MocOCL: A Model Checker for CTL-Extended OCL Specifications*

Robert Bill[1], Sebastian Gabmeyer[1], Petra Kaufmann[1], and Martina Seidl[2]

[1] Business Informatics Group
Vienna University of Technology
{bill,gabmeyer,kaufmann}@big.tuwien.ac.at
[2] Institute for Formal Models and Verification
JKU Linz
martina.seidl@jku.at

Abstract. We present the model checker MocOCL, a tool for model checking software models. The design rationale behind MocOCL is to close the gap between formal verification based on model checking and model-based engineering. Our approach avoids conversions that translate the software models into a format that a model checker can process. To this end, we implemented an explicit state model checker that directly processes the software model and verifies them against a specification formulated in a temporal extension of the constraint language OCL. MocOCL offers a web interface that interacts with the Eclipse Modeling Framework.

1 Introduction

We present the model checker MocOCL [2] which verifies whether a behavioral software model satisfies its specification. In contrast to other model checking approaches, MocOCL does not translate the software model to a formal representation processable by an off-the-shelf model checker, but performs the verification directly on the modeling level. Consequently, the results of the verification are presented at the level of the original model and not on a conversion of the models produced by the underlying verification engine.

The specifications are expressed in cOCL [2], an OCL extension enriched with temporal operators. With cOCL it is possible to define assertions for the current state of the system and, in addition, formulate requirements that have to hold in *some* or *all* future states. By using OCL we allow the modelers to formulate specifications at the modeling layer and in their common working environment.

The input for our MocOCL tool are hence an Ecore model that represents the static structure of the modeled system, graph transformations that specify its behavior, an instance model conforming to the Ecore model that captures

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018.

(`Always`|`Exists`) (`Next` ϕ|`Globally` ϕ|`Eventually` ψ| ϕ `Until` ψ| ϕ `Unless` ψ)

Fig. 2: Syntax of cOCL expressions

the system's initial state, and a specification formulated with cOCL. For the evaluation of such an expression, MocOCL implements an enumerative model checking approach that represents states explicitly. The results of the evaluation are then visualized in an informative user interface.

2 Running Example

For the tool demonstration, we use a variation of the well–known Pacman game. The game is played with a single Pacman and zero or more ghosts on a game board consisting of several fields, some of which host a treasure. At each turn, either Pacman or a ghost move to an adjacent field. The order of moves is not restricted, i.e., Pacman might move multiple times before a ghost moves. Pacman loses if he gets killed by a ghost, i.e., if both of them share the same field. Pacman wins if he finds a treasure without being killed by a ghost. Figure 1 depicts an initial state of the Pacman game.

Any implementation of this game should fulfill certain properties. The rules of the game require, for example, that *the game is over if Pacman reaches a treasure*. Other properties enforce that *Pacman must not get trapped* or that *Pacman is able to reach a treasure (initially)*. In the following, we will show how these properties are expressed using cOCL and how MocOCL evaluates these cOCL expressions on a certain initial state given as input model to find erroneous behavior.

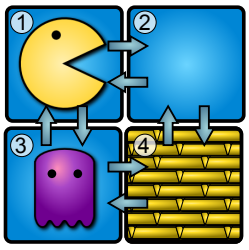

Fig. 1: Initial state

3 Using the MocOCL Framework

In this section we discuss our temporal, CTL-based extension of OCL, called cOCL, and the verification workflow of MocOCL using the running example presented in the previous section.

The cOCL *extension.* We extend the OCL standard with operators from the branching-time logic CTL. As shown in Fig. 2, φ, ψ are either Boolean OCL expressions or cOCL expressions, each of which evaluates to a Boolean value as well. In general CTL formulas are evaluated over sequences of states. In our case, the states correspond to Ecore model instances.

The semantics of these CTL operations are as follows. The path quantifier `Always` (`Exists`) requires the following temporal operator to evaluate to true in **all** (**at least one**) path(s) starting from the current state. The temporal

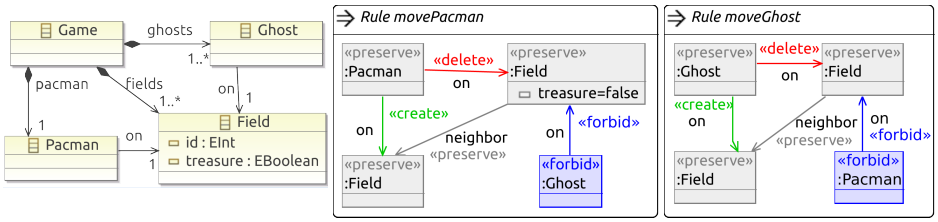

Fig. 3: Ecore model and Henshin transformation rules

operators `Next`, `Globally`, `Eventually`, `Until`, and `Unless` specify conditions on the path: `Next` ϕ requires ϕ to hold in the next state, `Globally` ϕ requires ϕ to hold in all states, `Eventually` ϕ requires ϕ to hold in at least one state and ϕ `Until` ψ requires ψ to hold eventually and ϕ holding before that, ϕ `Unless` ψ requires ϕ to hold forever or before ψ holds. For a more detailed overview on the cOCL language and property evaluation we kindly refer to [2].

Input preparation. The MocOCL framework verifies models of a system specified as an Ecore[3] model, an initial model instance, and graph transformations specified in Henshin.[4] Graphical editors are available for Henshin, and all Ecore models and instances thereof. In our Pacman game, we use two Henshin graph transformations as shown in Fig. 3 to describe the moves of Pacman and the ghost, respectively. The game is over if no more rules can be applied.

To verify the system, we need to formulate the system specification in cOCL. For example, to express that *the game is over if Pacman is on a treasure* we have to specify that in every state of the game if Pacman is on a treasure, there is no next state. This may be expressed in cOCL with **`Always Globally`** `self.pacman.on.treasure` **`implies`** `(`**`Always Next false`**`)` where the context of the expression is, by default, set to the root object of the initial model, that is, the `Game` class in this case.

The requirement that *Pacman does not get trapped* is equivalent to *Pacman might always change its position provided that the game is not over*. Thus, it may be expressed as cOCL property **`Always`** `(let x = pacman.on in` **`Exists Next`** `x <> pacman.on)` **`Unless`** `(`**`Always Next false`**`).`

The requirement that *it is possible that Pacman reaches all treasure fields at some point in time* is expressed as `fields->`**`select`**`(treasure)->`**`forAll`**`(f |` **`Exists Eventually`** `f = pacman.on).`

Usage. Once the input is prepared, MocOCL[5] loads the Ecore model, the initial model and the set of graph transformations as shown in Fig. 4. The prepared input files are selected via the respective input fields (1), (2), and (3). If necessary, only a subset of the rules defined in the Henshin file can be selected in (6a) and

[3] `http://www.eclipse.org/modeling/emf/?project=emf`
[4] `https://www.eclipse.org/henshin/`
[5] Available for download at `http://modelevolution.org/prototypes/mococl/` with a simplified version for the Pacman example for Google Chrome being usable online at `http://modelevolution.org/mococl/`

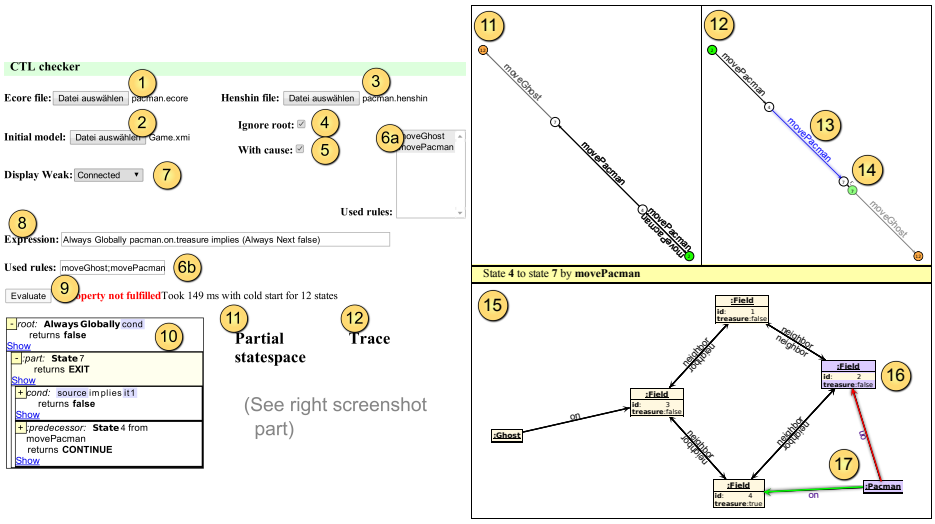

Fig. 4: The MocOCL state space and model view

(6b). Moreover, the user may choose to hide the Ecore model's root container (4). The tool's verbose output is useful for understanding verification result (see below), yet, its generation might consume a lot of memory. Thus, the user may opt to restrict the tool's output to a simple message that reports whether the property is fulfilled or not by deselecting checkbox (5). After the user has entered the cOCL expression into input field (8) and clicked on the *Evaluate* button (9), all files and the expression are sent to the MocOCL server which evaluates the expression and sends back the result to the Web GUI.

The verification result (9) is displayed next to the *Evaluate* button together with some performance information. The tree view (10) to the left of the output (9) shows the verification result together with its cause. The *cause* explains why the expression evaluated to either *true* or *false*. The cause of a CTL expression consists of the states leading to the result. If a counter example trace is generated, the cause consists of the last state of the trace. The cause of a state consists of the expressions evaluated in the state and predecessor and successor states. Note that due to the iterative state space exploration strategy, some states not relevant for the result are omitted. The state space is represented in two different views displayed in the right part of Fig. 4. The first view, the partial state space view (11), shows all states occurring in the evaluation tree together with all known transitions among these states. In the second view, the trace view (12), the states of the evaluation tree are visualized as a trace leading from the initial to the final states.

Initial states, i.e., states without incoming transitions, are highlighted in green. Final states, i.e., states without outgoing transitions, are highlighted in orange. Transitions between two states that result from evaluating a sub–expression are displayed semi-transparent and are connected to its immediate

parent–level expression using ϵ–transitions to differentiate between relevant parts of the parent- and its sub–expressions. If the size of the displayed state space together with all sub–expression transitions grows too large, these transitions can be hidden by changing the setting (7) from `Connected` to `none`.

Upon selecting a state or transition in the state space or trace view, the state or transition is highlighted (13) and its model instance is displayed in the state–visualization area (15). There, those parts of the model instance that occur in sub–expressions are highlighted in purple (16). Associations in red (green) indicate that these associations are deleted (added) (17) when transitioning from the source to the target state.

The result of the verification can be interpreted as follows. Recall that we wanted to ensure that the game is over if Pacman has found the treasure. The highlighted transition indicates that Pacman has really found the treasure, but the ghost is still able to move. In fact, our `moveGhost` transition only forbids the ghost from moving if he has killed Pacman, but not if Pacman has won the game.

4 Related Work

Several model checkers for graph-based languages have been proposed in the literature. CheckVML verifies systems defined by UML-like class diagrams whose behavior is captured by a set of graph transformations against safety properties provided as *property graphs* [8]. Groove [3] models the static structure of a system with attributed, typed graphs and, like CheckVML and MocOCL, defines the system's operations with graph transformations. Desirable properties are expressed with graph constraints which can be embedded in CTL or LTL specifications. Because Groove does not support quantifiers some temporal specifications defined in Groove grow linearly with the size of the model or require additional constructs to track the existence of model elements. Recently, neighborhood-based abstraction techniques have been implemented in Groove to reduce the size of the generated state space [7]. Henshin [1], an Eclipse plugin to build graph transformations for Ecore models, produces the state space starting from a user-provided initial model by applying a set of graph transformations recursively to the initial and all subsequently derived models. It supports the verification of safety properties specified as OCL invariants. Systems with infinitely many states can be verified with Augur2 [4]. It applies abstraction techniques and produces a so-called Petri graph that overapproximates the set of reachable states. Augur2 models systems with hypergraphs.

The SOCLe tool verifies systems that are specified by a class diagram and a set of state machines, which describe the behavior of each class, against an EOCL specification [6]. Like cOCL, EOCL extends OCL with CTL operators but its formal semantics are not aligned with those of OCL. Al Lail *et al.* [5] present a bounded model checker for specifications written in the LTL-inspired OCL extension TOCL [9].

5 Conclusion and Future Work

In this paper we present MocOCL, a model checker for specifications formulated in cOCL, an CTL–extension of OCL [2].We showcase the concrete syntax of cOCL and the verification workflow of MocOCL.

Similar to Groove, where the system's model and its specification are described with attributed graphs, MocOCL allows its users to express the verification problem at the level of the models that capture structure and behavior of a system. This and the fact that cOCL extends OCL, which is presumably well-known to modelers, lead to our conjecture that MocOCL bridges the gap between model checking and model-based engineering. The validation of this conjecture is subject to a future user study. In contrast to more mature tools the performance of MocOCL requires improvements. For example, MocOCL asserts that Pacman can win the game on a board with 32 fields and 2 ghosts after 33,65 seconds while Groove requires only 21,65 seconds on a i5-2410M machine with 2.3 GHz and 8 GB ram.

In future work we plan to switch to an incremental evaluation algorithm for cOCL specifications and remove expensive isomorphism checks currently necessary during the exploration of the state space. Further, the integration of MocOCL into Eclipse and an improved user interface are work in progress.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Model Driven Engineering Languages and Systems. LNCS, vol. 6394, pp. 121–135. Springer (2010)
2. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL Model Checking. In: Proceedings of the MODELS 2013 OCL Workshop. CEUR Workshop Proceedings, vol. 1092, pp. 13–22. CEUR-WS.org (2013)
3. Kastenberg, H., Rensink, A.: Model Checking Dynamic States in GROOVE. In: Model Checking Software. LNCS, vol. 3925, pp. 299–305. Springer (2006)
4. König, B., Kozioura, V.: Augur2 - A New Version of a Tool for the Analysis of Graph Transformation Systems. Electr. Notes Theor. Comput. Sci. 211, 201–210 (2008)
5. Lail, M.A., Abdunabi, R., France, R., Ray, I.: An Approach to Analyzing Temporal Properties in UML Class Models. In: Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa 2013). CEUR Workshop Proceedings, vol. 1069, pp. 77–86. CEUR-WS.org (2013)
6. Mullins, J., Oarga, R.: Model Checking of Extended OCL Constraints on UML Models in SOCLe. In: Formal Methods for Open Object-Based Distributed Systems. LNCS, vol. 4468, pp. 59–75. Springer (2007)
7. Rensink, A., Zambon, E.: Neighbourhood Abstraction in GROOVE. ECEASST 32, 13 (2010)
8. Schmidt, Á., Varró, D.: CheckVML: A Tool for Model Checking Visual Modeling Languages. In: UML 2003 - The Unified Modeling Language and Applications. LNCS, vol. 2863, pp. 92–95. Springer (2003)
9. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Perspectives of Systems Informatics. LNCS, vol. 2890, pp. 351–357. Springer (2003)