables detection of inconsistencies and interactions among profiles as soon as they are applied to a model by analyzing feature-selections made in the profiles. It also suggests extending the UML profile mechanism to include support for a transformation language (i.e. QVT Relations), which will allow profile designers to define custom cross-view relationships more efficiently. The approach benefits DSML designers as well end users. Designers can pick desired features of UML quickly and turn-off undesired ones producing lightweight languages to end users. In addition, end users can detect conflicts among UML profiles when applied to one system as soon as they are applied.

Keywords: DSML, UML, QVT, model driven engineering, domain specific languages.

Introduction: UML is a standard language for modeling object oriented software systems. It consists of highly reusable language features and widely used notations. It also provides extensibility mechanisms that tailor notations for specific domains. Thus, UML is actually regarded as a language family. A variant –or member- of the family tailors elements of the meta-model by creating new stereotypes and packaging them in a Profile [3, 5].

Unfortunately, due to the size of the UML meta-model, the process of creating UML profiles is very expensive. Exploring such a large meta-model is not only time-consuming but has also proven to cause many bad design practices. One very common bad practice is *notation fitting* [1] where a profile designer thrives to fit as much elements and views of the UML meta-model as possible rather than focusing on the needs of the domain. The final language is always very big and cumbersome for end users.

As a result, Domain Specific Modeling Languages (DSMLs) took significant attention as a rivalry approach and a more efficient alternative to UML [4]. DSML designers can define language features and tailor notations as demand arises for their specific domain. The final outcome is often a lightweight language that does domain specific tasks only but does them very well.

However, this comes to a cost. Most of the time, modeling one system requires engagement of end users from multiple disciplines with multiple DSMLs. Experience shows that for a given system, defining relationships across DSMLs is a challenge. A typical DSML would define from scratch most of its features (i.e. concepts, notations, views and cross-view relationships). As a result, to model one system, a lot of effort is required in bridging all involved DSMLs back together.

In this paper, we propose a new approach that combines the strength of lightweight DSMLs while avoiding their weaknesses. It reflects our experience in contributing to several domain-specific languages (Emergency Response, Financial Reporting, Queuing Network, Software Security, and Autonomic-SOA) as well as learning many others.

Challenges: The first challenge considered in this paper is to prevent a bad language design practice known as *notation fitting*. Rarely ever would a DSML need all language features supported by the UML meta-model. In fact, the first step in Profile design should be disabling or turning-off unneeded features of UML. We need UML profiles that give DSML designers the ability to make a "feature selection" and hide unwanted portions of the UML meta-model before they start. This challenge can be further broken down into two issues:

- The UML meta-model is not feature-based; we cannot simply disable unwanted features. As a result, any member of the UML family inherits the entire meta-model and all the views even if they are not needed.
- The UML profile mechanism is very limited with regards to hiding unwanted features in the base meta-model. This is an inherent problem caused by the absence of a coarse-granular layer of abstraction in the UML meta-model (such as a Feature-Model). Such layer would allow easy and fast projection of desired language features.

The second challenge is to reuse and maintain consistency of cross-view relationships (i.e. class-to-sequence diagram mappings, activity-to-usecase diagram mappings) in a UML profile. Today, such cross-view relationships are defined completely from scratch in the profile and mostly using OCL expressions. Besides the fact that this is totally cumbersome, when two UML profiles are applied to one system, we cannot detect at profile application time that there are inconsistencies. For instance, suppose that end users attempt to apply two UML profiles to one system: the first profile assumes that each "Use Case" maps to one "Activity" diagram; the second profile assumes that each "Action" in an activity diagram maps to one "Use Case". We can only expect inconsistencies to appear if these two profiles are applied to the models of one system.

We decompose this challenge into two issues:

- Defining cross-view relationships requires significant effort for new members of the UML family. UML does not specify relations between Views; therefore, any new member of the family needs to start from scratch. Most of the time there are well-known conventions for how certain views relate to one another; yet we always need to start from scratch.
- Cross-view relationships are often hidden features that are implicitly inherited by a UML profile unless language designers define explicit enforcement rules. UML profile designers must be able to select desired cross-view relationships explicitly to avoid inconsistencies. They also need to define their own cross-view relationships.

Proposed Solution: We introduce a new way of looking at the UML meta-model as a true product line for visual modeling (Figure 1). Notably, variations in the ***syntax*** of UML are finite and pre-anticipated at language design time. For instance, we do not expect a UML profile to introduce new diagram types. UML Profiles are rather means for adding semantic variations to fit a target domain. In other words, the syntax of a given UML model does not change when a profile is applied to it; only the semantics change.

Thus, introducing a feature model that modularizes UML into common and optional ***syntactic*** features, allows profile designers to create truly customized and lightweight members of the UML family. This prevents notation fitting from leaking into the profile design process. It also guarantees that produced languages are tailored for the needs of the domain.

- UML is a language family of a finite number of syntactic features; thus, we can define a feature model [2] to give members of the UML family the ability to turn-off unwanted features.
- UML-Profiles can be extended to make a feature-selection of the UML feature model. Unselected features will disappear during profile creation and no references are allowed to meta-elements belonging to unselected features.

The second challenge is actually a product of observing the first challenge. We noticed that cross-view relationships account for a significant portion of the syntactic variability in UML members. UML profile designers invest a lot of effort in defining such relationships between UML views because, otherwise, they render the risk of allowing inconsistent profile applications.

What we suggest to do in this research is to enumerate widely used cross-view relationships (e.g. class-to-state and class-to-sequence diagram mappings) and analyze common conventions in the community. We do not expect to find many ways for how relationships between two views are defined for instance. Therefore, capturing these common conventions of cross-view relationships and reusing them eliminates a significant amount of effort on profile designers' side.

- Cross-view relationships are significant portions of syntactic variability in UML. However, the number of widely used variations is finite. Therefore, we can provide predefined cross-view relationships (i.e. as QVT checking rules) and dedicate part of the UML feature model for picking desired relationships.

- UML-Profiles can be extended to include custom cross-view relationships (i.e. custom QVT-Relations checking rules [6]) in case predefined ones are not sufficient for the domain.

This part of the approach addresses the second challenge directly. That is, when end users apply multiple profiles to one system, inconsistencies and interactions among profiles can be detected automatically. For instance, if applied profiles select mutually exclusive features of the UML meta-model, the feature model can detect it. In addition, if custom QVT-Relations checking rules are defined in the profiles, we can use any static interaction analysis technique, such as critical pair analysis [7], to detect conflicts and dependencies among the rules. This way, end users from multiple disciplines can use the right profiles and avoid inconsistencies from the beginning.

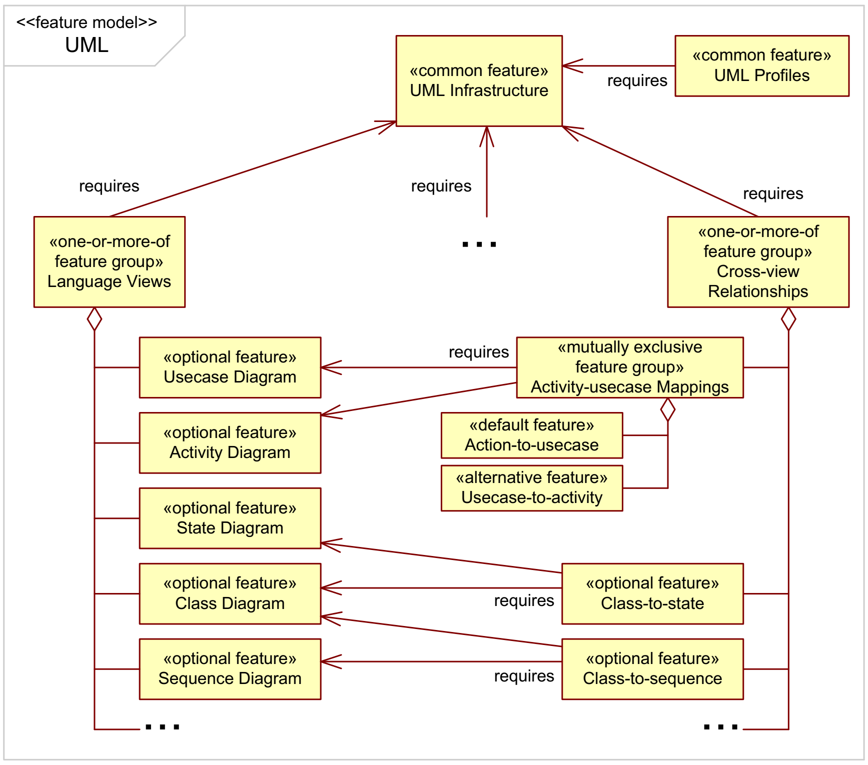

<<feature model>>
UML
«common feature» UML Infrastructure
«common feature» UML Profiles
requires
requires
requires
requires
«one-or-more-of feature group» Language Views
«one-or-more-of feature group» Cross-view Relationships
«optional feature» Usecase Diagram
«optional feature» Activity Diagram
«optional feature» State Diagram
«optional feature» Class Diagram
«optional feature» Sequence Diagram
«mutually exclusive feature group» Activity-usecase Mappings
«default feature» Action-to-usecase
«alternative feature» Usecase-to-activity
«optional feature» Class-to-state
«optional feature» Class-to-sequence
requires
requires
requires

Figure 1: A MOF based feature model of the UML product line *(PLUS [2] notation)*.

Conclusion: In this paper, we proposed an approach that reduces the expense of creating UML profiles and prevents bad language design practices, such as notation fitting, from leaking into it. The approach suggests creating a feature model for the UML meta-model itself. The feature model captures a finite number of syntactic variation points, which allows DSML designers to pick desired features of UML quickly and turn-off undesired ones.

The approach also benefits DSML end users in many ways. First, by providing lighter languages that can get the job done; and second, by enabling automated detection of conflicts among UML profiles when applied to one system as soon as they are applied.

In addition, the proposed approach overcomes a very critical weakness in current DSML approaches when multiple DSMLs are used to model one system: inconsistency of language features. The solution proposed enables detection of inconsistencies and interactions among profiles by analyzing feature-selections made and QVT specifications defined in the profiles.

As mentioned earlier, UML is a standard modeling language. A wide audience base had learned the syntax of UML and used it as a documentation tool. However, the main weakness of UML is how big it is today. Our goal is to enable large-scale and systematic reuse of meta-modeling efforts. Model driven engineering is still immature and a lot of experimentation effort is still going on at the meta-modeling level. It is important to be able to reuse and reduce technical meta-modeling efforts and shift focus towards domains' needs. Domains shall drive meta-modeling efforts and not the other way around.

References

1. France, R. B., Ghosh, S., Dinh-Trong, T., and Solberg, A.: Model-Driven Development Using UML 2.0: Promises and Pitfalls. Computer 39, 2 (Feb. 2006), 59.
2. Gomaa, H.: Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures. Software Product Line Conference, 2006 10th International, vol., no., pp. 218-218, 21-24 Aug. 2006.
3. Kleppe, A. Warmer, J. Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Object Technology Series. 2003.
4. Greenfield, J. Short, K. Cook, S. Kent, S. Software Factories. Wiley. 2004
5. Selic, B. A Systematic Approach to Domain-Specific Language Design Using UML. IBM Canada. 2007.
6. MOF QVT Final Adopted Specification. Object Management Group. 2007.
7. Ehrig, H. Ehrig, K. Prange, U. Taentzer, G. Fundamentals of Algebraic Graph Transformation. Springer-Verlag Berlin Heidelberg. 2006.

Validation challenges in model composition: The case of adaptive systems[*]

Freddy Munoz, Benoit Baudry

INRIA Bretagne Atlantique
Campus de Beaulieu F-35042, Rennes cedex
{fmunoz,bbaudry}@irisa.fr

Abstract. Model Driven Engineering helps dealing with complexity by promoting models as abstraction units. Aspect Oriented Modeling helps separating concerns that crosscut across different models. MDE and AOM have well identified challenges that need to be addressed. However, there are new challenges that appear when combining both techniques. In this paper we present the challenges that appear when validating the model composition in the context of MDE and AOM applied to adaptive systems.

Keywords: Model Driven Engineering, Aspect Oriented Modeling, Model Composition Validation, Model Validation, Model Driven Engineering for Adaptive Systems.

1 Introduction

Model Driven Engineering (MDE) promotes abstraction as a basis for managing complexity. MDE proposes the systematic use of models as primary engineering artifacts. Such models can have a variety of natures and range in abstraction and complexity. Models from higher abstraction level are refined (transformed) into lower levels until the implementation. Besides, models can be transformed from one domain to another in order to ease the resolution of a defined problem.

Aspect Oriented Modeling (AOM) helps separating crosscutting concerns at model level by encapsulating them into different modeling dimensions referred as *aspect*. AOM enables a clear modularization of the different concerns constituting a design model. It also allows designers to reason about each concern separately, and later composed into a global model.

Research challenges have been widely identified for AOM and MDE [11]. Model transformation testing[5], model verification and validation [9], and AOM weaving mechanism definition [2] are just a few examples of the challenges faced by these technologies.

Nevertheless, not all the challenges have been identified as far as validation in MDE and AOM is concerned. Challenges regarding the composition and refinement of aspect models need to be identified. This is critical to ensure that composed

[*] This work was partially funded by the DiVA project (EU FP7 STREP)

models will perform as expected, and therefore, their refined implementation will do so [4, 8]. This is a fundamental issue for the adoption of AOM as a mechanism for separation of concerns and MDE as a complexity coping mechanism.

In this paper we present the challenges that arise from the validation of model composition. We specially address the challenges that arise in the case of adaptive systems, where models are used to abstract from the executing platform and aspects represent the dynamic variability of the system. Such challenges range from the combinatorial explosion produced by the composition order of different aspects, to the specification of the model resulting from the composition.

The remainder of this paper is organized as follows. Section 2 presents MDE and AOM applied to adaptive systems. Section 3 presents the challenges that arise when validating AOM and MDE in the context of adaptive systems. Finally, section 4 concludes.

2 MDE and AOM for adaptive systems

Designing, developing, maintaining and executing adaptive systems is very complex and error prone. Model Driven and Aspect Oriented techniques can help dealing with this complexity. In this section we present the contribution of MDE and AOM to handle the complexity when dealing with adaptive systems.

Adaptive systems are software systems capable of change their internal structure and behavior in response to changes in their environment [1]. They are typically deployed in heterogeneous computing devices ranging from mobile devices such as phones or PDAs to large computer systems. Generally, several variation points are defined in order to develop an adaptive system. Each variation point represents a different option in the system implementation that might be chosen to adapt the system. The selection of different variation points to derive the adapted system leads to a huge number of possible configurations. Reasoning over that huge set of configurations to choose the best possible configuration to adapt is too time consuming because of the large number of evaluations needed. Moreover, the adaptation logic relies on reconfiguration policies that are generally complex low-level and hand-written in the application producing large and complex reconfiguration files. These factors make the construction, execution and maintenance of adaptive systems highly complex. MDE and AOM help dealing with this complexity by the meaning of abstraction and separation of concerns [19].

MDE techniques provide the means to automate and optimize the creation of reconfiguration scripts. Besides, MDE helps abstracting from the target platform by defining models independent of target devices and technologies. Models representing the system in execution (models at runtime) help to manage the execution at more abstract level; therefore, they enable designers to reason about the system properties and adaptation logic at higher level.

Aspect-Oriented modeling techniques [10, 13, 15, 18] help encapsulating distinct variation points into aspects separated from the base model functionalities. Different aspects might be composed with the base model in order to obtain different configurations. This reduces the reasoning space to a limited number of aspects, therefore avoiding the combinatorial explosion due to different variants.

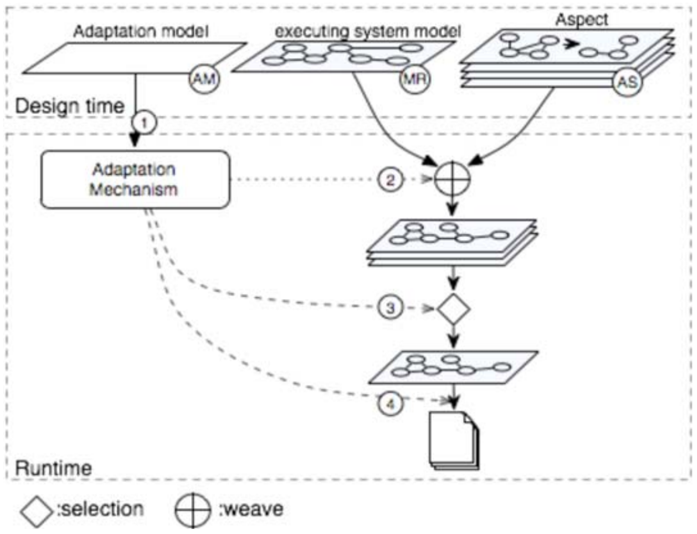


Fig. 1. Overall MDE/AOM approach for adaptive systems

Figure 1, presents an overall approach for adapting systems by using MDE and AOM techniques. At design-time, the application base (AM) and variant architecture (AS) models are designed. At this time, the adaptation model, which states when, and how to adapt is built. At runtime, the adaptation mechanism processes the adaptation model in order to adapt when needed (1). When an adaptation is required, the adaptation mechanism chooses (driven by the adaptation model) a set of aspects (variants) and weaves them into the base (2). This weaving results in multiple models that could be used to adapt the executing system. The adaptation mechanism chooses only one model (3) and then automatically generates the reconfiguration scripts used to adapt the executing system (4).

3 Challenges

MDE and AOM can help dealing with the complexity involved in the life cycle of adaptive systems. However, their usage raises new challenges regarding the validation of model composition. In the following we summarize the challenges and research questions related to the validation in the context of adaptive system.

Validation of composed models (3 in figure 1): The selection of the best possible configuration is critical for adaptive systems. The aspects modifying the base configuration must produce configurations that will not break down the system and that response in the best possible way to environmental changes . Therefore, it is crucial to ensure that the composed models fit the adaptation requirements. A model that is valid with respect to the adaptation requirements will lead to a correct adaptation. Testing techniques such as combinatorial testing [12, 24] and search based testing [17] may help validating that the composed model fits the adaptation requirements. Such testing techniques provide the means to explore a huge adaptation space and test whether the chosen configurations are fitted adaptations. Formal behavioral specification techniques [25] may also be useful to verify that the composed models will fit the adaptation invariants.

It is an intuition to think that, if the chosen aspects are valid and the base model is valid, then the composed model will be valid. Is this always true? How can it be ensured? These questions are fundamental because they may allow to reason about the aspects and model validity separately and then compose a valid model. However, event if this is true, it is still an issue how to validate the aspect models and how to validate the base model. Moreover, the composition engine must also be valid in order to produce valid compositions.

Combinatorial explosion of composed models (2 in figure 1): The weaving of different aspects leads to different composed models. Likewise, the weaving order of aspects may also generate different models. Therefore, the rate of models resulting from the composition of different aspects and composition orders grows exponentially with the amount of aspects to weave. Moreover, there is no assurance that the composed models will be valid or fits the adaptation requirements. This is a serious issue because it is necessary to validate a huge amount of composed models. Such validation may consume an unrealistic amount of time.

Aspects effects and interactions (2, 3 in figure 1): Different aspects have different effects on the base model. Some aspects may add new system properties whereas others may remove them. The effect of aspects may depend on the order in which they are weaved. For instance, consider two aspects, one relating the communication (C) and another relating the security concerns (S). When C is woven first, the system network response is very short, whereas when S is woven first, the system network response is slower but more secure. Some weaving orders or combinations of aspects could add or remove system properties unexpectedly. Controlling the emergence of properties introduced/ removed by aspects is very important to ensure that the composition result will be valid and aspects will perform as expected. Similar problems have been studied at code level. Solutions such as the specification of the aspect behavior [3, 6, 20] are used to increase the maintainability of aspect-oriented programs. The properties added/ removed by the aspects are controlled by the specifications.

Aspects adapting a system will interact in a variety of ways. Some interactions may include/ exclude the weaving of some aspects; other interactions may interfere or partially invalidate the effect of aspects over the system. Moreover, interactions and the effect of aspects may change according to the target model they are weaved into. Therefore, detecting aspects interactions in advance is very important to avoid composition conflicts and know before hand the aspects dependencies [22, 23]. This issue is related to critical pair analysis [7, 16, 21] in which the conflicts between different interacting features are detected via graph analysis. Critical pair analysis detects functional inclusive/ exclusive aspects configurations. However, interaction issues can be beyond functional interactions. Aspects can have a qualitative impact over the system, for instance making the quality of service better or worst. At code level, the characterization of interactions could be used to determine patterns of interactions for instance to detect aspect interferences [14].

Runtime / Design time validation: Since adaptation happens at runtime, adaptive systems have to respond to hard time and hardware constraints when adapting, a fundamental question is how much of the validation and analysis can be done statically? The ideal will be to calculate at design time *all* the possible interactions and effect of aspects and their possible weaving orders. However, this may not be possible due to the huge amount of possible weaving orders

An idea to cope with these issues is defining contracts on the aspect models. These contracts may be an abstract specification of the effect of the aspects over the system and the interactions between aspects. For instance, they can explicitly declare that an aspect will increase the overall system security but making it slower. They could also allow us to calculate optimal and valid weaving orders. By specifying include/ exclude relations between aspects. Moreover, they may be helpful to detect interactions conflicts at design time, thus saving some computation time when adapting at runtime. The abstract description of the aspects' effect will help determining whether aspects may be valid or not in relation to a base model.

4 Conclusions

In this paper we have identified the challenges that appear when validating the model composition in the context of adaptive systems. Issues such as the weaving order of aspect models, interaction issues and the validation of the composed model are not trivial. Tackling these issues is fundamental to assess the usage of MDE and AOM.

We have pointed out possible ways to address the validation challenges presented here. We specially suggest the definition of contracts to tackle several challenges and ease the solution of others. In future work we will explore how contracts must look like, which information they must contain and how to successfully use them at design time and runtime.

References

1. Aksit, M. and Z. Choukair, *Dynamic, Adaptive, and Reconfigurable Systems Overview and Prospective Vision*, in *23rd Int'l Conf. Distributed Computing Systems Workshops (ICDCSW)*. 2003: Providence, Rhode Island USA.
2. Aldawud, O., et al. *AOM: 11th International Workshop on Aspect-Oriented Modeling*. in *10th International Conference On Model Driven Engineering Languages And Systems*. 2007. Nashville, TN, USA.
3. Aldrich, J., *Open Modules: Modular Reasoning About Advice*, in *ECOOP 2005: 19th European Conference on Object-Oriented Programming*. 2005: Glasgow, UK.

4. Baleani, M., et al. *Correct-by-construction transformations across design environments for model-based embedded software development*. in *Design, Automation and Test in Europe, 2005. Proceedings*. 2005.
5. Baudry, B., et al. *Model Transformation Testing Challenges*. in *IMDT workshop in conjunction with ECMDA-FA 06*. 2006. Bilbao, Spain.
6. Clifton, C. and G.T. Leavens. *Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning*. in *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*. 2002. Enschede, The Netherlands.
7. Detlef, P., *Hypergraph rewriting: critical pairs and undecidability of confluence*, in *Term graph rewriting: theory and practice*. 1993, John Wiley and Sons Ltd. p. 201-213.
8. Dion, B., *Correct-by-Construction Methods for the Development of Safety-Critical Applications*. SAE transactions, 2004. **SAE Paper # 4AE-129**(SAE World Congress).
9. Faivre, A., S. Ghosh, and A. Pretschner. *MoDeVVA: 5th workshop on Model Driven Engineering, Verification, And Validation: Integrating Verification And Validation In MDE*. in *1st International Conference on Software Testing, ICST 2008*. 2008. Lillehammer, Norway.
10. France, R., et al., *Providing Support for Model Composition in Metamodels*, in *EDOC'07: 11th Int. Enterprise Computing Conference*. 2007: Annapolis, Maryland, USA.
11. France, R. and B. Rumpe, *Model-driven Development of Complex Software: A Research Roadmap*, in *2007 Future of Software Engineering*. 2007, IEEE Computer Society.
12. Grindal, M., *Handling Combinatorial Explosion in Software Testing*, in *Computer Science*. 2007, University of Skövde and Enea: Skövde, Sweden.
13. Jayaraman, P., et al., *Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis*, in *Model Driven Engineering Languages and Systems*. 2007. p. 151-165.
14. Katz, S., *Diagnosis of Harmful Aspects Using Regression Verification*, in *FOAL: Foundations Of Aspect-Oriented Languages*, C.C.a.R.L.a.G.T. Leavens, Editor. 2004: Lancaster, UK.
15. Lahire, P., et al., *Introducing Variability into Aspect-Oriented Modeling Approaches*, in *Model Driven Engineering Languages and Systems*. 2007. p. 498-513.
16. Leen, L., E. Hartmut, and O. Fernando, *Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs*. Electron. Notes Theor. Comput. Sci., 2008. **211**: p. 17-26.
17. McMinn, P., *Search-based software test data generation: a survey*. Software Testing Verification Reliability, 2004. **14**: p. 105 -- 156.
18. Morin, B., O. Barais, and J.-M. Jézéquel. *Weaving Aspect Configurations for Managing System Variability*. in *Second International Workshop on Variability Modelling of Software-intensive Systems*. 2008. Essen, Germany.
19. Morin, B., et al. *An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability*. in *11th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2008. Toulouse, France
20. Munoz, F., B. Baudry, and O. Barais, *Improving Maintenance in AOP Through an Interaction Specification Framework*, in *ICSM08: 24th IEEE International Conference on Software Maintenance*. 2008, IEEE Computer Society: Beijing, China.
21. Praveen, J., et al., *Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis*. Model Driven Engineering Languages and Systems, 2007: p. 151--165.
22. Sanen, F., et al. *Classifying and documenting aspect interactions*. in *ACP4IS06 Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. 2006. Bonn, Germany.
23. Truyen, E., et al., *Support for distributed adaptations in aspect-oriented middleware*. Proceedings of the 7th international conference on Aspect-oriented software development, 2008: p. 120-131.
24. Yilmaz, C., M.B. Cohen, and A.A. Porter, *Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces*. IEEE Transactions on Software Engineering, 2006. **32**: p. 20--34.
25. Zhang, J. and B.H.C. Cheng, *Model-based development of dynamically adaptive software*, in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. 2006, ACM: Shanghai, China. p. 371--380.

A Tentative Analysis of the Factors Affecting the Industrial Adoption of MDE

Adrian Rutle[1] and Alessandro Rossini[2]

[1] Bergen University College, p.b. 7030, N-5020 Bergen, Norway aru@hib.no
[2] University of Bergen, p.b. 7803, N-5020 Bergen, Norway rossini@ii.uib.no

Abstract. Model-Driven Engineering (MDE) has been widely accepted by the academy as the new vision for software development. However, as with other new approaches, its industrial acceptance is still very limited. We believe that many of the challenges that are impairing this industrial adoption can be related to the lack of good tool support. In this paper we will focus on two areas where present-day tools and frameworks are not quite sufficient: developing metamodels and version control.

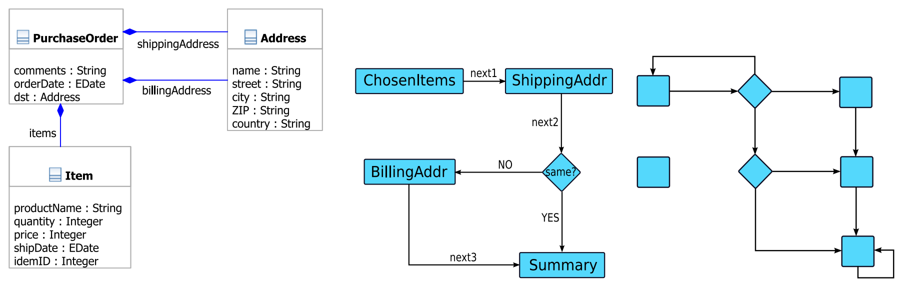
1 Introduction

An important step in the adoption of MDE, in general, lies in the development of techniques and tools that support developing models and model transformations which are both precise and intuitive. In fact, developing metamodels together with model transformations for a given problem domain can be difficult and time-consuming. The ambiguity of models which is introduced by informal or quasi-formal modelling languages leads to the generation of code and models which are either not easily readable or not as intended by the designer, or both. This motivates for the usage of formal modelling languages for developing (meta)models. However, current formal modelling languages are complex and error-prone, have a textual syntax, and generally lack good tool support. In addition, they are based on First Order Logic (FOL) which does not always match the internal logic of modelling. This is because models in software engineering are graph-based while FOL is string-based [1]. Our claim is that a formal diagrammatic approach for (meta)models and model transformations is worth investigating. The aim is to combine the intuitiveness of graphical languages with the semantic rigour of formal approaches. This challenge is further discussed in Section 2.

Moreover, present-day tools give software engineers means to simplify the development process. Therefore, convincing developers to start development with modelling requires tool support to provide features of the same quality as most IDEs provide. One of those features – version control – is of crucial relevance in high-quality software development processes. However, current MDE tools offer only limited support for version management. Typically the problem is addressed using a "lock-modify-unlock" solution. On the contrary, conventional version control systems such as Subversion are based on the "copy-modify-merge" approach, but they are focused on the management of text-based files like application code. That is, difference calculation and conflict detection are based on a per-line text comparison. The structure of models is graph-based, rather than text- or tree-based, hence existing *differencing* techniques are not suitable for MDE. In Section 3, we analyse the problem and outline a possible solution.

2 Difficulties in Developing Metamodels

In our opinion, one of the important challenges in MDE lies in the lack of methods, languages, frameworks and tools to design precise metamodels. However, specifying metamodels may not be a straight forward task and may introduce a big challenge for engineers who are not trained to design (meta)models. It requires a thorough understanding of metamodelling techniques and constraints as well as an *intuitive* framework which is based on a *formal*, *expressive* language. To face this challenge, we believe that any solution must have at least two components: (1) a repository of well-proven metamodels for successful systems, that the developers may reuse or consider as guideline, and (2) a formal framework with a diagrammatic logic for the specification of these metamodels. In this section, we will explain the difficulty of developing metamodels with an example and outline a solution to tackle this difficulty.



(a) UML class diagram of the purchase order example, adopted from [2]

(b) Model of the wizard for the registration of purchase orders

(c) A valid model of an arbitrary wizard

Fig. 1: The purchase order example with a simple wizard.

We adopt the example of purchase orders (Fig. 1a) from [2]. Suppose that we want to build a wizard for the registration of purchase orders. The wizard may be fairly generic, such as a desktop- or a web-application. For simplicity, we suppose that users start from a page that shows the chosen items. The wizard consists of a set of pages (represented by □) which are viewed through a finite number of steps in order to register a purchase order. We use action links (→)

and decision nodes (◊) to model the navigation from one page (or decision) node to another.

We require the wizard to satisfy the following constraints:

- each action link must have a source and a target page (or decision);
- a page may stand alone, or be the source of an action link, or the target of multiple action links;
- a decision node can not stand alone, may be the source of multiple action links, and must be the target of only one action link.

Fig. 1b and Fig. 1c show two wizards satisfying these constraints.

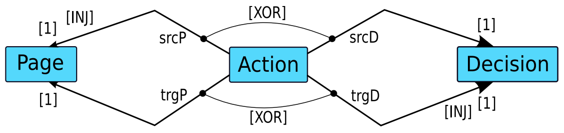


Fig. 2: The metamodel for our wizard system specified in the DPL framework

We have used the formal framework Diagrammatic Predicate Logic (DPL) [1,3,4,5], to outline a solution for our example (see Fig. 2). We give a short and informal explanation of the semantics of some of the constraints here. References above can be consulted for a formal description of the constraints. The semantics of the constraints `[1]`, `[XOR]` and the filled circles on the arrows `srcP` and `srcD` enforce that each instance of `Action` *must have exactly one* source which is *either* an instance of a `Page` *or* a `Decision`, *but not both.* Moreover, the constraint `[INJ]` on the arrow `srcP` means that an instance of `Page` can be the source of *only one* instance of `Action` (mono or injective). Finally, the filled arrow-caps of `trgD` and `srcD` mean that each instance of `Decision` *must* be the source and the target of some instance of `Action` (cover or surjective).

For comparison reasons, consider the same metamodel specified as an Eclipse Modeling Framework (EMF) model. The differences will not only be in the visualisation of the metamodel, but also in the way in which the constraints are specified. Constraints in EMF models (which are specified in OCL or Java) will involve writing an Eclipse plug-in based on the EMF Validation Framework, or will involve the usage of the `EValidator` API of EMF. In both cases, writing code will be necessary to express the properties of the models, something which we consider as against the principles of MDE. In our opinion, these processes lack means for reasoning about the constraints without digging into the details of the validation code. On the contrary, the DPL framework allows for this kind of reasoning without the need for hand-written code by providing a diagrammatic logic.

3 Version Control

The literature encompasses several works related to the problem of model versioning: [6,7] for the difference calculation, [8,9] for the difference representation and [10,11] for the conflict detection, to cite a few. However, MDE tools available nowadays do not implement those features in an effective way. For example EMF Compare and DSMDiff are two model differencing tools which are based on a similar technique. The difference calculation is divided in two phases. The first is the detection of model mappings, where all the elements of the two input models are compared using metrics like signature matching and structural similarity. The second phase is the determination of model differences, where all the additions, deletions and changes are detected. While this technique has the great benefit of being general, the algorithm is too inefficient to be used in a production environment.

On the other hand, it is not feasible to implement more efficient differencing techniques with the current OMG standards. Our claim is that one of the main problems is the identification of model elements as defined in the XMI specification [12]. The XMI standard provides two attributes to identify model elements: `xmi.id`, used to reference elements inside the XMI document, and `xmi.uuid`, used to reference elements globally and persistently. Since the former can be useful only in a limited number of cases, we will focus our discussion on the latter. Universally Unique Identifiers (UUIDs) guarantee uniqueness across space and time, without a centralised registration process. As such, once generated, the UUIDs are supposed to be read-only identifiers for all the MDE tools handling them. Unfortunately, as observed in [13], this is not the case of current systems: in most cases, exporting and importing an XMI document from one MDE tool to another causes UUIDs to be ignored or regenerated, but this is not the expected behaviour defined in the XMI standard. However, even in the case where MDE tools all behave as expected, the main drawback is that the standard itself does not constrain the way UUIDs have to be generated. Even though OMG suggests [14] as a reference algorithm, every tool may implement the identifier generation in a radically different way.

If we want to design efficient differencing techniques based on unique identifiers, we need a different approach to element identification specifically designed for MDE. We believe that an ideal identifier generation algorithm should have the following properties: (1) syntactically equivalent model elements should be assigned the same identifier, and (2) the generation of an identifier for a model element should be based on its components. In other words, the modification of the components (e.g. the attributes and/or methods of a class) should lead to the regeneration of the element's identifier. Hence, the differencing technique would just compare the identifiers of each model element before comparing its components; only in the case of a difference in the elements' identifiers it would proceed with the comparison of their components. In addition, having a mature XMI standard with MDE-specific identifiers, would also be of benefit for tools integration, model checking, and model transformations.

4 Conclusion

The adoption of MDE by the industry will be a considerable challenge unless techniques, tools and standards for (meta)modelling has become ready for production environment. That is, MDE tools should provide support for development, maintenance and exchange of (meta)models and model transformations of the same quality as the features provided by code-centric IDEs. To achieve this, a closer collaboration between academy, OMG, tool providers and industry is necessary.

References

1. Wolter, U., Diskin, Z.: Generalized Sketches: Towards a Universal Logic for Diagrammatic Modeling in Software Engineering. In: ETAPS 2007: European Joint Conferences on Theory and Practice of Software, ACCAT Workshop. (To appear)
2. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0 (2^{nd} Edition). Addison-Wesley Professional (2006)
3. Rutle, A., Wolter, U., Lamo, Y.: Generalized Sketches and Model Driven Architecture. Technical Report 367, Department of Informatics, University of Bergen, Norway (2008)
4. Rutle, A., Wolter, U., Lamo, Y.: A Formal Approach to Modeling and Model Transformations in Software Engineering. Technical Report 48, Department of Informatics, University of Turku, Finland (2008)
5. Rutle, A., Wolter, U., Lamo, Y.: A Diagrammatic Approach to Model Transformations. In: EATIS 2008: Euro American Conference on Telematics and Information Systems. (To appear)
6. Brun, C., Musset, J., Toulmé, A.: EMF Compare. `http://wiki.eclipse.org/index.php/EMF_Compare`.
7. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems **16**(4, Special Issue on Model-Driven Systems Development) (2007) 349–361
8. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6**(9, Special Issue on TOOLS Europe 2007) (October 2007) 165–185
9. Rivera, J.E., Vallecillo, A.: Representing and Operating with Model Differences. In: TOOLS Europe 2008: Objects, Components, Models and Patterns, 46^{th} International Conference. Volume 11 of Lecture Notes in Business Information Processing., Springer (2008) 141–160
10. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electronic Notes in Theoretical Computer Science **127**(3) (2005) 113–128
11. Cicchetti, A., Rossini, A.: Weaving Models in Conflict Detection Specifications. In: SAC 2007: 2007 ACM Symposium on Applied Computing, ACM (2007) 1035–1036
12. Object Management Group: XML Metadata Interchange. (December 2007) `http://www.omg.org/cgi-bin/doc?formal/2007-12-01`.
13. Alanen, M., Porres, I.: Model Interchange Using OMG Standards. In: EUROMICRO 2005: 31^{st} EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE Computer Society (2005) 450–459
14. Internet Engineering Task Force: RFC4122: A Universally Unique IDentifier (UUID) URN Namespace. (July 2005) `http://www.ietf.org/rfc/rfc4122.txt`.