Challenges in Model Refactoring

Tom Mens, University of Mons-Hainaut, Belgium
tom.mens@umh.ac.be
Gabriele Taentzer, Dirk Müller, Philipps-Universität Marburg, Germany
{taentzer,dmueller}@mathematik.uni-marburg.de

1. Introduction

Refactoring is a well-known technique to improve the quality of software. Fowler (1999) defines it as "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour". At the level of models, research on refactoring is still in its infancy. Many open questions remain that are worthy of further investigation. From a practical point of view, very few tools provide integrated support for model refactoring. Also, the types of models for which refactoring is supported is very limited. Therefore, the goal of this position paper is to discuss some of the most important challenges in model refactoring.

As the scenario depicted in Figure 1 shows, it is far from trivial to deal with model refactorings. Suppose that we have a model built up from many different views, using a variety of notations (e.g., class diagrams, state diagrams, interaction diagrams, and many more). Suppose that part of the corresponding source code is generated automatically, and another part is hand-written. When we apply a model refactoring to a single view in the model (step 1), we will need to synchronise all related views, in order to avoid them becoming inconsistent (step 2). Next, since the model has been changed, part of the code needs to be regenerated (step 3). Finally, the hand-written code needs to be adapted as well (step 4).

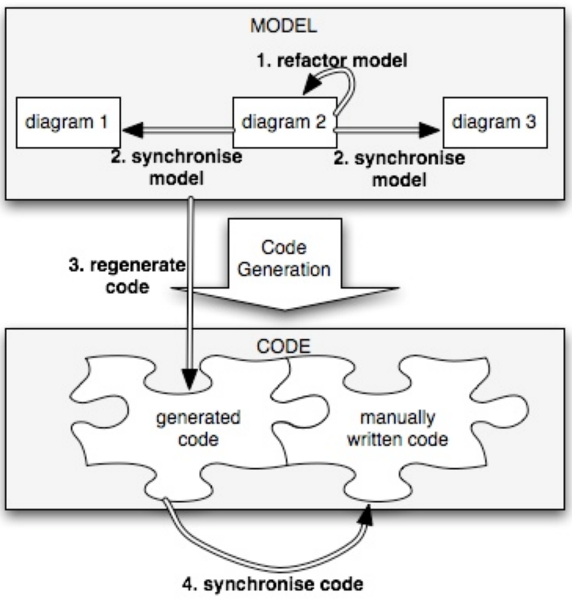

Figure 1: A scenario for model-driven software refactoring.

2. State-of-the-art in model refactoring

In research literature, mainly UML models are considered as suitable candidates for model refactoring (Sunyé *et al.*, 2001) (Astels, 2002) (Boger *et al.*, 2002). In particular, refactoring of class diagrams has been investigated by various researchers. Many of the refactorings known from object-oriented programming (Fowler, 1999) can be ported to UML class diagrams as well. Of course, additional techniques are needed in order to ensure traceability and consistency between class diagrams and their corresponding source code when applying class diagram refactorings (Bouden, 2006).

When it comes to reasoning about the behaviour preservation properties of models, we need to rely on OCL constraints, behavioural models (e.g., state diagrams, interaction diagrams or activity diagrams), or program code. With respect to refactoring of behavioural models, not much work is available. We are aware of a few approaches that address the problem of refactoring state diagrams, and try to prove their behaviour preservation properties in a formal way. Van Kempen *et al.* (2005) use a formalism based on CSP to describe statechart refactorings, and show how this formalism can be used to verify that a refactoring effectively preserves behaviour. Pretschner and Prenninger (2006) provide a formal approach for refactoring state machines based on logical predicates and tables. Gheyi *et al.* (2005) suggest to specify model refactorings in *Alloy*, an object-oriented modelling language used for formal specification. It can be used to prove semantics-preserving properties of model refactorings.

Various formalisms have been proposed to understand and explore model refactoring. Most of these approaches suggest expressing model refactoring in a declarative way. Van Der Straeten *et al.* (2004) use description logics; Van Der Straeten & D'Hondt (2006) use a forward-chaining logic reasoning engine to support composite model refactorings. Gheyi *et al.* (2005) specify model refactorings using Alloy, a formal object-oriented modelling language. They use its formal verification system to specify and prove the soundness of the transformations. Biermann *et al.* (2006) and Mens *et al.* (2007) use graph transformation theory as an underlying foundation for specifying model refactoring, and rely on the formal properties to reason about and analyse these refactorings.

3. Challenges in model refactoring

Model quality.

A first challenge is to provide a precise definition of model quality. A model can have many different non-functional properties or quality characteristics that may be of interest. Usability, readability, performance, adaptability, security are some examples of desirable qualities that may be important to different types of stakeholders. How these qualities can be specified formally, and how they relate to one another, remains an open challenge.

Since the main goal of refactoring is to improve certain aspects of the software quality, we need means to assess this quality at the model level in an objective way. On the one hand, this will allow software modelers to identify which parts of the model contain symptoms of poor quality, and are hence potential candidates for model refactoring. On the other hand, quality assessment techniques can be used to verify to which extent model refactorings actually improve the model quality.

One of the ways to assess 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 in (Fowler, 1999) to refer to structures in the code that suggest opportunities for refactoring. 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. What is also needed is a good 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.

Another way to assess and control model quality is by resorting to *model metrics*. In analogy with software metrics (Fenton & Pfleeger, 1997) they are used to measure and quantify desirable aspects of models. How to define model metrics in such a way that they correlate well with external model quality characteristics remains an open question. Another important issue is to explore the relation between model metrics and model refactoring, and particular to assess to which extent model refactorings affect metric values. These issues have been addressed by (Demeyer *et al.*, 2000) (Du Bois, 2006) (Tahvildari & Kontogiannis, 2004) though mainly at code level.

A final way to improve model quality is by introducing design patterns, which are proven solutions to recurring patterns (Gamma *et al.*, 1994). At code level, Kerievsky (2004) explored the relation between refactorings and design patterns. It remains to be seen how similar results may be achieved at the level of models.

Model synchronisation.

One of the key questions is how model refactoring actually differs from program refactoring. Can the same ideas, techniques and even tools used for program refactoring be ported to the level of models? If not, what is it precisely that makes them different?

One answer to this question is that models are typically built up from different views, using different types of diagrams, that all need to be kept consistent. This in contrast to programs, that are often (though not always) expressed within a single programming language.[1]

Perhaps a more important difference is that models are used as a basis to generate a large portion of the source code that would otherwise need to be written manually. Since 100% full code generation is unfeasible in practice, the additional challenge therefore consists in the need to synchronise and maintain consistency between models and their corresponding program code, especially when part of this program code has been specified or modified manually. In the context of model transformation, this implies that automated model refactorings may need to be supplemented with code-level transformations in order to ensure overall consistency. Vice versa, program refactorings may need to be supplemented with model-level transformations to ensure their consistency. Though no general solutions exist yet, the problem of synchronising models and code is well known in literature. For example, Van Gorp *et al.* (2003) discuss the problem of keeping the UML models consistent with its corresponding program code. Bottoni *et al.* (2005) use distributed graph transformation concepts to specify coherent refactorings of several software artefacts, especially UML models and Java programs. The triple graph grammar approach by Königs & Schürr (2006) can also be used to synchronise models and programs.

Behaviour preservation.

Another important challenge of model refactoring has to do with *behaviour preservation*. By definition, a model refactoring is supposed to preserve the observable behaviour of the model it is transforming. In order to achieve this, we need a precise definition of "behaviour" in general, and for models in particular. One of the main problems is the lack of a generally accepted formal semantics for UML. Many different interpretations exist, and this has an important effect on how the behaviour is formally defined. Even assuming that such a semantics would exist, we still need formalisms that allow us to specify and reason about behavioural invariants, i.e., properties that need to be preserved by the model refactoring.
Note that, in practice, we may not always desire full behaviour preservation. In many cases we may be more flexible, as long as we are able to determine precisely how a given model transformation modifies the behaviour.

Testing model refactoring.

Many researchers have looked at how to combine the ideas of testing with model-driven engineering. Test-driven development is suggested as good practice for writing high-quality software. In combination with refactoring, it implies that before and after each refactoring step, tests are executed to ensure that the behaviour remains unaltered. Clearly, the code has to satisfy the same test cases before and after refactoring it. Considering refactoring within model-driven development, the same testing procedure should be possible. Future investigations should clarify the impact of this kind of refactoring on test suites (Van Deursen *et al.*, 2002).

[1] Of course, programs also need to be synchronised with related software artefacts (such as databases, user interfaces, test suites) that may have been expressed using a different language.

Domain-specific modelling.

Another challenge is the need to define model refactorings in domain-specific modelling languages. These refactorings should be expressible in a generic yet customisable way. Indeed, given the large number of very diverse domain-specific languages, it is not feasible, nor desirable, to develop dedicated tools for all of them from scratch. Zhang *et al.* (2004) proposed such a generic model transformation engine that can be used to specify refactorings for domain-specific models.

Analysing model refactorings.

More advanced support for model refactorings can be envisaged if we have a precise means to analyse and understand the relationships between refactorings. This will enable us, among others, to build up complex refactorings from simpler ones; to detect whether refactorings are mutually exclusive, in the sense that they are not jointly applicable; to analyse causal dependencies between refactorings. These techniques have been explored in detail by Mens *et al.* (2005-2007), where graph transformation theory is proposed to offer more guidance to the developer on what is the most appropriate refactoring to apply in which context. Compared to other approaches it has a number of advantages: it allows one to specify refactorings for various languages in a uniform and generic way, by representing the software artefact under consideration as a graph, and by specifying the refactorings as graph transformation rules. In addition, one can benefit from the formal properties of graph transformation theory to reason properties such as termination, composition, parallel and sequential dependencies.

Tool support.

Tool support for model refactoring should preferably be integrated into open source software development environments with a large user base. Since the Eclipse Modelling Framework (EMF) has become a key reference for model specification in the world of model-driven development, we suggest to implement model refactoring as EMF model transformations (Biermann *et al.* 2006). This does not preclude the ability to formally reason about such refactorings, since EMF transformations can be translated to graph transformations under certain circumstances.

Another important aspect of tool support has to do with performance and scalability. Is it possible to come up with model refactoring tools that scale up to industrial software? Egyed (2006) provided initial evidence that this is actually the case, by providing an instant model synchronisation approach that scales up to large industrial software models.

Distributed software development.

When developing large software systems in a model-driven manner, several development teams might be involved. In this case, it would be advantageous if the model could be subdivided into several parts that could be developed in a distributed way. Considering refactoring in this setting, model elements from different sub models might be involved. Thus, several distributed refactoring steps have to be performed and potentially synchronized if they involve common model parts. Distributed refactoring steps could be considered as distributed model transformations (Goedicke *et al.*, 1999) (Bottoni *et al.*, 2005).

4. References

Astels, D. (2002). Refactoring with UML, In: *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering* (pp. 67-70)

Biermann, E., Ehrig, K., Köhler, C., Taentzer, G., & Weiss, E. (2006). Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: *Proc. Int'l Conf. Model Driven Engineering Languages and Systems* (pp. 425-439), LNCS 4199, Springer

Boger, M., Sturm, T., & Fragemann, P. (2002). Refactoring Browser for UML, In: *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering* (pp. 77-81)

Bottoni, P., Parisi-Presicce, F., Mason, G., & Taentzer, G. (2005). Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations. In: *Handbook on Transformation of Knowledge, Information, and Data: Theory and Applications* (pp. 95-125), Idea Group Publishing

Bouden, S. (2006). *Étude de la traçabilité entre refactorisations du modèle de classes et refactorisations du code*. PhD Thesis, Université de Montréal, Canada.

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2000). Finding Refactorings Via Change Metrics, In: *Proc. Int'l Conf. OOPSLA 2000* (pp. 166-177), ACM SIGPLAN Notices 35(10), ACM Press.

Du Bois, B. (2006). *Quality-Oriented Refactoring*. PhD Thesis, Universiteit Antwepen, Belgium

Egyed, A. (2006). Instant consistency checking for the UML. In: *Proc. Int'l Conf. Software Engineering* (pp. 31-390), ACM

Fenton, N., & Pfleeger, S. L. (1997). *Software Metrics: A Rigorous and Practical Approach* (2nd edition), International Thomson Computer Press

Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems.* Addison-Wesley

Gheyi, R., Massoni, T., & Borba, P. (2005). Type-safe Refactorings for Alloy. In: *Proc. Brazilian Symposium on Formal Methods* (pp. 174-190). Porto Alegre, Brazil.

Goedicke, M., Meyer, T., & Taentzer, G. (1999). Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In: *Proc. Int'l Conf. Requirements Engineering* (pp. 92-99), IEEE Computer Society.

Kerievsky, J. (2004). *Refactoring to Patterns*, Addison-Wesley. ISBN 978-032-121-335-8

Königs, A. & Schürr, A. (2006) Tool Integration with Triple Graph Grammars - A Survey . In: R. Heckel (Ed.), Proc. SegraVis School on Foundations of Visual Modelling Techniques (pp. 113-150), ENTCS 148, Elsevier.

Mens, T. (2006). On the use of graph transformations for model refactoring. In: *Generative and Transformational Techniques in Software Engineering* (pp. 219-257), LNCS 4143, Springer.

Mens, T., Van Eetvelde, N., Demeyer, S., & Janssens, D. (2005). Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution* 17(4), 247-276, Wiley.

Mens, T., Taentzer, G., & Runge, O. (2007). Analyzing Refactoring Dependencies Using Graph Transformation. *Journal on Software and Systems Modeling*. Springer, Heidelberg. To appear.

Pretschner, A., & Prenninger, A. (2006). Computing Refactorings of State Machines, *Journal on Software and Systems Modeling*. To appear.

Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.-M. (2001). Refactoring UML models, In: *Proc. Int'l Conf. Unified Modeling Language* (pp. 134-138), LNCS 2185, Springer.

Tahvildari, L., & Kontogiannis, K. (2004). Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach, *Journal of Software Maintenance and Evolution* 16(4-5), 331-361, Wiley.

Van Deursen, A., & Moonen, L. (2002). The Video Store Revisited: Thoughts on Refactoring and Testing, In: *Proc. Int'l Conf. Extreme Programming and Flexible Processes in Software Engineering* (pp. 71-76)

Van Deursen, A., Moonen, L., van den Bergh, A., & Kok, G. (2002). Refactoring Test Code, In: *Extreme Programming Perspectives* (pp. 141-152), Addison-Wesley.

Van Der Straeten, R., & D'Hondt, M. (2006). Model refactorings through rule-based inconsistency resolution, In: *Proc. Symposium on Applied computing* (pp. 1210-1217), ACM Press.

Van Der Straeten, R., Jonckers, V., & Mens, T. (2004). Supporting Model Refactorings through Behaviour Inheritance Consistencies, In: *Proc. Int'l Conf. Unified Modeling Language* (pp. 305-319), LNCS 3273, Springer.

Van Gorp, P., Stenten, H., Mens, T., & Demeyer, S. (2003). Towards automating source-consistent UML refactorings, In: *Proc. Int'l Conf. Unified Modeling Language* (pp. 144-158). LNCS 2863, Springer, Heidelberg

Van Kempen, M., Chaudron, M., Koudrie, D., & Boake, A. (2005). Towards Proving Preservation of Behaviour of Refactoring of UML Models. In: *Proc. SAICSIT 2005* (pp. 111-118).

Zhang, J., Lin, Y., & Gray, J. (2005). Generic and Domain-Specific Model Refactoring using a Model Transformation Engine, In *Model-driven Software Development - Research and Practice in Software Engineering*, Springer.