Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support *

Thorsten Arendt, Sieglinde Kranz, Florian Mantz, Nikolaus Regnat, Gabriele Taentzer

Philipps-Universität Marburg, Germany
Siemens Corporate Technology, Germany
Høgskolen i Bergen, Norway

{arendt,taentzer}@mathematik.uni-marburg.de
{sieglinde.kranz,nikolaus.regnat}@siemens.com
fma@hib.no

Abstract: The paradigm of model-based software development has become more and more popular, since it promises an increase in the efficiency and quality of software development. Following this paradigm, models become primary artifacts in the software development process where quality assurance of the overall software product considerably relies on the quality assurance of involved software models. In this paper, we concentrate on the syntactical dimension of model quality which is analyzed and improved by model metrics, model smells, and model refactorings. We propose an integration of these model quality assurance techniques in a predefined quality assurance process being motivated by specific industrial needs. By means of a small case study, we illustrate the model quality assurance techniques and discuss Eclipse-based tools which support the main tasks of the proposed model quality assurance process.

1 Introduction

In modern software development, models play an increasingly important role, since they promise more efficient software development of higher quality. It is sensible to address quality issues of artifacts in early software development phases already, for example the quality of the involved software models. Especially in model-driven software development where models are used directly for code generation, high code quality can be reached only if the quality of input models is already high.

In this paper, we consider a model quality assurance process which concentrates on the syntactical dimension of model quality. Syntactical quality aspects are all those which can be checked on the model syntax only. They include of course consistency with the language syntax definition, but also other aspects such as conceptual integrity using the same

*This work has been partially funded by Siemens Corporate Technology, Germany.

patterns and principles in similar situations and conformity with modeling conventions often specifically defined for software projects. In [FHR08], the authors present a taxonomy for software model quality distinguishing between inner and outer quality aspects of models. Inner quality aspects are concerned with a single model only. The consistency of a model, its conceptual integrity, and its conformity to standards are typical examples for inner quality aspects of models. Outer quality aspects are concerned with relations of a model to other artifacts of a software engineering process. The completeness of a design model wrt. to its analysis model on the one hand and to code on the other hand falls in this category of quality aspects. These and further quality aspects are discussed in [FHR08].

In the literature, typical syntactical quality assurance techniques for models are model metrics and refactorings, see e.g. [GPC05, SPLTJ01, MB08, Por03]. They origin from corresponding techniques for software code by lifting them to models. Especially class models are closely related to programmed class structures in object-oriented programming languages such as C++ and Java. For behavior models, the relation between models and code is less obvious. Furthermore, the concept of code smells can be lifted to models leading to model smells. Again code smells for class structures can be easily adapted to models, but smells of behavior models cannot directly deduced from code smells.

In this paper, we present a **quality assurance process for syntactical model quality** which can be adapted to specific project needs. It consists of two phases: First, project- and domain-specific quality checks and refactorings have to be specified which should be done before a software project starts. Model quality checks are formulated by model smells which can be specified by model metrics or anti-patterns. Thereafter, the specified quality assurance process can be applied to concrete software models by reporting all their model smells and applying model refactorings to erase at least some of the model smells found. However, we have to take into account that also new model smells can come in by refactorings. This check-improve cycle should be performed as long as needed to get a reasonable model quality. The process is supported by tools, i.e. model metrics reports, smell detection and the application of refactorings are supported by Eclipse plug-ins being based on the Eclipse Modeling Framework.

This paper is organized as follows: In the next section, the need for syntactical model quality assurance in industrial software development is motivated. In Section 3, we present a two-phase quality assurance process for syntactical model quality which can be adapted to specific project needs. In Section 4, we present Eclipse plug-ins EMF Metrics, EMF Smell, and EMF Refactor at a small example. Finally, related work is discussed and a conclusion is given.

2 The need for syntactic model quality assurance in industrial software development

A typical model-based software development project at Siemens covers between 10 - 100 developers. These developers use models in different ways, e.g. specifying the software architecture and design, using input for code implementation or getting information for

tests and test case generation. Often these developers are not on one site, e.g. architects are located in Germany and the implementers are located at a site in east Europe. In this case, models are an essential part of development communication and their quality influences the quality of the final product to a great extent. In addition, model-based software projects are often part of mechatronic systems with safety relevant parts. In these cases safety aspects must also be observed. Standards like the IEC 61508 requires that for a mechatronic system all intermediate results during the development process including software models must be of an appropriate quality.

At the moment the most commonly used modeling language is the UML. It is a very comprehensive and powerful language, but does not cover any particular method and comes without built-in semantics. On the one hand, this allows a flexible use but includes a high risk of misuse on the other hand. Without tailoring a project specific usage of UML before starting development, the practical experience showed that the created models can be difficult to understand or even misinterpreted. Detailed project-specific guidelines are very important therefore and their compliance must be enforced and controlled.

An essential aspect of modeling is the capability to consider a problem and its solution from different perspectives. The existence of different perspectives increases, however, the risk of inconsistencies within a model. To avoid large models and to deal with different organizational responsibilities, the model information is frequently split into a set of smaller models with relationships in between. Information contained in one model is reused (and probably more detailed) in other models. Consistency is therefore not only needed within a model, but also between a project-specific set of models. Unfortunately, the support of available UML modeling tools to prevent the user from modeling inconsistencies and to easily find contradictions is very limited. Often, parallel changes in one model cannot be avoided in development projects. A frequent reason for that is an established feature oriented software development process. Required subsequent merges can easily result in inconsistencies, especially since the merge functionality of most UML modeling tools is not satisfactory.

Since model-driven software development is not yet a well established method in industry, software development is often faced with the problem that at least a part of its project members have no or limited experience with modeling in general as well as with the modeling language used and its tools. This often leads to misuse such as modeling of unnecessary details on a higher abstraction level and removing of no longer needed model elements only from diagrams instead from the model. Especially at the beginning of a model-based project such problems should be identified as soon as possible to avoid that the misuse is copied by other modelers and to ensure that the effort to correct these problems is still low.

Unfortunately, the existing and established quality assurance methods *document review* and *code inspection* cannot be used one-to-one within model-based development. The manual review of models is very time consuming and error prone. Models cannot be reviewed in a sequential way because of the existence of links between its elements and to other models. To reduce the quality risks of models mentioned above and unburden the review effort, it is essential that each project defines a specific list of guidelines at the beginning of its model-based software development, derives syntactic checks from these guidelines and automates these checks by a tool.

3 The syntactical model quality assurance process

In this section we propose the definition and application of a structured model quality assurance process that can be used to address project-specific needs as described in the previous section. The approach uses already known model quality assurance techniques like model metrics, model smells, and model refactorings which are combined in an overall process for structured model quality assurance concerning syntactical model issues.

3.1 A two-phase model quality assurance process

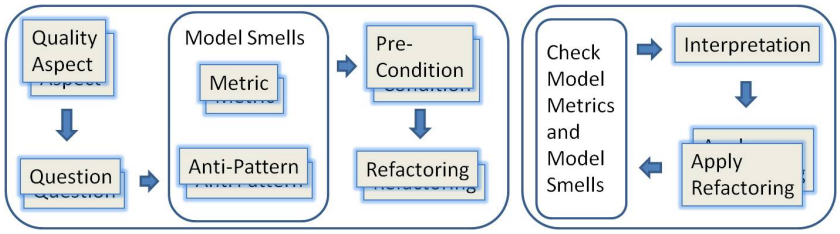

Figure 1: Project-Specific Quality Assurance Process - Specification and Application

As already mentioned, it is essential to define quality-related issues at the beginning of model-based software development. On the left side of Figure 1, we show how to define a model quality assurance process for a specific project. Firstly, it is required to determine those quality aspects which are important for project-specific software models. In the next step, static syntax checks for these quality aspects are defined. This is done by formulating questions that should lead to so-called model smells which hint to model parts that might violate a specific model quality aspect. Here, we adopt the Goal-Question-Metrics approach (GQM) that is widely used for defining measurable goals for quality and has been well established in practice [BCR94]. The formulated questions need answers which can be given by considering the model syntax only. Some of these answers can be based on metrics. Other questions may be better answered by considering patterns. Here, the project-specific process can reuse general metrics and smells as well as special metrics and smells specific for the intended modeling purpose.

Finally, a specified model smell serves as pre-condition of at least one model refactoring that can be used to restructure models in order to improve model quality aspects but appreciably not influence the semantics of the model. Since every model refactoring comes along with initial and final pre-conditions which have to be checked before respectively after user input, a mapping of a certain model smell to some initial pre-conditions might be a hint for using the corresponding refactoring in order to eliminate the smell.

During model-based software development, the defined quality assurance process can be applied as shown on the right side of Figure 1. For a first overview, a report on model

metrics might be helpful. Furthermore, a model has to be checked against the existence (respectively absence) of specified model smells. Each model smell found has to be interpreted in order to evaluate whether it should be eliminated by a suitable model refactoring or not. It is recommended that the process is supported by appropriate tools as presented in the next section.

3.2 Example case

Here, we describe a small example case for the proposed project-specific model quality assurance process. Please note that due to space limitations we concentrate on one quality aspect only. We also do not specify each process implementation issue in detail.

The quality aspect we consider in our example is *consistency*. This quality aspect has several facets. We concentrate on inner consistency wrt. the modeling language used and wrt. modeling guidelines to be used. The modeling language comprises at least class models and state machines. Example questions can be:

- Are there any elements not shown in any diagram of the model?
- Are there any cycles in the element dependency graph?
- Are there any equally named classes in different packages?
- Are there any abstract classes that are not specialized by at least one concrete class?
- Are there any attributes redefining other ones within the same inheritance hierarchy?
- Are there any state diagrams without initial or final state?

These questions can lead to the definition of the following model smells, partially known from literature: (1) *element not shown in diagram*, (2) *dependency cycle* [Mar02], (3) *multiple definitions of classes with equal names* [Lan07], (4) *no specification* [Rie96], (5) *attribute name overridden*, and (6) *missing initial/final state* [RQZ07].

As already mentioned, there are at least two different ways to check a quality aspect by model smells. One alternative is to define a metric-based model smell which can be evaluated on the model. In our case study, we can use metrics *NOCS* (number of concrete subclasses of a given class), *NOIS* (number of initial states of a given state diagram region), and *NOFS* (number of final states of a given state diagram region) to address model smells (4) and (6), for example. If any of these metrics is evaluated to zero in given model contexts, the corresponding smell is identified. The second alternative to define model smells is to specify an anti-pattern representing a pattern which should not occur. It is defined based on the abstract syntax of the modeling language. In our case study we use anti-patterns *Equally named classes*, *No concrete subclass*, and *Redefined attribute* to address model smells (3) - (5). Note that model smell (4) can be specified in both ways: using model metric *NOCS* or anti-pattern *No concrete subclass*, respectively. Figure 4 shows anti-pattern *No specification*.

In order to eliminate specific model smells, corresponding model refactorings have to be specified. To address smells *Multiple definition of classes with equal names* and *No specification* of our example, we can use the well known model refactorings *Rename Class* and *Insert Subclass*, respectively. Of course, it is very important to consider all possible effects of a specific model refactoring ahead of its application.

4 Tool support

Since a manual model review is very time consuming and error prone, it is essential to automate the tasks of the proposed model quality assurance process as effectively as possible. Therefore, we implemented three tools supporting the included techniques metrics, smells, and refactorings (MSR) for models based on the Eclipse Modeling Framework (EMF) [EMF], a common open source technology in model-based software development.

Each tool consists of two independent components. The *generation module* addresses project managers respectively project staff who are responsible for the definition of the project-specific model quality assurance process. Project-specific metrics, smells, and refactorings are defined wrt. a specific modeling language specified by an EMF model. The second component of each tool supports the *application* of the specified MSR tools using the Java code produced by the generation module.

MSR tools can be specified by implementing metrics, smells and refactorings directly in Java or by using a model transformation tool such as Henshin [Hen], a new approach for in-place transformations of EMF models [ABJ+10]. Henshin uses pattern-based rules which can be structured into nested transformation units with well-defined operational semantics. We implemented a number of metrics, smells, and refactorings for EMF-UML models. In the following we shortly present the Eclipse plug-ins *EMF Metrics*, *EMF Smell*, and *EMF Refactor*.

4.1 EMF Metrics

Using the *EMF Metrics* prototype, metrics can be defined and calculated wrt. specific EMF-based models. When defining a new metric, the context of the metric has to be specified, i.e. the meta model given by its *NamespaceUri* as well as the model element type (e.g. *UML::Class*) to which the metric shall be applied. The prototype supports two distinct methods for defining new metrics. They are either defined directly by model transformations or two existing metrics are combined using an arithmetic operation.

Figure 2 shows the Henshin rule defining UML model metric *NOCS* (see previous section) specified on the abstract syntax of UML. *EMF Metrics* uses this rule to find matches in a concrete UML model. Starting from node *selectedEObject* of type *Class* being the context element, the Henshin interpreter returns the total number of matches that can be found in the model. This number represents the value of the corresponding model metric.

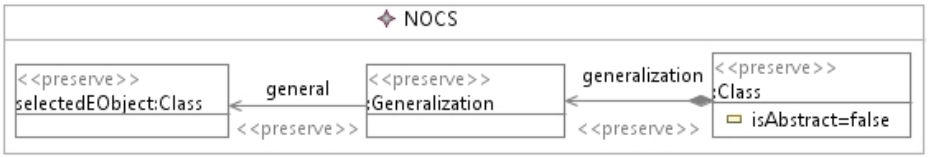

Figure 2: Henshin rule defining UML model metric *NOCS*

Figure 3 shows an example model and the corresponding result view after calculating five different metrics on abstract class *Vehicle*. This class owns four attributes *horsepower*, *seats*, *regNo*, and *owner*. Only the latter attribute has public visibility, so the AHF value of class *Vehicle* is evaluated to 0.75 . Each result contains a time stamp to trace metric values over time. For reporting purposes, *EMF Metrics* provides an XML export of its results.

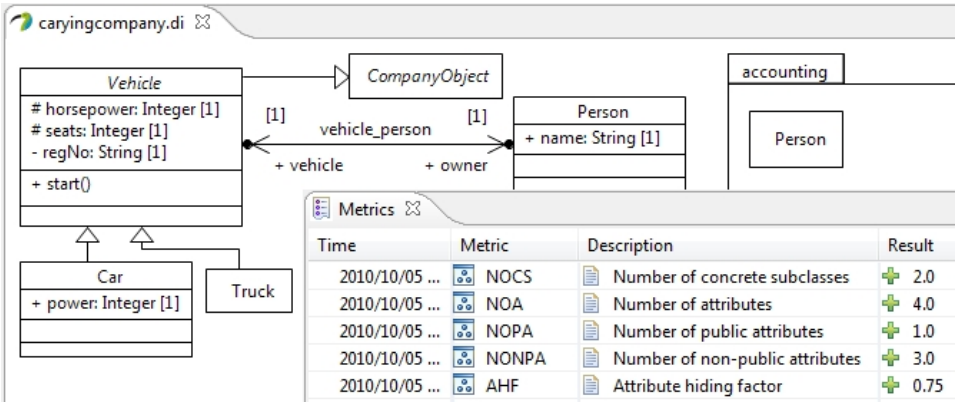

Figure 3: Model metrics result view after calculating five different metrics on abstract class *Vehicle*

4.2 EMF Smell

Similarly to *EMF Metrics*, *EMF Smell* consists of a generation and an application module. In the generation module, model smells can be specified using Henshin rules in order to define anti-patterns. There is no context that has to be specified because the anti-pattern has to be identified along the entire model. Of course, the modeling language has to be referred to. The definition of smells based on metrics is up to future work.

Figure 4 shows the Henshin rule for checking UML model smell *No Specification*. The pattern to be found specifies an abstract UML class (on the left) that is not specialized by a non-abstract UML class (on the right). Please note that parts of the pattern that are not allowed to be found are tagged with *forbid*. Additionally, parameter *modelElementName* is

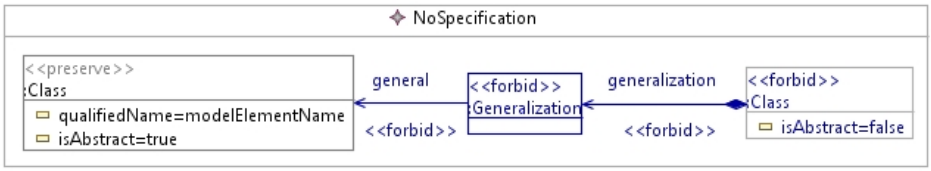

Figure 4: Henshin rule for checking UML model smell *No specification*

set by each pattern match to return model element instances participating in the identified model smell.

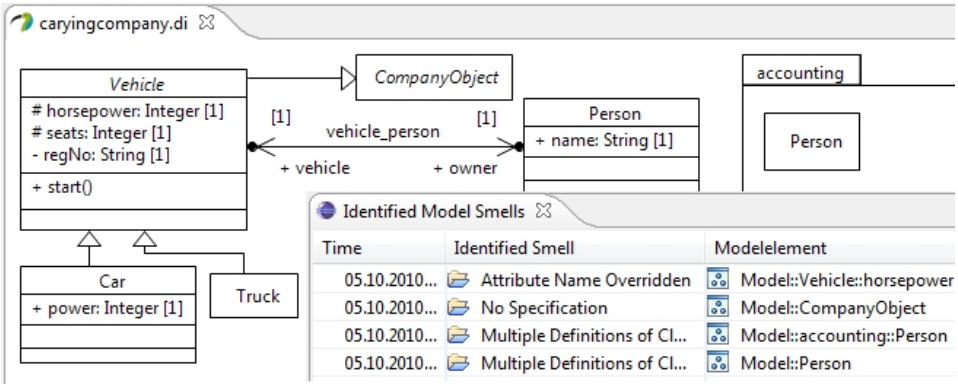

Figure 5: Model smell result view after checking the sample UML model

Figure 5 shows the application of the *EMF Smell* checking component on our example UML class model. The smell checking process can be triggered from within the context menu of the corresponding model file. Four smells have been found in the example model: There are two equally named classes *Person* while abstract class *CompanyObject* has no (direct) concrete subclass. Furthermore, attribute *horsepower* of class *Vehicle* is redefined by attribute *power* of class *Car*[1]. Like in *EMF Metrics*, model smells found are presented in a special view.

4.3 EMF Refactor

The third model quality assurance tool, *EMF Refactor* [Ref], is a new Eclipse incubation project in the Eclipse Modeling Project consisting of three main components. Besides a code generation module and a refactoring application module, it comes along with a suite

[1]This fact is not visible in the graphical view!

of predefined EMF model refactorings for UML and Ecore models. [2]

Since the application module uses the Eclipse Language Toolkit (LTK) technology [LTK], a refactoring specification requires up to three parts, implemented in Java and maybe generated from model transformation specifications, that have to be defined. They reflect a primary application check for a selected refactoring without input parameter, a second one with parameters and the proper refactoring execution.

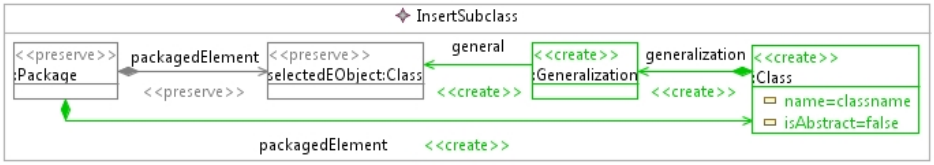

Figure 6: Henshin rule for executing UML model refactoring *Insert Subclass*

Figure 6 shows the Henshin rule for executing UML model refactoring *Insert Subclass*. The invocation context is given by node *selectedEObject* of type *Class*. The rule inserts a new non-abstract class named *classname* to the same package owning the selected class. Furthermore, the newly created class becomes a specialization of the selected class.

EMF Refactor provides two ways to configure the set of model refactorings. First, so-called refactoring groups are set up in order to arrange model refactorings for one modeling language. Second, these groups can be referenced by a specific Eclipse project to address project-specific needs.

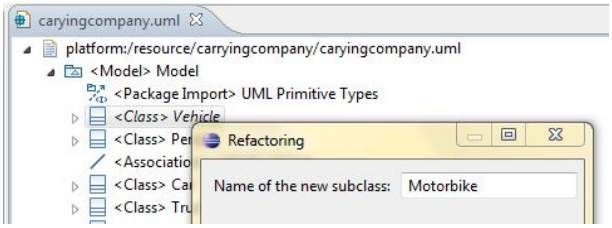

Figure 7: UML model refactoring *Insert Subclass* with parameter input

The application of a model refactoring mirrors the three-fold specification of refactorings based on LTK. After specifying a trigger model element such as class *Vehicle* in Figure 7, refactoring-specific initial conditions are checked. Then, the user has to set all parameters, for example the name of the new subclass in our *Insert Subclass* refactoring. *EMF Refactor* checks whether the user input does not violate further conditions. In case of erroneous parameters a detailed error message is shown. In our concrete example, it is checked whether the package owning the selected class already owns an element with the specified

[2] Of course, *EMF Metrics* and *EMF Smell* shall also come along with suites of predefined metrics and smells, respectively, in future.

name. If the final check has passed, *EMF Refactor* provides a preview of the changes that will be performed by the refactoring using EMF Compare as shown in Figure 8. Last but not least, these changes can be committed and the refactoring can take place.

Currently, *EMF Refactor* supports refactoring invocation from within three different kinds of editors: tree-based EMF instance editors (like in Figure 7), graphical model editors generated by Eclipse GMF (Graphical Modeling Framework), and textual model editors used by Xtext.

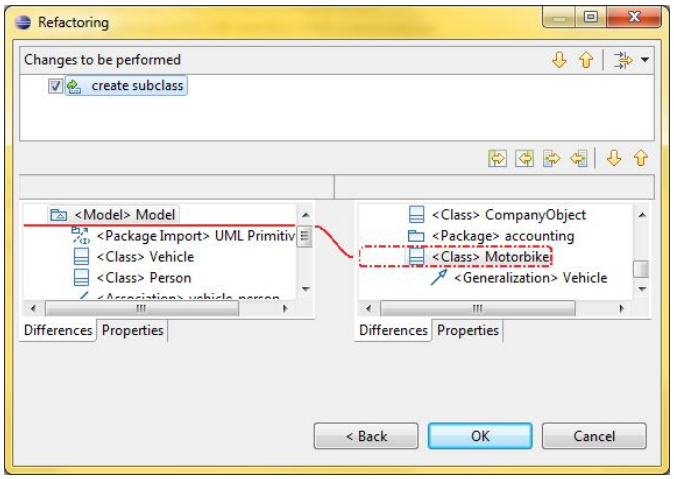

Figure 8: EMF Compare preview dialog of UML model refactoring *Insert Subclass*

5 Related work

Software quality assurance is concerned with software engineering processes and methods used to ensure quality. The quality of the development process itself can be certified by ISO 9000. CMMI [CMM] can help to improve existing processes. Considering the quality of a software product instead, the ISO/IEC 9126 standard lists a number of quality characteristics concerning functionality, reliability, usability, etc. Since models become primary artifacts in model-based software development, the quality of a software product directly leads back to the quality of corresponding software models.

There is already quite some literature on quality assurance techniques for software models available, often lifted from corresponding techniques for code. Most of the model quality assurance techniques have been developed for UML models. Classical techniques such as software metrics, code smells and code refactorings have been lifted to models, especially to class models. Model metrics have been developed in e.g. [GPC05] and model smells as well as model refactorings are presented in e.g. [SPLTJ01, MB08, Por03, MTR07, PP07]. In [Amb02], Ambler transferred the idea of programming guidelines to UML models.

Since EMF has evolved to a well-known and widely used modeling technology, it is worthwhile to provide model quality assurance tools for this technology. There are further tools for the specification of EMF model refactorings around as e.g. the Epsilon Wizard Language [KPPR07]. We compare our approach with other specification tools for EMF model refactorings in [AMST09]. To the best of our knowledge, related tools for metrics calculation and smells detection in EMF models are not yet available.

6 Conclusion

Since models become primary artifacts in model-based software development, model quality assurance is of increasing importance for the development of high quality software. In this paper, we present a quality assurance process for syntactical model quality being based on model metrics, model smells and model refactorings. The process can be adapted to specific project needs by first defining specific metrics, smells and refactorings and applying the tailored process to the actual models thereafter. Smell detection and model refactoring are iterated as long as a reasonable model quality has not reached.

As a next step, we plan to evaluate the proposed process as well as the presented tool-support in a bigger case study by integrating the tools in IBM's Rational Software Architect and applying the process on a large-scale model at Siemens.

There are model smells which are difficult to describe by metrics or patterns: For example, *shotgun surgery* is a code smell which occurs when an application-oriented change requires changes in many different classes. This smell can be formulated also for models, but it is difficult to detect it by analyzing models. It is up to future work to develop an adequate technique for this kind of model smells.

Furthermore, future work shall take complex refactorings into account which need to be performed in the right order depending on inter-dependencies of basic refactorings. Moreover, if several modelers refactor in parallel, it might happen that their refactorings are in conflict. In [MTR07], conflicts and dependencies of basic class model refactorings are considered and execution orders as well as conflict resolution are discussed.

As pointed out in Section 2, model consistency is not only a subject within one model but also between several models. Thus, refactorings have to be coordinated by several concurrent model transformations. A first approach to this kind of model transformations is presented in [JT09]. It is up to future work, to employ it for coordinated refactorings.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In *Model Driven Engineering Languages and Systems, 13th International Conference, MoDELS 2010. Proceedings*, LNCS, pages 121–135. Springer, 2010.

[Amb02] Scott W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2002.

[AMST09] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study. http://www.modse.fr/modsemccm09/doku.php?id=Proceedings, 2009.

[BCR94] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, 1994.

[CMM] Capability Maturity Model Integration (CMMI). http://www.sei.cmu.edu/cmmi/.

[EMF] EMF. Eclipse Modeling Framework. http://www.eclipse.org/emf.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik Spektrum*, 31(5):408–424, 2008.

[GPC05] M. Genero, M. Piattini, and C. Calero. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4(9):59 – 92, 2005.

[Hen] Henshin. http://www.eclipse.org/modeling/emft/henshin.

[JT09] Stefan Jurack and Gabriele Taentzer. Towards Composite Model Transformations using Distributed Graph Transformation Concepts. In Andy Schuerr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*. Springer, 2009.

[KPPR07] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[Lan07] Christian F.J. Lange. *Assessing and Improving the Quality of Modeling: A series of Empirical Studies about the UML*. PhD thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, The Netherlands, 2007.

[LTK] The Language Toolkit (LTK). http://www.eclipse.org/articles/Article-LTK.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, 2002.

[MB08] Slavisa Markovic and Thomas Baar. Refactoring OCL Annotated UML Class Diagrams. *Software and Systems Modeling*, 7:25–47, 2008.

[MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.

[Por03] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In G. Booch P. Stevens, J. Whittle, editor, *Proc. UML 2003: 6th Intern. Conference on the Unified Modeling Language*, LNCS, pages 159–174. Springer, 2003.

[PP07] Alexander Pretschner and Wolfgang Prenninger. Computing Refactorings of State Machines. *Software and Systems Modeling*, 6(4):381–399, December 2007.

[Ref] EMF Refactor. http://www.eclipse.org/modeling/emft/refactor/.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[RQZ07] C. Rupp, S. Queins, and B. Zengler. *UML 2 glasklar*. Hanser Fachbuchverlag, 2007.

[SPLTJ01] G. Sunye, D. Pollet, Y. Le Traon, and J. Jezequel. Refactoring UML models. In *Proc. UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer-Verlag, 2001.