Model/Analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML

Alexander Reder
Institute for Systems Engineering and
Automation
Johannes Kepler University Linz, Austria
alexander.reder@jku.at

Alexander Egyed
Institute for Systems Engineering and
Automation
Johannes Kepler University Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Integrated development environments are widely used in industry and support software engineers with instant error feedback about their work. Modeling tools often react to changes at a coarse level of granularity that make reasoning about errors inefficient and late. Furthermore, there is often a lack of appropriate visualizations of model errors and information on how to fix them. This paper presents the *Model/Analyzer* tool, an eclipse-based plug-in for the IBM Rational Software Modeler (RSM). The tool lets software engineers define arbitrary design rules and provides instant feedback on their validity in context of a model. Design errors are then visualized together with the information on what parts of the model contributed to them and how to fix them. The tool is fully automated and currently supports OCL and Java as languages for defining the design rules; and UML as the modeling language. The main benefit for the software engineer is the tool's incremental nature – providing instant feedback for many kinds of design errors even for large models.

Categories and Subject Descriptors: I.6.4 Simulation and Modeling: Model Validation and Analysis

General Terms: Design, Performance

Keywords: OCL, UML, Consistency Checking

1. INTRODUCTION

To reduce the cost of fixing design errors in software models, it is important to detect them early in the development process. Today's programming environments are efficient in detecting many/most syntax and semantic errors instantly during the writing of source code. In model-based development such support is rare. Most approaches for detecting design errors in software models are batch based and require all design rules to evaluate at once – a computing intensive problem that is done after long intervals only and typically generates vasts amount of errors that are hard to associate to the model changes that caused them. The few approaches that support incremental error detection in software models require non-trivial, manual annotations [6] which makes it difficult for software engineers to use them. This is particularly then a problem when software engineers like to define application/domain-specific design rules (e.g., derived from

requirements) or like to adapt the semantics of the underlying modeling language.

This paper introduces the *Model/Analyzer* tool which lets the software engineer define design rules arbitrarily without requiring manual annotations of any kind. The tool is implemented as a plug-in for the eclipse based IBM Rational Software Modeler. The *Model/Analyzer* is currently able to handle design rules written in Java and OCL (standard language without adaptions). The underlying approach for this tool is presented in [1]. The two main capabilities of the *Model/Analyzer* tool are 1) the detection of design errors in UML models and 2) the generation of fixing actions for eliminating the detected errors – mostly in form of abstract fixing actions (where to fix) [5] and in some cases also in form of concrete fixing actions (how to fix).

The detection of errors, their visualization, and generation of appropriate fixing actions is done incrementally and fully automatically without manual overhead. Design rules written in OCL are freely adaptable (add, modify/delete) [3] while the ones written in Java currently need to be compiled in. While our tool is implemented for UML models and OCL/Java design rules, the approach is generic [1, 2] and does not require annotations (neither model nor rule annotations, in contrast to Xiong et al.[6], where an own, OCL based language has been developed). The detection of inconsistencies and the generation of fixing actions is seamlessly integrated into the tool, like the approach from Nentwich et al. [4, 5], but our tool has the ability to 1) detect design errors very efficiently due to a fine granular understanding of the impact of model changes on design rules and 2) its generated fixing actions are very precise due to tailoring to specific inconsistencies based on the rule structure.

The *Model/Analyzer* tool has been evaluated on 24 UML models ranging from 100 model elements (small) to tens of thousands of model elements (very large) and 18 design rules. We demonstrated that our tool has near-instant response times and scales with the size of the model – both for detecting design errors and generating fixing actions.

2. TOOL AND EXPERIENCES

The core component of the *Model/Analyzer* is the evaluation mechanism for the design rules. A design rule is a condition on a model that needs to be satisfied for the model to be correct. Each design rule has to be defined for a specific context, to which the design rule applies. For example, a design rule that ensures that messages in a sequence diagram are declared as methods in their corresponding class diagrams needs to be evaluated for every message in a model – hence
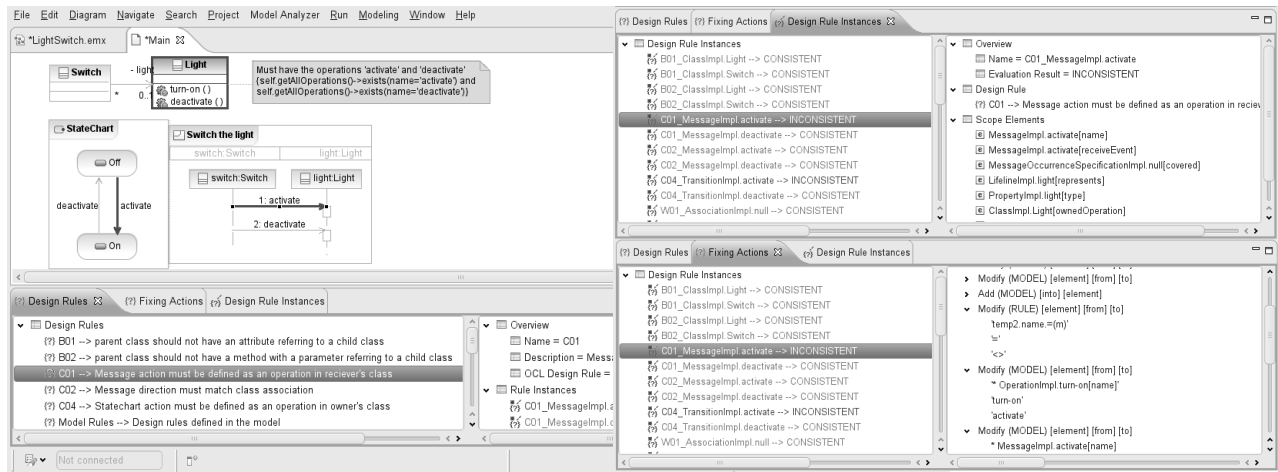
Figure 1: Views of the *Model/Analyzer* in the Rational Software Modeler

its context element is the UML message (a type of model element). For each message in the model an instance of such a design rule is created and evaluated (called rule instance). During each evaluation of a rule instance, the accessed model elements and its fields are observed and recorded - called the scope of a rule instance. This information is then used to identify how model changes affect the design rules. The key to incremental reasoning is the observation that only those rule instances need to be re-evaluated that previously accessed the changed model element. Since our approach investigates each evaluation of a rule instances separately, it is able to determine the exact impact of a model change for individual instances of design rules which greatly reduces the computational overhead.

Fixing actions are generated based on the structure of the violated design rule. If an error is found, the rule is parsed to understand the individual expressions it is made of and how these expressions contributed to the error. Note that design rules are generally written in a style that is similar to predicate logic and involves expressions such as *not*, *and*, *or*, *implies*, *forall*, *exists*, or property and operation calls of model elements. To determine what parts of the design rule contributes to the error, the *Model/Analyzer* tool evaluates all expressions individually to compare their (intermediate) results with their expected results. The result of the parsing and evaluation of its individual expressions is a tree which we call the evaluation tree. In essence the evaluation tree reflects the evaluation of an inconsistency with all its intermediate evaluation results and expectations. Out of this tree, the fixing actions are generated for those expressions where the evaluated result does not match the expected result – a very fine granular and precise approach that avoids false fixing actions.

Figure 1 depicts a screen shot of the modeling tool. The center shows the created model and is the default perspective of the RSM. Below the main window are the additional views added by the *Model/Analyzer* tool. The first view shows the freely definable design rules. The left side of that view lists currently defined design rules and the right side the details for a specific design rule. In this view, design rules are managed (added, removed and modified) although it is also possible to add/remove design rules for specific model

elements directly in RSM (see Figure 1 center, right of the class `Light`). A new design rule is evaluated immediately after its definition and the result of its evaluation is visualized in the corresponding views and in the model. Subsequent model changes are then evaluated incrementally.

The right side of Figure 1 shows the views for the rule instances and the generated fixing actions. On the left side of these views, the rule instances and their evaluation results are shown. The right side of the rule instance view shows a short description of the rule, the corresponding design rule and lists all the scope elements of this rule instance. The view for the fixing actions shows the action for a particular inconsistency. The fixing actions are depicted hierarchically as it may contain a list of alternatives and conjunctions. Three categories of actions are generated: Actions that are independent of the design rule structure (trivial actions), actions on the model (model actions), and actions on the rule (user definable rules my be erroneous much like model elements).

3. ACKNOWLEDGMENTS

4. REFERENCES

[1] A. Egyed. Instant consistency checking for the UML. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 381–390. ACM, 2006.

[2] A. Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *ICSE*, pages 793–796. IEEE Computer Society, 2007.

[3] I. Groher, A. Reder, and A. Egyed. Incremental Consistency Checking of Dynamic Constraints. In D. S. Rosenblum and G. Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2010.

[4] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.

[5] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *ICSE*, pages 455–464. IEEE Computer Society, 2003.

[6] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 315–324. ACM, 2009.