Model-driven software evolution: An alternative research agenda

Tom Mens[1], Xavier Blanc[2], and Kim Mens[3]

[1] Service de Génie Logiciel, Université de Mons-Hainaut
Av. du champ de Mars 6, 7000 Mons, Belgium
tom.mens@umh.ac.be
[2] LIP6
Paris, France
xavier.blanc
[3] UCL
kim.mens

1 Introduction

In the realm of software engineering, we are witnessing an increasing momentum towards the use of models as primary artefacts for developing software systems. This gave rise to a new paradigm commonly referred to as model-driven software engineering [1]. This use of models promises to cope with the intrinsic complexity of software-intensive systems by raising the level of abstraction, and by hiding the accidental complexity of the underlying technology as much as possible [2]. The use of models thus opens up new possibilities for creating, analysing, manipulating and formally reasoning about systems at a high level of abstraction. Evolution of models can be achieved by relying on sophisticated mechanisms of model transformation [3]. Model transformation techniques and languages enable a wide range of different automated activities such as translation of models (expressed in different modelling languages), generating code from models, model refinement, model synthesis or model extraction, model restructuring, etc.

Most of the research in model-driven engineering (MDE) focuses on the vertical dimension (see Figure 1), such as how to generate code from models (forward engineering), how to obtain high-level abstractions from code (reverse engineering), and how to synchronise and co-evolve both levels of abstraction. This is for example very clear in the OMG's MDA approach [4] that explicitly addresses the problem of migrating from platform-independent models (PIM) to platform-specific models (PSM). It is also an integral part of Microsoft's software factory approach targeted towards domain-specific languages [5].

2 Research Challenges

van Deursen *et al.* [6] have established a research agenda of challenges to be addressed in this vertical dimension. The goal of this paper, is to shed an alternative view on the problem, by identifying those challenges that have to do with the horizontal dimension instead, i.e., all activities related to the process of model evolution.

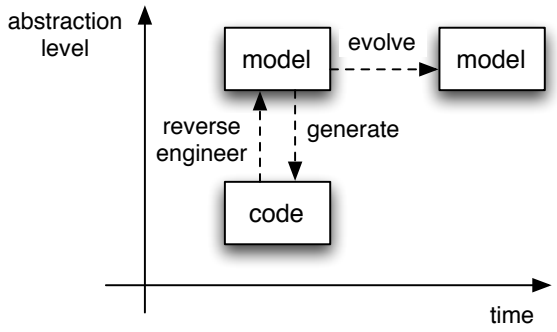

Fig. 1. Two orthogonal dimensios of model transformation, according to [3]. The focus of this article is on the horizontal dimension.

2.1 Model independence

How can we represent and manipulate different types of models in a uniform way, without needing to change the infrastructure (tools, mechanisms and formalisms) for reasoning about these models? Such model independence is of scientific as well as practical importance, because we want our solutions to be sufficiently generic, in order to be applicable beyond mere software models. Indeed, we want to be able to support an as wide range of models as possible. [7] have illustrated the feasibility of achieving such model independence, by implementing a generic model transformation that can be used it to transform domain-specific models.

2.2 Metamodel evolution

As pointed out by various authors [8, 9], not only models evolve, but so do their metamodels or languages in which the models are expressed, though at a lower pace. In order to ensure that models do not become obsolete because their languages have evolved, we need mechanisms to support the co-evolution between models and their associated metamodels. In a similar vein, the model transformation languages may evolve in parallel with the model transformations being used, so we also need to support co-evolution at this level.

2.3 Model quality

With respect to model quality, several research questions naturally arise. The first question is how to provide a precise definition of model quality. A model can have many different non-functional properties or quality characteristics that may be desirable (some examples are: usability, readability, performance and adaptability). It remains an open challenge to identify which qualities are necessary and sufficient for which type of

stakeholder, as well as how to specify these qualities formally, and how to relate them to one another.

A related logical question concerns how we can objectively measure, predict and control the quality of models during model evolution. One possible solution is by resorting to *model metrics*, the model-level equivalent of software metrics [10]. The challenge here is to define model metrics in such a way that they correlate well with external model quality characteristics. A more pragmatic way of assessing model quality is by resorting to what we call *model smells*. These are the model-level equivalent of bad smells, a term originally coined by Kent Beck to refer to structures in the code that suggest opportunities for improvement. Typical model smells have to do with redundancies, ambiguities, inconsistencies, incompleteness, non-adherence to design conventions or standards, abuse of the modelling notation, and so on. The challenge is to come up with a comprehensive and commonly accepted list of model smells, as well as tool support to detect such smells in an automated way.

2.4 Model improvement

In order to improve model quality, we need to resort to the technique of model refactoring, the model-level equivalent of program refactoring. An important point of attention is the study of the relation between model metrics and model refactoring. In particular, we need to assess to which extent model refactorings affect metric values. [11–13] have started to explore this problem, though mainly at code level. A formal specification of model refactoring is required to address these issues at model level.

In a similar vein, we also require a precise understanding of the relation between model smells and model refactoring, in order to be able to suggest, for any given model smell, appropriate model refactorings that can remove this smell. The other way around, we need to ensure that model refactorings effectively reduce the number of smells.

In recent work, we have started to explore the formalisms of graph transformation and description logics as underlying foundations for model refactoring [14–16]. An important point of attention here is the need for a composition mechanism that allows us to reason about composite refactorings in a scaleable way. We also need to study to which extent the formalisms allow us to verify that a given transformation preserves certain properties (e.g. structure-preserving, behaviour-preserving, quality-preserving).

2.5 Model inconsistency

In a model-driven development approach, inconsistencies inevitable arise, because a system description is composed of a wide variety of diverse models, some of which are maintained in parallel, and most of which are subject to continuous evolution. Therefore, there is a need to formally define the various types of model inconsistencies in a uniform framework, and to resort to formally founded techniques to detect and resolve these model inconsistencies. A prerequisite for doing so is to provide traceability mechanisms, by making explicit the dependencies between models. In recent work, we have proposed to manage model inconsistencies by relying on the formalisms of graph transformation [17, 18] and description logics [19, 20]. In interesting way to maintain

the consistency between different model views seems to be the use of triple graph grammars [21].

2.6 Conflict analysis

Another important challenge has to do with the ability to cope with conflicting goals. During model evolution, trade-offs need to be made all the time:

- When trying to improve model quality, different quality goals may be in contradiction with each other. For example, optimising the understandability of a model may go at the expense of its maintainability.
- In the context of inconsistency management, for a given model inconsistency there may be different inconsistency resolution strategies that are in mutual conflict.
- In the context of model refactoring, a given model smell may be resolved in various ways, by applying different model refactorings. Vice versa, a given model refactoring may simultaneously remove multiple model smells, but may also introduce new smells.

It should be clear from the discussion above that uniform formal support for analysing and resolving conflicts during model transformation is needed. Recently, we have started to explore formal techniques based on critical pair analysis and sequential dependency analysis to detect and reconcile conflicting concerns [17, 18, 15].

2.7 Collaborative modelling

Another important challenge in model-driven software evolution is how to cope with models that evolve in a distributed collaborative setting? This naturally leads to a whole range of problems that need to be addressed, such as the need for model differencing, model versioning, model merging or model integration, model synchronisation, and so on. These issues are under active study by various research groups [22] and are being integrated as services in the so-called ModelBus framework [23].

2.8 Choice of formalism

A currently unresolved question is which formalisms are most suited to specify and reason about model evolution, model quality and model consistency. Each type of formalism will have its own advantages (in terms of the formal properties it can express). Given the current state-of-the-art, we can primarily discern three types of formalisms: based on graph transformation, based on logic, and based on tree rewriting.

Graphs are an obvious choice for representing models, since most types of models have an intrinsic graph-based structure. As a direct consequence, it makes sense to express model evolution by resorting to graph transformation approaches. Many useful theoretical properties exist, such as results about parallellism, confluence, termination and critical pair analysis. Model independence of the approach (i.e., the ability to apply it to a wide range models) can be achieved by resorting to a technique similar to metamodelling in model-driven engineering: each graph and graph transformation needs to

conform to the constraints imposed by a type graph, used to describe the syntax of the language.

If we want to reason about the behaviour of software, logic-based formalisms seem to be the most interesting choice. Their declarative or descriptive semantics allow for a more or less direct representation of the behaviour of software artefacts. One of the logic formalisms that we will explore is the use of description logics [24]. This is a family of decidable logic-based knowledge representation formalisms with expressiveness as the characterising factor. While all state-of-the-art description logics are decidable, the higher the expressiveness, the higher the computational complexity of reasoning tasks becomes. Tool support for different variants of description logics is readily available (e.g., RACER), which facilitates practical experiments with these formalisms. In the past we have already experimented with description logics in the context of consistency maintenance of UML models [19] and model refactoring [20]. Other logic-based variants, such as temporal logics, may also be suitable for reasoning about model evolution.

A challenge is how to combine these different formalisms toghether into a single uniform framework. Initial results are available that combine graph transformation techniques with a logic-based approach in order to support software evolution (Rtschke and Schrr, 2006).

2.9 Scaleability and incrementality

An important practical challenge is the ability to provide tool support that is scalable to large and complex models, as used in industry. This requirement imposes important restrictions on the underlying formalisms to be used. As an example, consider existing approaches to formal verification and model checking, typically based on some variant of temporal logics. Their main problem is that they only allow to verify properties on a model in its entirety. Even a smal incremental change to this model typically requires reverification of the proven properties for the entire model. With an incremental approach, the effort needed to reverify a given property would become proportional to the change made to the model. It is therefore essential to find truly incremental techniques, in order to close the gap between formal techniques and pragmatic software development approaches, which are inherently evolutionary in nature.

That it is indeed possible to come up with such an incremental approach, is illustrated by Egyed [25, 26], who proposes a lightweight incremental approach to model consistency checking that scales up to large industrial models. In a similar vein, De Fombelle [27] provides a formal treatment of incremental consistency checking of UML models. We therefore propose to build further on this work in order to come to a truly incremental, yet scalable support for model evolution tools.

3 Summary

Clearly, most of the challenges identified above interact and overlap. Therefore, they cannot be addressed in isolation. For example, formal solutions that allow us to specify and perform model refactorings, will have to be directly linked to solutions addressing

model quality and model consistency, since the main goal of applying a model refactoring is to improve model quality in a well-defined way, while preserving the overall model consistency. Hence, whatever technique and formalism that we come up with needs to guarantee that this is actually the case. Therefore, the research community on model evolution needs to combine its efforts to tackle these issues in an integrated way.

Also, let us repeat that the challenges put forward in this paper show only one part of the problem, as they focus on the horizontal dimension only. For a detailed list of other challenges related to model-driven software evolution, we refer to [6].

References

1. Stahl, T., Völter, M.: Model Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
2. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering. 20th anniversary edn. Addison-Wesley (1995)
3. Mens, T., Gorp, P.V.: A taxonomy of model transformation. In: Proc. International Workshop on Graph and Model Transformation (GraMoT 2005). Volume 152., Elsevier (2006)
4. Kleppe, A., Warmer, J., Bast, W. Addison-Wesley (2003)
5. Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004)
6. van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: A research agenda. In: Proc. CSMR Workshop on Model-Driven Software Evolution. (2007)
7. Zhang, J., Lin, Y., Gray, J.: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In: Model-driven Software Development - Research and Practice in Software Engineering. Springer (2005)
8. Favre, J.M.: Languages evolve too! changing the software time scale. In: Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), Los Alamitos, CA, USA, IEEE Computer Society (2005) 33–44
9. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Proc. Int'l Workshop on Principles of Software Evolution (IWPSE). (2005)
10. Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. 2nd edn. International Thomson Computer Press, London, UK (1997)
11. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). Volume 35 of SIGPLAN Notices., ACM Press (2000) 166–177
12. Tahvildari, L., Kontogiannis, K.: A metric-based approach to enhance design quality through meta-pattern transformations. In: Proc. European Conf. Software Maintenance and Reengineering (CSMR), IEEE Computer Society Press (2003) 183–192
13. Du Bois, B.: A Study of Quality Improvements by refactoring. PhD thesis, University of Antwerp (2006)
14. Mens, T.: On the use of graph transformations for model refactoring. In Ralf Lämmel, João Saraiva, J.V., ed.: Generative and transformational techniques in software engineering. Volume 4143 of Lecture Notes in Computer Science., Springer (2006) 219–257
15. Mens, T., Taentzer, G., Runge, O.: Analyzing refactoring dependencies using graph transformation. Software and Systems Modeling (2007)
16. Van Der Straeten, R., Mens, T., Jonckers, V.: A formal approach to model refactoring and model refinement. Software and Systems Modeling (2007) 139–162

17. Mens, T., Van Der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems. Volume 4199 of Lecture Notes in Computer Science., Springer-Verlag (2006) 200–214
18. Mens, T., Van Der Straeten, R.: Incremental resolution of model inconsistencies. In Fiadeiro, J.L., Schobbens, P.Y., eds.: Algebraic Description Techniques. Volume 4409 of Lecture Notes in Computer Science., Springer-Verlag (2007) 111–127
19. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 326–340
20. Van Der Straeten, R., Jonckers, V., Mens, T.: Supporting model refactorings through behaviour inheritance consistencies. In Thomas Baar, Alfred Strohmeier, A.M., ed.: UML 2004 - The Unified Modeling Language. Volume 3273 of Lecture Notes in Computer Science., Springer-Verlag (2004) 305–319
21. Guerra, E., Lara, J.D.: Model view management with triple graph transformation systems. In: Proc. Int'l Conf. Graph Transformation. Volume 4178 of Lecture Notes in Computer Science., Springer (2006) 351–366
22. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting collaborative development in an open mda environment. In: Proc. Int'l Conf. Software Maintenance. (2006) 244–253
23. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus : Towards the interoperability of modelling tool. In: Proc. European workshop on Model Driven Architecture: Foundations and Applications (MDAFA). Volume 3599 of Lecture Notes in Computer Science., Springer (2005)
24. Baader, F., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
25. Egyed, A.: Instant consistency checking for the UML. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM (2006)
26. Egyed, A.: Fixing inconsistencies in uml models. In: Proc. Int'l Conf. Software Engineering,, ACM Press (2007)
27. De Fombelle, G.: Gestion incrémentale des propriétés de cohérence structurelle dans l'ingénierie dirigée par les modèles. PhD thesis, Université Pierre et Marie Curie, Paris (2007)