Softw Syst Model (2004) 3: 179–180 / Digital Object Identifier (DOI) 10.1007/s10270-004-0068-8

Editorial

Assessing model quality

Robert France, Bernhard Rumpe

Published online: 13 August 2004 –

Students in software engineering courses that cover modeling often ask some variant of the following question: "How do I know that my model is a good model?". It is not easy to provide a satisfactory response to this question. Good instructors provide students with some criteria and guidelines in the form of patterns (e.g., Craig Larman's GRASP patterns), rules of thumb (e.g., "minimize coupling, maximize cohesion", "keep inheritance depth shallow"), and exemplar models to better understand good modeling practices. While these help, the reality is that students ultimately rely on feedback from their instructors to determine the quality of their models. The instructors play the role of expert modelers and the students are their apprentices. The state of the practice in assessing model quality in the classroom and in industry seems to indicate that modeling is still in the craftsmanship phase.

Research on rigorous assessment of model quality has given us a glimpse of how we can progress to the next phase in which models are engineered. A number of researchers are working on developing rigorous static analysis techniques that are based on well-defined models of behavior. Articles on model-checking of modeled behavior published in SoSyM are a good reflection of the work in this area. Another promising area of research is systematic model testing (i.e., systematic dynamic analysis of modeled behavior). Systematic dynamic analysis of code (i.e., code testing) involves executing programs on a selected set of test inputs that satisfy some test criteria. These ideas can be extended to the modeling phases when models with operational semantics are used. Most educators in the modeling community have heard students gripe about their inability to animate or execute the models they have created in order to explore the behavior they have modeled. Model testing is concerned with providing modelers with this ability. Systematic model testing techniques provide opportunities for automating the testing process and for reusing tests. Systematic regression testing techniques in particular can enable more rigorous model evolution. The notion of model testing is not new. For example, SDL (Specification and Description Language) tools of provide facilities for exercising the state-machine based SDL models using an input set of test events. Work on executable variants of the UML also aims to provide modelers with feedback on the adequacy of their models. More recently a small, but growing, number of researchers have begun looking at developing systematic model testing techniques. This is an important area of research and helps pave the way towards more effective use of models during software development. There are a number of lessons from the systematic code testing community that can be applied, but the peculiarities of modeling languages also requires the development of new and innovative approaches. In particular, innovative work on defining effective test criteria that are based on coverage of model elements and on the generation of model-level test cases that provide desired levels of coverage is needed.

It is also useful to look at how other engineering disciplines determine the quality of their models. Engineers in other disciplines typically explore answers to the following questions when determining the adequacy of their models: Is the model a good predictor of how the physical artifact will behave? What are the (simplifying) assumptions underlying the model and what impact will they have on actual behavior? The answer to the first question is often based on evidence gathered from past applications of the model. Evidence of model fidelity is built up by comparing the actual behavior of systems built using the models with the behavior predicted by the models. Each time engineers build a system the experience gained either reinforces their confidence in the predictive power of the models used or the experience is used to improve the predictive power of models. Answers to the second

question allow engineers to identify the limitations of analyses carried out using the models and develop plans for identifying and addressing problems that arise when the assumptions are violated. Are similar questions applicable to software models? There are important differences between physical and software artifacts that one needs to consider when applying practices in other engineering disciplines to software, but there probably also exists some experience that can be beneficially applied to software modeling.

We can be sure that static analysis through context condition checking in various forms and dynamic checking through different kinds of testing strategies will be important parts of the newly emerging model engineering discipline.

Papers in this issue

The first two papers "**Dynamic Meta Modeling with time: Specifying the semantics of multimedia sequence diagrams**" by the authors *Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer* and "**Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3**" by the authors *Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca* are the second part of the Special section on graph transformations and visual modeling techniques. This special section was organized by guest editors *Paolo Bottoni* and *Mark Minas* and is explained in the corresponding Editorial of the last SoSyM Issue 2004/2.

In our series of expert voice papers, we this time are pleased to present a paper "**Use cases – Yesterday, today, and tomorrow**" by *Ivar Jacobson*, where he reflects on Use Case Diagrams originally invented by him in the early nineties. Since then Use Cases have become a vital part of the Unified Modeling Language and have been used in many projects to capture an overall view on the stakeholders and the pieces of functionality ("use cases") the system provides.

In his regular paper "**Plug-and-play composition of features and feature interactions with statechart diagrams**" the author *Christian Prehofer* presents an approach to separate highly entangled state-based systems into separate features, where individual features are described using Statecharts. He also provides a composition form based on graphical refinement relations between statecharts to allow constructing larger systems from smaller parts, as e.g. common in the telecommunication industry.

Tracking requirements down to their implementation is particularly important when evolving systems through changing requirements. Although this usually is the case in our long living software system, this tracking is also particularly difficult. In the paper "**Reconciling software requirements and architectures with intermediate models**" the authors *Paul Grünbacher, Alexander Egyed, and Nenad Medvidovic* provide such a tracking technique by using intermediate models that ease to track und understand the relationships between both worlds.

We hope you enjoy reading the articles in this issue,

Robert France, Bernhard Rumpe
Editors in Chief