Quality Assurance in MBE Back and Forth

Sebastian Gabmeyer

Institute of Software Technology and Interactive Systems
Vienna University of Technology, Vienna, Austria
gabmeyer@big.tuwien.ac.at

Abstract. The maturing of model-based engineering (MBE) has led to an increased interest and demand on the verification of MBE artifacts. Numerous verification approaches have been proposed in the past and, in fact, due to their diversity it is sometimes difficult to determine whether a suitable approach for the verification task at hand exists. In the first part of this tutorial, we thus present a classification for verification approaches of MBE artifacts that allows us to categorize approaches, among others, according to their intended verification goal and the capabilities of their verification engine. Based thereon, we briefly overview the landscape of existing verification approaches. In the second part, we iteratively build and verify the behavioral correctness of a small software system with our OCL-based model checker MocOCL.

1 Introduction

Software models and model transformations are the core development artifacts in model-based engineering (MBE) [7,8]. Following the MBE paradigm, programs are specified in terms of software models and the executable code and other deliverables are generated through successive applications of model transformations. Therefore, the correctness of the software models and the model transformations correlates directly with the correctness of the generated artifacts. Hence, errors made in the former most likely propagate to the latter. As a consequence, recent years have seen an impressive rise of verification approaches that aim to prevent the propagation of defects from the modeling to the code layer by verifying the system directly at the modeling layer. In fact, the diversity of the proposed approaches, which stems from, e.g., the different verification scenarios that the approaches support and the capabilities of their underlying verification engines, make it hard to overview existing verification approaches.

In the first part of this tutorial we will thus review the state-of-the-art of verification approaches for MBE artifacts and present a classification that will help us to categorize and compare these approaches [4,5]. We will focus, but not limit, our presentation to formal verification techniques and, wherever appropriate, highlight connections and possible applications to testing-based approaches. In particular, we will survey model checking and theorem proving-based approaches that

– check whether a set of models provides a consistent view onto the system,

– verify whether a transformation performs a semantically consistent conversion from the source to the target model, or
– assert whether the behavior of a model satisfies its specification.

Contrasting this theoretic presentation, the second part of the tutorial demonstrates an iterative, model-based development process that builds the implementation and specification of a small software system hand in hand. At the end of each iteration we assess the system's behavioral correctness by verifying that its implementation satisfies the specification. The software system's implementation consists of a MOF-conforming structural description and a set of model transformations that capture the system's behavior. The specification is formulated in cOCL [2], an extension that augments OCL [6] with temporal operators based on Computation Tree Logic (CTL) [3]. The verification is then conducted with MocOCL, a model checker for cOCL specifications [2].

2 Outline of the Tutorial

After motivating the need and usefulness of formal verification techniques in model-based engineering, we start with an introduction to the two predominant verification techniques, model checking and (interactive) theorem proving. Next, we analyze common verification tasks in MBE. These tasks can be categorized according to their *verification goal*. We will subsequently identify three such verification goals, namely *consistency checking*, *translation correctness*, and *behavioral correctness*. The identification of these goals leads naturally to a classification scheme for verification approaches. In the following, we amend this classification by additional categories. In particular, we refine the classification to distinguish approaches according to the input format of the problem description, that is, the type of software models they can verify. Further refinements of our classification allow us to categorize approaches according to the their verification technique, where we distinguish between model checking and theorem proving-based approaches. Final elaborations on the classification list different specification languages, that various approaches use to formulate/phrase verification properties, and formal encodings, which are generated from the inputted problem description and subsequently analyzed by the underlying verification engine. In this way, we develop a feature model that allows us to identify and compare suitable approaches for a given verification task quickly. Moreover, after categorizing existing approaches according to this feature-based classification we point out possible future research direction that have not or only partly been investigated according to our classification.

In the second part of the tutorial we present the explicit-state model checker MocOCL and show how it can be used to eliminate bugs in MBE artifacts. After a short discussion of its features, we will define structure and behavior of our running example. The system's static structure is captured by an Ecore model and its behavior is defined by model transformations. We will then formulate the cOCL specification and verify with MocOCL whether the modeled system satisfies this specification. The interpretation of the verification results will unveil, at first, that the specification is incorrect, and, after refining the specification, we notice that a bug in our implementation causes erroneous behavior. We fix the implementation and deem our model correct with respect to its specification after a final verification run.

3 cOCL and MocOCL

Among the insights that we derive during the discussion of existing verification approaches we also conclude that the majority of the presented approaches translates the models under verification into some lower level representation that corresponds to the input format of the underlying verification engine. For example, if the verification engine SPIN is used, the models are converted into PROMELA code. From a practical perspective, this requires to translate back the results produced by the verification engine to the modeling layer, and, from a theoretical perspective, it is necessary to verify whether the translation from model to low-level representation and back is indeed correct in order to trust the produced results. These considerations motivated the work on cOCL and MocOCL that allow us to verify the software system directly at the modeling layer; thus, translations between models and their verification representations are avoided.

In the following we introduce the concrete syntax of cOCL and discuss the verification workflow of our model checker MocOCL.

cOCL. The cOCL language extends OCL by five temporal operators, *Next*, *Globally*, *Eventually*, *Until*, and *Unless*, that must be preceded by a path quantifier, either *Always* or *Exists*. The semantics of these operators follow those defined for the branching-time logic CTL [3]. Expressions formulated with cOCL evaluate to Boolean values and must adhere to the following syntax definition:

$$(\mathit{Always} \mid \mathit{Exists})(\mathit{Next}\ \varphi \mid \mathit{Globally}\ \varphi \mid \mathit{Eventually}\ \varphi \mid \varphi\ \mathit{Until}\ \psi \mid \varphi\ \mathit{Unless}\ \psi)$$

where φ, ψ are either Boolean OCL expressions or cOCL expressions. An OCL invariant can only assert properties of single states, whereas a cOCL expression allows us to formulate assertions over a sequence, or *path*, of system states. Consider, for example, the Dining Philosophers problem. We formulate the following cOCL expression to check whether it is possible that we finally reach a state from which a path starts where one of the philosophers is always *hungry*:

```
philosphers→
        exists( p | Exists Eventually Exists Globally p.status = St :: hungry)
```

If this property is found to hold, it shows that there exists at least one philosopher that starves forever. Further, we can check safety properties and assert with the following expression that the number of forks remains constant:

```
let initialNumForks = forks→size() in
                    Always Globally forks→size() = initialNumForks
```

MocOCL. The model checker MocOCL verifies cOCL expressions. For this purpose, the modeler provides an Ecore (meta)model that defines the static structure of the system, a set of graph transformations that define the system's behavior, an initial model, and a cOCL expression as input. Starting with the initial model MocOCL applies the graph transformations iteratively to incrementally derive new states represented by graphs. At each iteration it evaluates the cOCL expression and, if the evaluation fails, produces a counter-examples. Otherwise, MocOCL continues its state space exploration until it either (*a*) concludes that the expression holds for the given system or (*b*) runs out of memory. In all but the last case, MocOCL reports back the *cause* of the verification that explains why the cOCL expression evaluated successfully or why it failed.

Currently, MocOCL uses Henshin [1] to apply graph transformations to Ecore models during the state space exploration. Moreover, it extends the OCL engine[1] integrated into Eclipse to evaluate cOCL expression over sequences of Ecore (instance) models. MocOCL is available for download at `http://modelevolution.org/prototypes/mococl`.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg, Germany (2010)
2. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL Model Checking. In: Cabot, J., Gogolla, M., Ráth, I., Willink, E.D. (eds.) Proceedings of the MODELS 2013 OCL Workshop. CEUR Workshop Proceedings, vol. 1092, pp. 13–22. CEUR-WS.org, Aachen, Germany (2013)
3. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg, Germany (1981)
4. Gabmeyer, S., Brosch, P., Seidl, M.: A Classification of Model Checking-Based Verification Approaches for Software Models. In: Proceedings of the STAF Workshop on Verification of Model Transformations (VOLT 2013). pp. 1–7 (2013)
5. Gabmeyer, S., Kaufmann, P., Seidl, M.: A feature-based classification of formal verification techniques for software models. Tech. Rep. BIG-TR-2014-1, Institut für Softwaretechnik und Interaktive Systeme; Technische Universität Wien (2014), `http://publik.tuwien.ac.at/files/PubDat_228585.pdf`
6. OMG, O.M.G.: Object Constraint Language (OCL) V2.2. `http://www.omg.org/spec/OCL/2.2/` (February 2010)
7. Selic, B.: The Pragmatics of Model-driven Development. Software, IEEE 20(5), 19–25 (2003)
8. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Softw. 20(5), 42–45 (Sep 2003), `http://dx.doi.org/10.1109/MS.2003.1231150`

[1] `http://projects.eclipse.org/projects/modeling.mdt.ocl`