Model Checking of CTL-Extended OCL Specifications*

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

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

Abstract. In software modeling, the Object Constraint Language (OCL) is an important language to specify properties that a model has to satisfy. The design of OCL reflects the structure of MOF-based modeling languages like UML and its tight integration results in an intuitive usability. But OCL allows to express properties only in the context of a single instance model and not with respect to a sequence of instance models that capture the execution of the system.
In this paper, we show how OCL can be extended with CTL-based temporal operators to express properties over the lifetime of an instance model. We formally introduce syntax and semantics of our OCL extension cOCL. The properties specified with our OCL extension can be verified with our explicit state space model checking framework, called MocOCL. In a case study, we illustrate the expressiveness and usability of our approach and evaluate the performance of our implementation.

1 Introduction

In software and hardware verification [9,14,18], *model checking* is currently one of the most widely used verification techniques to show that a system satisfies its specification.[3] Model checking requires a formal representation of the system and a specification that often consists of a set of temporal logic formulas formulated in, e.g., the branching-time logic CTL [6].

In the context of model-based engineering (MBE), software models[4] are the core artifacts to specify and develop a system. In contrast to traditional software engineering, where models mainly serve as design artifacts during the early project phases, an MBE project uses models at every stage of the development process and finally generates executable code and other deliverables therefrom.

* This work is supported by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018.

[3] Usually, a specification consists of a set of properties that the system should satisfy. We will, however, often use the terms *specification* and *property* interchangeably.

[4] The term *model* is heavily overloaded in computer science. We encounter logical models in the context of model checking and software models in the context of MBE. In case of ambiguities we use the term *software model* when referring to the latter.

Hence, the correctness of the models is a prerequisite for the correctness of the system that is presented to the end-user [25]. Consequently, formal verification techniques find their way into the MBE processes to help detect and avoid errors in the models. A popular choice for this task is model checking. Recent works and tools like Hugo/RT [19], Groove [17], and Proco [15], to name but a few, show that software models can be verified with model checking. In general, the verification of software models by model checking abides the following scheme. Throughout its lifetime a system, which is described by the software model, passes through many states; each such state is represented by a distinct instance model. A sequence of states, called a *trace*, describes the execution of the system from an initial to some intermediate or final state. The system's specification describes the set of allowed, i.e., *valid*, traces. A model checker then verifies whether the execution traces of the system starting from a given initial state are a subset of the valid traces described by the specification. If the model checker determines a violating trace, it reports the found *counterexample trace* to the user.

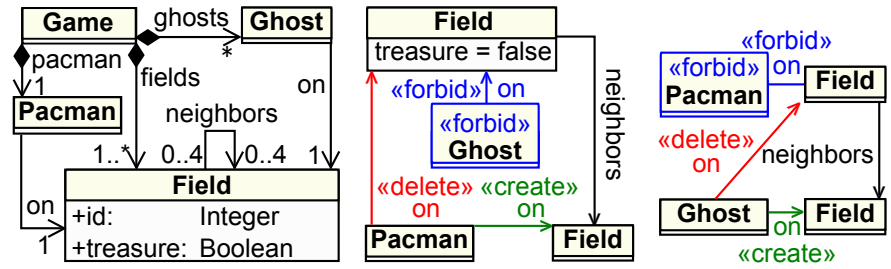
Currently, many approaches use an off-the-shelf model checker and require the modeler to express the specification in the language of this model checker. Therefore, the modeler is required to study and understand the translation of the system's software model to the model checker's input format. Moreover, the specification is often expressed in a language that is different from the languages available in the modeling environment. To overcome this drawback, we present a CTL-based temporal extension for the Object Constraint Language (OCL), called cOCL. While OCL can only express constraints on a single instance model, cOCL can formulate constraints over sequences of instance models representing execution traces of the system. For verifying properties expressed in cOCL, we realized the model checker MocOCL.

The structure of this paper is as follows. We introduce a motivating example in Section 2 and explain the core ideas behind our approach. In Section 3, we present the syntax and semantics of our CTL-based OCL extension and introduce our model checking framework in Section 4 together with numerous examples in the concrete syntax of our model checker. We then discuss the model checking algorithm as well as some realization details of our tool. In Section 5, we present a first evaluation of our approach regarding its usability and performance. Finally, we review related approaches in Section 6 and conclude with an outlook to future work. This paper is a substantial revision of the extended abstract [3] presented at the OCL 2013 workshop. An unabridged version that includes the questionnaire of the case study (Sec. 5) is also available [4].

2 Motivating Example

To motivate the work presented in this paper we use a variation of the well-known Pacman game,[5] which we use due to its intuitiveness and its easy adaptability to larger game instances, i.e., increasing board size and number of ghosts, to test the scalability of our model checking algorithm.

[5] `http://en.wikipedia.org/wiki/Pac-Man`



(a) Class Diagram of Pacman's World

(b) "Move Pacman"-Rule

(c) "Move Ghost"-Rule

Fig. 2. Implementation of the Pacman game with graph transformations.

Structure and Game Play. The game is played on a board consisting of square fields, each of which has at most four neighboring fields. Each field has a unique ID and some fields contain a treasure, indicated by a Boolean flag. Pacman plays against one or more ghosts. Each player, Pacman or the ghosts, is placed on one field of the board. The static structure of the game's implementation is shown in Figure 2a. Figure 1 uses the graphical syntax and shows a Pacman game instance with four fields, a treasure on field 4, Pacman on field 1, and a ghost on field 3. The game is played as follows. The players move turn-wise in no fixed order. Pacman has to find one of the treasures, which are placed somewhere on the board. If he finds one, he wins the game. If, however, Pacman moves onto a field with a ghost or if a ghost moves onto Pacman's field, Pacman looses the game.

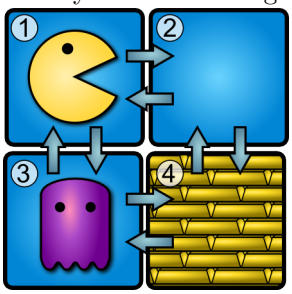


Fig. 1. Pacman's World.

Implementation. We use graph transformation rules to implement the behavior of the game. The first rule, *Move Pacman*, is depicted in Figure 2(a) and describes one move of Pacman. The second rule, *Move Ghost* (Figure 2(b)), describes one move of a ghost. Pacman and the ghosts are only allowed to move if the game is not over yet, that is, no one moves if Pacman is on a treasure field or if Pacman and a ghost meet on the same field. Note, however, that the first restriction is not enforced by the *Move Ghost* rule; hence, ghosts may still move if Pacman already found the treasure. In Section 4.2, we show how to detect this violation of the rules. Figure 3 exemplarily illustrates two applications of the *Move Pacman* graph transformation and the subsequent changes to the current state. First, Pacman moves from field 1 to field 2 and in the next round Pacman moves from field 2 to field 4, which contains the treasure. In this scenario, Pacman wins.

Verification Tasks. In the example above, we have seen one specific trace showing a winning strategy for Pacman. Yet, if we want to verify that the game always terminates when Pacman found a treasure, it is not enough to consider only some specific traces but all possible traces have to be explored.

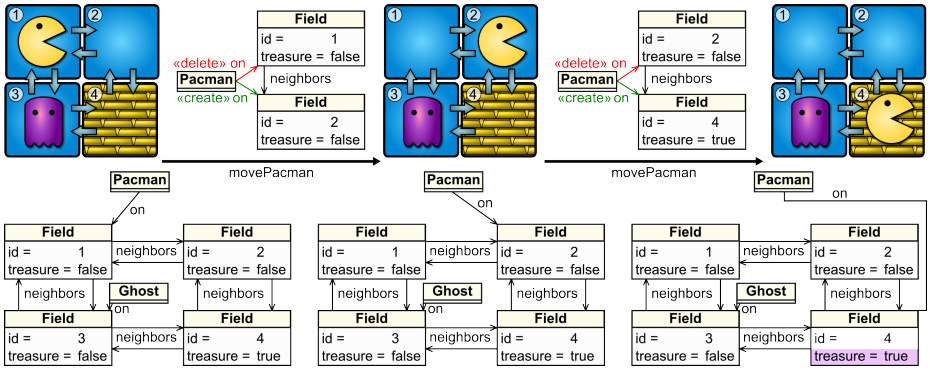


Fig. 3. Example for Transformations.

For expressing and solving these verification tasks, temporal aspects of the system behavior have to be considered. Such verification questions are difficult to express in OCL because it neither provides operators to express constraints that must hold, e.g., *always* or *eventually*, nor the semantic notions to describe execution traces. To this end, we propose to use our OCL extension cOCL.

MocOCL at a Glance. Our tool MocOCL realizes an explicit state model checking approach. We construct the state space of the Pacman game iteratively. In our implementation, we use the graph transformation tool Henshin [1] that explores the state space by recursively applying all matching graph transformation rules to the user-provided initial model. The full state space resulting from recursively applying the rules *Move Pacman* and *Move Ghost* to the initial model (Fig. 1) is depicted in Figure 4. The *initial state* in the bottom-left corner of the figure is highlighted in green with a bold border and the end states are marked red with a dashed border. The transitions between the states show possible moves of Pacman and the ghost. Overall there are $4 * 4 = 16$ different states (the ghost has to be placed on each field and Pacman has to be placed on each field). After each exploration step MocOCL evaluates the cOCL expression and, if enough states have been explored to conclude that the expressions either holds or fails, the verification stops. Finally, MocOCL returns a report that explains the result of the verification.

3 A Temporal Extension of OCL

In this section, we formally introduce syntax and semantics of cOCL, which extends OCL with CTL operators. We assume familiarity with model checking and CTL [2,7]. We integrate cOCL into the formal semantics of OCL and kindly refer to the work of Richters and Gogolla [24] for the details on the syntax and semantics of OCL. Due to space constraints and for ease of presentation, we reproduce only those definitions that are essential to the understanding of the subsequent explanations.

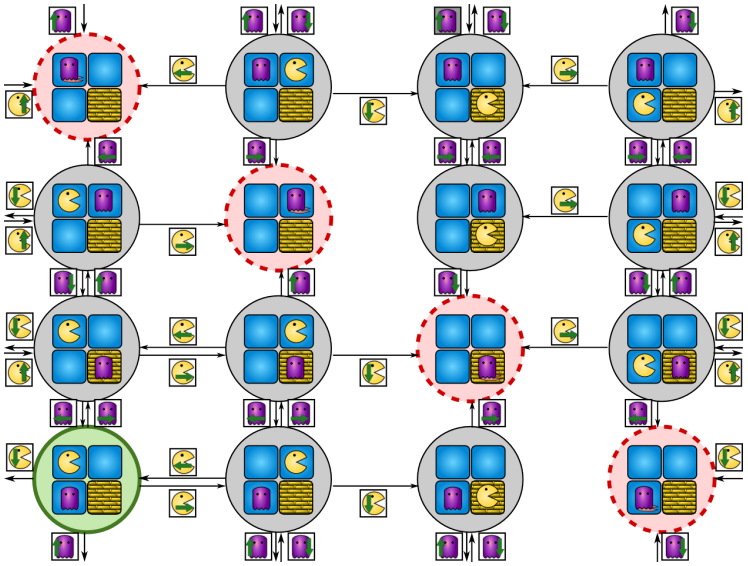


Fig. 4. State Space of the Pacman Game.

OCL expressions are always defined w.r.t. a model M consisting of classes which are described by their attributes and operations as well as associations between classes characterized by multiplicities and roles. Such a model provides the basis for defining OCL expressions in form of a signature $\Sigma_M = (\mathcal{T}_M, \Omega_M, \mathcal{V})$ where $\mathcal{T}_M$ is a set of types, Ω_M is a set of operations, and $\mathcal{V}$ is additionally a set of variables. By $V_t \subseteq \mathcal{V}$ we denote the set of variables of type $t \in \mathcal{T}_M$. The instantiation of such a model is given by objects, links, and attribute values and is also called *snapshot*. In the following, we denote a specific snapshot of a model M by $\sigma(M)$. Due to space restrictions, we abstain from a complete formal introduction of the notion of model.

Definition 1 (Syntax of OCL). *Let $\Sigma_M = (\mathcal{T}_M, \Omega_M, \mathcal{V})$ be the signature of model M as described above. Then Expr_t is the set of expressions of type t defined as follows.*

i. *If $v \in \mathrm{Var}_t$ then $v \in \mathrm{Expr}_t$.*
ii. *If $v \in \mathrm{Var}_{t_1}, e_1 \in \mathrm{Expr}_{t_1}, e \in \mathrm{Expr}_t$ then $(\mathsf{let}\ \ v = e_1\ \ \mathsf{in}\ e) \in \mathrm{Expr}_t$.*
iii. *If $\omega : t_1 \times \ldots \times t_n \to t \in \Omega_M$ and $e_i \in \mathrm{Expr}_{t_i}$ then $\omega(e_1, \ldots, e_n) \in \mathrm{Expr}_t$.*
iv. *If $e_1 \in \mathrm{Expr}_{Bool}$ and $e_2, e_3 \in \mathrm{Expr}_t$ then $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \mathsf{endif} \in \mathrm{Expr}_t$.*
v. *If $e_1 \in \mathrm{Expr}_{Collection(t_1)}, v_1 \in \mathrm{Var}_{t_1}, v_2 \in \mathrm{Var}_t$, and $e_2, e_3 \in \mathrm{Expr}_t$ then $e_1 \to \mathsf{iterate}(v_1; v_2 = e_2 \mid e_3) \in \mathrm{Expr}_t$.*

The set of OCL expressions over Σ denoted by OCL_M is given by $\bigcup_t \mathrm{Expr}_t$.

Due to space restrictions, Definition 1 does not contain the definitions related to type hierarchies and inheritance. Adding these definitions neither changes nor impacts our approach presented below.

Definition 2 (Syntax of cOCL). *Let M be a model with OCL expressions OCL_M. Then cOCL is defined as follows.*

i. If $e \in \mathrm{OCL}_M$, then $e \in cOCL$.
ii. Let $\mathrm{Expr}_{Bool} \subseteq cOCL$ be the set of boolean expressions in cOCL. If $e_1, e_2 \in \mathrm{Expr}_{Bool}$ then $\mathsf{AX}e_1$, $\mathsf{EX}e_1$, $\mathsf{A}e_1\mathsf{W}e_2$, $\mathsf{E}e_1\mathsf{W}e_2$, $\mathsf{A}e_1\mathsf{U}e_2$, $\mathsf{E}e_1\mathsf{U}e_2 \in \mathrm{Expr}_{Bool}$.

Our extension introduces three temporal operators, *next* (X), *weak until* (W), and (strong) *until* (U), which are quantified either existentially (E) or universally (A). We define two additional operators, *eventually* (F) and *globally* (G), by the following equivalences: $\mathsf{EF}\varphi \equiv \mathsf{E}\,\mathit{true}\,\mathsf{U}\,\varphi$ and $\mathsf{AF}\varphi \equiv \mathsf{A}\,\mathit{true}\,\mathsf{U}\,\varphi$, and $\mathsf{EG}\varphi \equiv \mathsf{E}\,\varphi\,\mathsf{W}\,\mathit{false}$ and $\mathsf{AG}\varphi \equiv \mathsf{A}\,\varphi\,\mathsf{W}\,\mathit{false}$. Note that *next*, *eventually*, and *globally* have a single subformula as argument, whereas the *weak until* and *until* operators have two subformulas. Before we define the semantics of the temporal operators, we formally introduce the term *transition system* which describes all possible executions of a system.

Definition 3 (Transition System). *The transition system $\mathcal{TS}_M$ associated with a model M is a hextuple $(\mathcal{S}, \iota, T, \mathcal{A}, \mathcal{B}, \mathcal{E})$ consisting of a set $\mathcal{S}$ of states, an initial state $\iota \in \mathcal{E}$, a transition relation $T \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$, a set $\mathcal{A}$ of actions, a set $\mathcal{B}$ of variable assignments, and the environment relation $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{B}$. An environment $\tau \in \mathcal{E}$ is a pair (σ, β) with $\sigma \in \mathcal{S}$ and $\beta \in \mathcal{B}$.*

For each state $\sigma \in \mathcal{S}$ the set of possible objects is given by σ_{class}, the set of possible associations by σ_{assoc}, and the set of possible attributes by σ_{attrs}. A variable assignment is a function $\beta : Var_t \to Val_t$ that, given a variable name, returns the current value of the associated variable of type t. An action is a partial function $\alpha : \sigma_{class} \to \sigma_{class} \cup \{\bot\}$ mapping objects from one state to corresponding objects of another state or to $\bot$ if no such object exists.

The concept of an *environment* $\tau = (\sigma, \beta)$ has been introduced in [24]. For specific execution traces, we define the term *path* as follows.

Definition 4 (Path). *Let $\mathcal{TS}_M = (\mathcal{S}, \iota, T, \mathcal{A}, \mathcal{B}, \mathcal{E})$ be the transition system associated with a model M. A* path *π is a finite or infinite sequence of environments $(\tau_0\tau_1\tau_2\ldots)$ with $\tau_i \in \mathcal{E}$ and $(\tau_i, \tau_{i+1}) \in \{((\sigma_i, \beta_i), (\sigma_{i+1}, \beta_{i+1})) \mid \beta_{i+1} = \mathrm{mapvar}(\beta_i, \alpha), (\sigma_i, \alpha, \sigma_{i+1}) \in T\}$ for all $0 \leq i$. For a path $\pi = (\tau_0\tau_1\tau_2\ldots)$, we define the projection function $\pi(i) = \tau_i$. The length of a path $|\pi| = n$ for finite paths $\pi = (\tau_0 \ldots \tau_n)$, and $|\pi| = \infty$ for infinite paths $\pi = (\tau_0\tau_1\tau_2\ldots)$. By $\mathsf{pth}(\mathcal{TS}_M)$ we denote the set of all possible paths of $\mathcal{TS}_M$.*

The function **mapvar** : $\mathcal{B} \times \mathcal{A} \to \mathcal{B}$ takes a variable assignment β_s of source state σ_s and an action $\alpha \in \mathcal{A}$ and updates β_s with respect to α resulting in a variable assignment β_t for the successor state σ_t. We are now able to define the semantics of cOCL as follows.

Definition 5 (Semantics). *Let $\mathcal{TS}_M = (\mathcal{S}, \iota, T, \mathcal{A}, \mathcal{B}, \mathcal{E})$ be the transition system associated with model M as defined above. The semantics of a cOCL expression w.r.t. a context $\tau \in \mathcal{B}$ with $\tau = (\sigma, \beta)$ is defined by the rules i.–vi. originating from Definition 2 of [24] and the additional rules vii.–xi. for the temporal extension.*

i. $I[\![v]\!](\tau) := \beta(v)$

ii. $I[\![\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2]\!](\tau) := I[\![e_2]\!]((\sigma, \beta[v/I[\![e_1]\!](\tau)]))$

iii. $I[\![\omega(e_1, \ldots, e_n)]\!](\tau) := I(\omega)(\tau)(I[\![e_1]\!](\tau), \ldots, I[\![e_n]\!](\tau))$

iv. $I[\![\,\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \mathrm{e}_3\ \mathsf{endif}]\!](\tau) = \begin{cases} I[\![e_2]\!](\tau) & \text{if} I[\![e_1]\!](\tau) := \mathit{true} \\ I[\![e_3]\!](\tau) & \text{if} I[\![e_1]\!](\tau) := \mathit{false} \\ \bot & \text{otherwise} \end{cases}$

v. $I[\![e_1 \to \mathsf{iterate}(v_1; v_2 = e_2 | e_3)]\!](\tau) := I[\![e_1 \to \mathsf{iterate}'(v_1 | e_3)]\!](\tau')$ *where* $\tau' = (\sigma, \beta')$ *and* $\tau'' = (\sigma, \beta'')$ *are environments with modified variable assignments:* $\beta' := \beta[v_2/I[\![e_2]\!](\tau)]$, $\beta'' := \beta'[v_2/I[\![e_3]\!](\sigma, \beta'[v_1/x_1])]$

vi. $I[\![\mathsf{A}\, e_1\, \mathsf{U}\, e_2]\!](\tau) := \{\pi \mid \pi \in \mathsf{pth}(\mathcal{TS}_M), \pi(0) = \tau, \nexists n \in \mathbb{N}, n \leq |\pi| : (I[\![e_2]\!](\pi(n)) = \text{true} \wedge \forall\, 0 \leq i < n : I[\![e_1]\!](\pi(i)) = \text{true})\} = \emptyset$

vii. $I[\![\mathsf{E}\, e_1\, \mathsf{U}\, e_2]\!](\tau) := \{\pi \mid \pi \in \mathsf{pth}(\mathcal{TS}_M), \pi(0) = \tau, \exists n \in \mathbb{N}, n \leq |\pi| : (I[\![e_2]\!](\pi(n)) = \text{true} \wedge \forall\, 0 \leq i < n : I[\![e_1]\!](\pi(i)) = \text{true})\} \neq \emptyset$

viii. $I[\![\mathsf{A}\, e_1\, \mathsf{W}\, e_2]\!](\tau) := \{\pi \mid \pi \in \mathsf{pth}(\mathcal{TS}_M), \pi(0) = \tau, \nexists n \in \mathbb{N}, n \leq |\pi| : (I[\![e_1]\!](\pi(n)) = \text{false} \to \exists i \in \mathbb{N}, i \leq n : I[\![e_2]\!](\pi(i)) = \text{true})\} = \emptyset$

ix. $I[\![\mathsf{E}\, \phi\, \mathsf{W}\, \psi]\!](\tau) := \{\pi \mid \pi \in \mathsf{pth}(\mathcal{TS}_M), \pi(0) = \tau, \exists n \in \mathbb{N}, n \leq |\pi| : (I[\![e_1]\!](\pi(n)) = \text{false} \to \exists i \in \mathbb{N}, i \leq n : I[\![e_2]\!](\pi(i)) = \text{true})\} \neq \emptyset$

x. $I[\![\mathsf{E}\, \mathsf{X}\, e]\!](\tau) := \{\pi \mid \pi \in \mathsf{pth}(\mathcal{TS}_M), \pi(0) = \tau \wedge |\pi| \geq 1 \wedge I[\![e]\!](\pi(1)) = \text{true}\} \neq \emptyset$

xi. $I[\![\mathsf{A}\, \mathsf{X}\, e]\!](\tau) := \{\pi \mid \pi \in \mathsf{pth}(\mathcal{TS}_M), \pi(0) = \tau \wedge |\pi| \geq 1 \wedge I[\![e]\!](\pi(1)) \neq \text{true}\} = \emptyset$

The semantics of the *eventually* and *globally* operators follow directly from the above definitions. The only free variable allowed in cOCL expressions is `self`, corresponding to the root object of the state, called $root(\sigma)$ in the following, the cOCL expression is evaluated in. We define *satisfiability* of cOCL expressions as follows.

Definition 6 (Satisfiability). *A cOCL expression* ϕ *is* satisfiable *w.r.t. a transition system* $\mathcal{TS}_M$ *with initial state* ι *iff* $I[\![\phi]\!](\iota) = \text{true}$.

In the remainder of this paper, we discuss how the model checker MocOCL verifies cOCL specifications, discuss its implementation, and evaluate its performance and usability. In Table 1 we list examples of cOCL expressions, which illustrate typical application scenarios of cOCL in the context of the `Game` class (Fig. 2a) of the previously introduced Pacman game. The cOCL expressions are phrased in the concrete syntax of MocOCL that we introduce in the next section.

4 The Model Checker MocOCL

The implementation of MocOCL consists of two parts, a backend that realizes an explicit state model checker and a graphical user interface.

4.1 Backend

The backend consists of a parser for the textual concrete syntax of cOCL and the model checker MocOCL that verifies cOCL specifications.

Table 1. Examples of cOCL expressions in the concrete syntax of MocOCL.

natural language	**cOCL expression**
Initially, there is a field containing a treasure.	`self.fields->exists(field | field.treasure)`
The game is over/not over.	`Always Next false/Exists Next true`
The game will surely be over sometimes.	`Always Eventually (Always Next false)`
Pacman will find the treasure in all cases.	`Always Eventually self.pacman.on.treasure`

If the treasure is next to Pacman, he can always find it in the next turn.	`Always Globally`
	`  self.pacman.on.neighbor->exists(field.treasure)`
	`   implies (Exists Next self.pacman.on.treasure)`

As long as not all fields next to Pacman are occupied by ghosts, there is a possibility that the game is not over after the next turn.	`Always Globally`
	`  self.pacman.on.neighbor->exists(field |`
	`  self.ghosts->forAll(g | field <> g.on) implies`
	`  (Exists Next (Exists Next true)))`

As long as the game is not over, every ghost may move to at least two different positions.	`Always self.ghosts->`
	`  forAll(g | g.on.neighbor->select(field |`
	`  Exists Next g.on = field)->size() >= 2)`
	`Unless (Always Next false)`

The *concrete syntax* enhances the readability of cOCL expressions. It allows us to write the temporal operators in their familiar long forms, i.e., $\mathsf{X}\varphi$, $\mathsf{F}\varphi$, $\mathsf{G}\varphi$, $\varphi\,\mathsf{W}\,\psi$, and $\varphi\,\mathsf{U}\,\psi$ become **Next** φ, **Eventually** φ, **Globally** φ, φ **Unless** ψ, and φ **Until** ψ. The universal and existential path quantifiers preceding the temporal operators become **Always** and **Exists** or, alternatively, **Sometimes**. Table 1 shows examples of MocOCL expressions in the concrete syntax. In our implementation, we extended the concrete syntax of OCL given by an Xtext grammar[6] resulting in an editor with syntax highlighting for cOCL expressions and a Java API.

The prototypical, EMF-based[7] implementation of the MocOCL model checker performs the actual verification task as follows. Given an Ecore-conformant model, an instance model that represents the system's initial state, a set of model transformations, and a cOCL specification, MocOCL generates the state space

[6] `http://www.eclipse.org/Xtext/`
[7] `http://www.eclipse.org/modeling/emf/`

iteratively and, at every step, it verifies the cOCL specification on-the-fly. Finally, it reports to the modeler information on the reason of the verification result.

In MocOCL, the state space consists of a set of graphs. Each graph corresponds to an instance of the system and thus represents a system's state at a discrete point in time. Given a graph transformation system $\mathcal{G} = (\mathcal{R}, \iota)$ with graph rewrite rules $\mathcal{R}$ and an initial state ι, the function $\mathbf{step}_{\mathcal{R}}\colon \mathcal{S} \to P(\mathcal{S} \times \mathcal{M})$[8] handles the step-wise exploration of the state space where $\mathcal{S}$ denotes the set of all states and $\mathcal{M}$ the set of all partial mappings between states $\sigma_1, \sigma_2 \in \mathcal{S}$. It expects as input a state σ_s and returns a set of pairs (σ_t, m) where σ_t denotes the successor state of σ_s and $m \in \mathcal{M}$ defines a morphism $m : \sigma_{Class} \to \sigma_{Class} \cup \{\bot\}$ that maps objects in σ_s to corresponding objects in σ_t or to $\bot$ if no such object exists. The successor state σ_t is obtained from σ_s by applying a rewrite rule $r \in \mathcal{R}$ to the graph represented by σ_s. We write $\sigma_s \overset{r,m}{\Rightarrow} \sigma_t$ to denote that σ_s is rewritten to σ_t by rule $r \in \mathcal{R}$ at match m [11]. The state space exploration function is then defined as $\mathbf{step}_{\mathcal{R}}(\sigma_s) = \bigcup_{r \in \mathcal{R}} \{(\sigma_t, m) | \sigma_s \overset{r,m}{\Rightarrow} \sigma_t\}$. The helper function $\mathbf{succ}\colon \mathcal{E} \to P(\mathcal{E})$ returns all environments reachable by a transition from the source environment $\tau_s = (\sigma_s, \beta_s)$ and is defined by $\mathbf{succ}((\sigma_s, \beta_s)) := \{(\sigma_t, \beta_t) | (\sigma_t, m) \in \mathrm{step}_{\mathcal{R}}(\sigma_s), \beta_t = \mathrm{mapvar}(\beta_s, m)\}$ with mapvar as defined in the previous section.

This implementation gives us a transition system $\mathcal{TS}_M = (\mathcal{S}, \iota, T, \mathcal{A}, \mathcal{B}, \mathcal{E})$ with initial state $\iota \in \{(\sigma, \beta_\iota[\texttt{self}/\mathrm{root}(\sigma)]) | \sigma \in \mathcal{G}\}, \beta_\iota = \emptyset, T = \{(\sigma_s, m, \sigma_t) | \sigma_s \overset{r,m}{\Rightarrow} \sigma_t, r \in \mathcal{R}, \sigma_s, \sigma_t \in \mathcal{S}\}$, $\mathcal{A}$ being the set $\mathcal{M}$ of partial state mappings, $\mathcal{E}$ being the transitive closure of the **succ** function applied to the initial environment ι, and $\mathcal{S}$ and $\mathcal{B}$ being all states and variable assignments occurring in an environment.

The algorithm for *evaluating* cOCL expressions of the form $(\mathsf{A}|\mathsf{E})\,\phi\,(\mathsf{U}|\mathsf{W})\,\psi$ is shown in Figure 5. To ease the presentation we drop intermediate checks allowing the algorithm to abort early in some cases, i.e. if a cycle was found during the evaluation or an element is added to a set required to be empty if the property holds. The algorithm proceeds as follows. First, it constructs the sets Φ and Ψ that contain all states where φ or ψ hold, respectively, and a third set η that contains all states reachable from a φ-state but where neither φ nor ψ hold. The worklist ω contains all nodes that need to be processed. The algorithm sets the worklist to the initial environment τ_ι and uses the **succ** function to iteratively expand the set of reachable environments. It evaluates φ and ψ in each environment τ and assigns τ to the corresponding sets Φ and Ψ, or to η if neither φ or ψ hold. Once every reachable environment is assigned to either Φ, Ψ, or η, the algorithm constructs the set Δ, which contains all environments from Φ that do not lie on an infinite path that does not leave Φ. That is, all environments in Φ that are part of a circular path are not in Δ. Finally, the algorithm builds the set Z that contains all deadlocked environment in Φ, i.e., environments that have no successor. Then, $I[\![\mathsf{A}\,\phi\,\mathsf{U}\psi]\!](\tau)$ holds if η is empty, and Φ contains neither cycle nor deadlock; $I[\![\mathsf{E}\,\phi\,\mathsf{U}\psi]\!](\tau)$ holds if Ψ is not empty; $I[\![\mathsf{A}\,\phi\,\mathsf{W}\psi]\!](\tau)$ holds if η is empty; and $I[\![\mathsf{E}\,\phi\,\mathsf{W}\psi]\!](\tau)$ holds if Ψ is not empty or Φ contains a cycle. Expressions $(\mathsf{A}|\mathsf{E})\,\mathsf{X}\phi$

[8] $P(X)$ is the set of all finite subsets of X.

```
/*Evaluates the given cOCL expression.
τι: start environment; POp: Path operator, Always or Exists;
TOp: Temporal operator, Until or Unless; returns: true iff the expression holds*/
function evaluate(τι, (POp φ TOp ψ)) : Bool
1  ω := {τι}; /*worklist                          */
2  Φ := ∅;    /*fulfilling φ, but not ψ          */
3  Ψ := ∅;    /*fulfilling ψ                     */
4  η := ∅;    /*fulfilling neither φ nor ψ       */
5  while ω ≠ ∅
6     pick τ = (σ, β) ∈ ω;
7     ω := ω \ {τ};
8     if I⟦φ⟧(τ) or I⟦ψ⟧(τ) then
9        if I⟦ψ⟧(τ) then
10          Ψ := Ψ ∪ {τ};
11       else
12          Φ := Φ ∪ {τ};
13          ω := ω ∪ succ(τ) \ (Φ ∪ Ψ ∪ η);
14       end if
15    else
16       η := η ∪ {τ}
17    end if
18 end while
19 Δ := ∅;
20 Δl := ∅;
21 repeat
22    Δl := Δ;
23    Δ := {τ ∈ Φ | succ(τ) ∩ (Φ \ Δl) = ∅};
24 until Δ = Δl
25 Z := {τ ∈ Φ | succ(τ) = ∅};
26
27 switch (POp, TOp)
28  case (Always, Until):
29    return Φ = Δ and Z = ∅ and η = ∅;
30  case (Always, Unless):
31    return η = ∅
32  case (Exists, Until):
33    return Φ ≠ ∅;
34  case (Exists, Unless):
35    return Φ ≠ ∅ or Z ≠ ∅ or Φ ≠ Δ;
36 end switch
```

Fig. 5. Until/Unless Algorithm Pseudo Code.

are implemented as $I[\![(\mathsf{A}|\mathsf{E})\,\mathsf{X}\phi]\!]((\sigma,\beta)) := (\forall|\exists)n \in \mathrm{succ}(\sigma,\beta) : I[\![\phi]\!](n) = \mathrm{true}$, where we check if all (at least one) successor of the current state satisfies φ.

The evaluation of a cOCL expression yields a *report* that, besides returning the result of the evaluation, contains a *cause* or explanation for the result. A cause is associated with a cOCL expression. It stores the result of the evaluation of the associated expression and, for each relevant sub-expression, a sub-cause. A sub-expression is *relevant* if it influences the result of its super-expression. For example, if the sub-expression φ in φ **or** ψ evaluates to **true** then no sub-cause is generated for ψ as the evaluation of φ uniquely determines the result of φ **or** ψ. If, however, both φ and ψ evaluate to **false**, then a sub-cause for each of the two sub-expressions is generated and stored in the cause of φ **or** ψ. Note that the cause generation is not necessarily deterministic, as is the case, for example, if both φ and ψ evaluate to **true** in φ **or** ψ.

4.2 Frontend

The MocOCL implementation, which is based on the Eclipse OCL project,[9] works in two phases, (i) step-wise exploration of the state space and evaluation of the provided cOCL expression on the thus far generated state space and

[9] `http://www.eclipse.org/projects/project_summary.php?projectid=modeling.mdt.ocl`

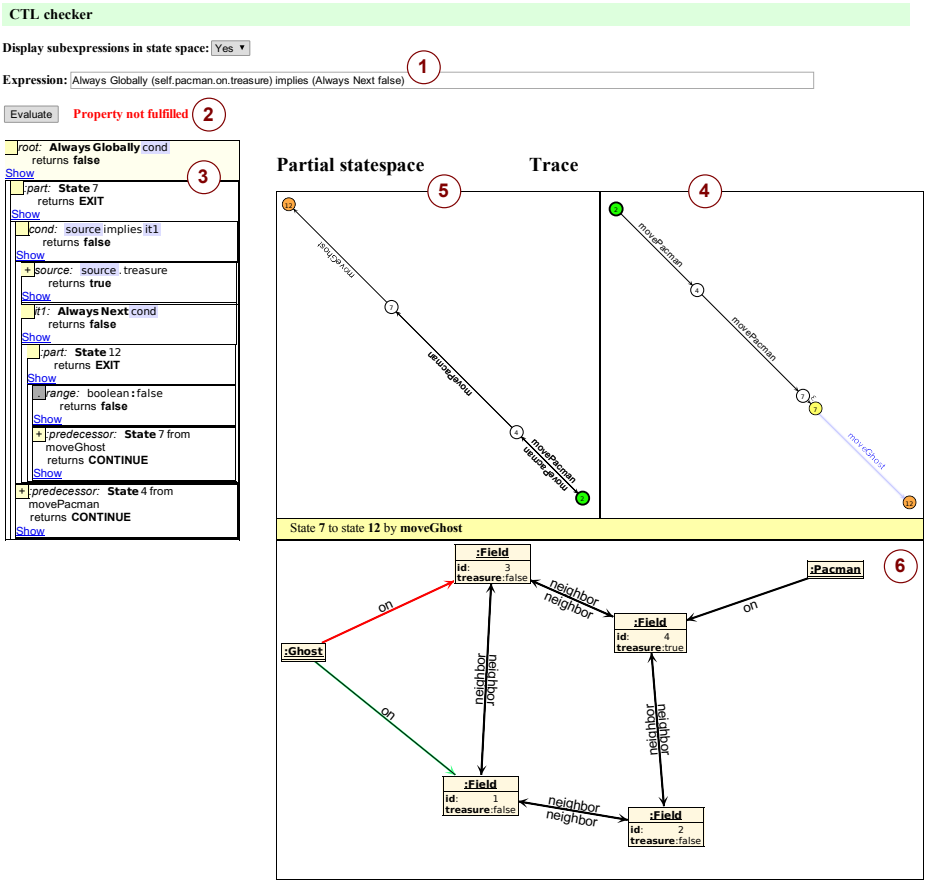


Fig. 6. Visualization of a cause in the MocOCL tool

(ii) visualization and report generation that provide useful information for the modeler on the reason of a specific result. The realization of the first phase is discussed above; in the following, we present the user interface and the report generation of our tool.

Figure 6 depicts a screenshot of MocOCL that displays the verification result for the initial 2×2 board (Fig. 1), the graph transformation rules *Move Pacman* and *Move Ghost* (Fig. 2), and the cOCL expression `Always Globally (self.pacman.on.treasure) implies (Always Next false)`. This cOCL expression states that whenever Pacman finds the treasure, no further states can be reached, i.e., the game ends. The MocOCL user interface consists of the following parts: (1) an input field for the cOCL specification, (2) the result of the verification, i.e., whether the cOCL specification is satisfied or not, (3) the cause that textually describes (4) the trace of the evaluation, which is embedded in (5) the partial state space. Further, upon clicking on a state or transition from

(3) the cause, (4) the trace, or (5) the partial state space, the selected state or transition is visualized in (6) the object diagram pane. The changes caused by a transition are highlighted in red and green indicating the deletion and creation of an association, respectively.

In the example displayed in Figure 6, the specification is not satisfied, i.e., the game does not end if Pacman finds a treasure. The cause shows a scenario where Pacman finds the treasure in two moves starting from the initial state (state 2) and moving first to state 4 and then to state 7. However, there is a transition `moveGhost` leading from state 7 to state 12. This transition is selected in (4) the trace and is highlighted in blue. The changes associated with the transition are displayed in (6) the object diagram pane. The deletion and creation of the `on` relation between the ghost and two adjacent fields describes the ghost's move. Consequently, the ghost may perform moves after Pacman already resides on the treasure field. Thus, the implementation does not satisfy the specification of the game and needs to be fixed by introducing an additional *Negative Application Condition* for the *Move Ghost* rule such that a ghost may no longer move once Pacman found the treasure.

A demo version of MocOCL is available as a browser version at

`http://www.modelevolution.org/mococl/`

and can be used without any installation efforts. In the demo version the initial model is fixed to the 2×2 board shown in Fig. 1 due to memory limitations on the server. A browser-based version for custom installations, which is not restricted to the Pacman model, is available for download at

`http://www.modelevolution.org/prototypes/mococl`.

5 A First Experimental Case Study

We performed an evaluation of cOCL's and MocOCL's *usability* and *performance*. In both cases we used the Pacman game described above because (i) its game play is simple and (ii) its complexity can be increased easily by raising the number of fields on the game board or the number of ghosts.

5.1 Usability

Experimental Setup. Concerning the evaluation of the usability of our verification framework, we are interested in (i) the intuitiveness of the cOCL language, i.e., the combination of OCL expressions and temporal operators, and (ii) the usability of MocOCL's user interface, most notably the presentation of the cause. Thus, we conducted a series of qualitative, semi-structured interviews with 11 researchers with expertise in MBE or in formal verification, and some in both. Each test person was interviewed separately for one to two hours. The interviews were structured as follows.

The interview started with an introduction to model checking in the context of MBE. The Pacman game discussed in Section 2 served as the running example.

Table 2. Evaluation results based on self–estimated proficiency

Prior Knowledge	**Exercises**			**Subjective Evaluation**		
	Low	**Medium**	**High**	**Low**	**Medium**	**High**
Structural Models	12	8	10.5	8	7.5	7
Behavioral Models	8	10.1	11	7	7.7	6
OCL	12	9.6	11.5	8	6.9	8
Graph transform.	10	9.4	11.7	7.3	6.8	7.7
Standard Logics	9.5	10.3	10.4	5	7.4	8
Temporal Logics	10.4	9.9	—	6.8	8.3	—
Model checkers	10	–	10.4	6.5	–	8.0

Depending on the expertise of the test persons, background on either structural and behavioral modeling or model checking was given to ensure a common level of understanding. Next, the cOCL language was presented with several examples similar to those in Table 1. Then, the test persons had to solve exercises and were encouraged to use MocOCL's web interface to find the solutions. These exercises were grouped into three blocks, each block raising the level of difficulty gently. First, the test persons were required to match a set of cOCL expressions to their corresponding natural language explanations. Next, the test persons were asked to explain the meaning of several cOCL expressions in natural language. Finally, the test persons had to formulate cOCL expressions on their own. The last question of the exercises the test person to assert required whether the game is over after Pacman finds the treasure. The task setup was identical to the scenario depicted in Figure 6. In the final part of the interview the test persons were asked to provide feedback on whether it was "Easy", "Medium", "Hard", or "Infeasible" to (i) read cOCL expressions, (ii) write cOCL expressions, and (iii) use MocOCL's interface.

The questionnaire used during the interview, including the exercises and the subjective evaluation, is shown in the extended technical report.

Results. All participants successfully revealed the defect in the graph transformation rule *Move Ghost* (Fig. 2) with the help of MocOCL. Even in this small example, however, only few of the test persons were able to detect the defect without the tool. Thus, we may conclude that MocOCL is supportive when model checking is performed in the context of MBE. The interviews also showed that some background on CTL is indispensable to apply the temporal operators and path quantifiers correctly. While most participants reported that reading the cOCL expressions is intuitive, test persons without any prior exposition to formal verification and model checking in particular expressed difficulties phrasing such expressions on their own. In particular, the existential path quantifier which wo originally called "`sometimes`" caused confusion among the test persons and many suggested to use the more intuitive term "exists". To avoid the name clash and ambiguities with OCL's `exists` operator we revised cOCL's concrete syntax such that (i) all cOCL keywords are capitalized and (ii) the keyword `Exists` was introduced as an additional existential path quantifier. Further feedback resulted

in slight visualization improvements; in particular, we now color start and end nodes of the evaluation trace.

Table 2 summarizes the overall evaluation results. Initially each participant was asked to provide a self-assessment of his/her expertise in various domains that we considered relevant for using MocOCL. Each participant was then assigned to his/her matching expertise group ("Low", "Medium" or "High") in each domain.

The table contains the average number of points given by persons of a specific expertise group in a certain domain. In total, a person could score a maximum number of twelve points in the exercise part and award up to nine points during the subjective evaluation. Each task was awarded either zero points for a wrong or missing answer, one point for a partially correct answer, i.e. the use of $\rightarrow$ instead of the OCL `implies`, and two points for a completely correct answer. The first block, was considered an single task while the three cOCL expressions which had to be interpreted and the two cOCL expressions which had to be written were considered as individual tasks each. A test person that solves the matching task and provide the correct meaning of two cOCL expressions and only a single, partially correct, solution for writing a cOCL expression scores seven points.

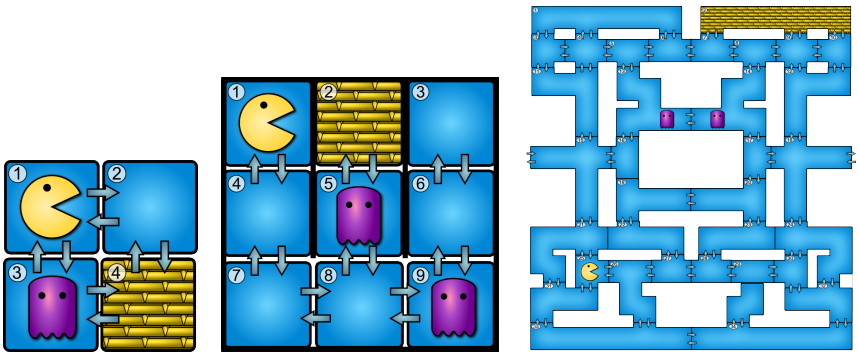
For the subjective evaluation, each person had to decide how hard "Reading cOCL", "Writing cOCL" and "Tool use" were. The answer "Easy" yielded three points, "Medium" two points, "Hard" one point and "Infeasible" zero points. The total value is the sum of values for answers for the individual domains. A test person that experienced *reading cOCL* was easy, *writing cOCL* was hard, and *using the tool* was medium awards six out of nine points.

Discussion. The evaluation provided valuable insights on the usability of our tool. However, to obtain statistically relevant results we have to increase the number and the diversification of our test persons. We plan to contribute such an extensive user study in the course of our Model Engineering class, a master course offered during the winter term providing a test-bed of up to 100 students.

Overall, we could observe a trend that the knowledge of behavioral models and logics increases the odds of successfully applying MocOCL to verification tasks, while expertise in graph transformations, OCL, and standard modeling does not. In contrast, persons knowing model checking and logics, but not knowing graph transformations gave lower ratings in the subjective evaluation.

We concluded that MocOCL should provide other facilities to specify dynamic behavior, for example, state machines or a subset of the Java programming language. In future evaluations, we will also have to consider direct comparisons to other tools like Groove.

Additional feedback that we received is hard to capture by facts in tables. This includes the way some people were interested in using the tool by playing around with various features. This encouraging observation seems to confirm the chosen approach of how to realize MocOCL. In contrast, the language itself seems to be too hard for immediate use since no one tried out custom expressions beyond those required for the tasks. Finally, even though the interviews were scheduled for a duration of up to two hours, we felt that the time required for an in–depth evaluation with a single person should be even higher. As this seems



(a) Small board. (b) Medium board. (c) Large board.

Fig. 7. Different configurations of the Pacman game used for the evaluation.

to expect too much from a volunteering test person we plan to restructure the exercise part such that the tasks can be solved before the actual interview.

5.2 Performance

Experimental Setup. In order to asses the performance of our implementation, we measured runtimes required for different board sizes and different numbers of ghosts. Along these parameters we are able to scale the size of the state space and observe the behavior of our tool with increasing state space sizes. An upper bound for the state space size is $n^{(g+1)}$ with n being the number of fields and g being the number of ghosts. The initial configurations of the used boards are shown in Fig. 7. We ran our performance tests with three different configurations, (i) a 2×2 board with one ghost (Fig. 7(a)), (ii) a 3×3 board with two ghosts (Fig. 7(b)), and (iii) a maze of 34 fields with zero, one, and two ghosts (Fig. 7(c)).

On each game configuration, we evaluated the following three queries:

- `Always Globally true`
- `Exists Eventually pacman.on.treasure`
- `Exists Eventually pacman.on.treasure and ghosts->forAll(g | g.on <> pacman.on)`

Although the first expression is trivially true, MocOCL traverses the entire state space to assert its correctness because it does not implement any simplification rules for the input query yet. Thus, we use this first expression to analyze MocOCL's runtime behavior when traversing state spaces of different sizes. The second expression queries whether Pacman eventually finds a treasure. The last query contains a more complicated OCL sub-expression in order to validate if Pacman can always win the game. The experiments were performed on an Intel i5-2410M Machine with 2.30 GHz and 8 GB RAM.

Table 3. Runtimes of MocOCL (times are given in ms).

	Field	Gh.	St.	gentime		evaltime		total	
				avg	std	avg	std	avg	std
State space generation	small	1	16	25	6.1	20	5.9	46	7
	medium	2	405	1051	623.3	114	42.6	1165	657.6
	large	0	34	128	63.8	20	5.1	148	68.9
	large	1	1156	7712	381.4	258	68.6	7970	437.4
	large	2	20230	213k	16.3k	5164	432.6	218k	16.4k
Pacman on treasure	small	1	10	19	21.3	29	2.4	48	22.4
	medium	2	120	124	18.3	63	19.9	188	36.2
	large	0	34	85	9.9	28	0.4	113	10
	large	1	631	1932	57.8	114	28.9	2046	38.9
	large	2	6920	30685	167.9	1819	34.5	32504	187.3
Pacman wins	small	1	10	15	19.7	65	9.7	80	19
	medium	2	176	128	115.4	266	94.8	393	128.3
	large	0	34	88	18.1	45	7.4	133	18.9
	large	1	631	2095	223.5	316	66.3	2411	224.6
	large	2	6920	22878	557.8	10772	16.2	33650	566.1

Results. The runtimes of our experiments are summarized in Table 3. The first query is called *state space generation*, the second query is called *Pacman on treasure*, and the third query is called *Pacman wins*. The column *Gh.* contains the number of ghosts and the column *St.* contains the number of generated states. Further, the table shows the overall runtime of our tool (column *total*), which we split into the time necessary to generate the state space (column *gentime*) and the time required to evaluate the cOCL expressions (column *evaltime*) by caching the state space. We repeated each run five times and report the average runtime as well as the standard deviation. Overall all queries could be answered within less than five minutes. But if we add a third ghost to the large field, the 8 GB of memory are insufficient to answer the given queries.

Discussion. In its current state, we observe that our tool is not competitive in terms of performance, even without a direct comparison to other tools. For the moment, however, we clearly focus on the tight integration of OCL and model checking-based verification, not so much on the performance. This is directly reflected in the performance results of the current implementation shown in Table 3, which we discuss in the following. We observe a high standard deviation for all expressions when run on the more complicated 3×3–field. We suspect this to be due to the various online JVM optimizations. These optimizations are also likely the cause for the generation time of the "Pacman wins"–expression being significantly lower then the generation time of the "Pacman on treasure"–expression even though the same number of states are generated. The excessive increase in evaluation time for the "Pacman wins"–expression for more ghosts originates from the `forAll`–expression covering a different number of ghosts. In the case of no ghosts, the expression just needs to ensure that there are no ghosts

in each state which is fast. In the case of one or more ghosts, the expression has to check that the position of each ghost is different to the position of Pacman.

Our approach scales approximately as well as comparable solutions like GROOVE. Our benchmarks show that our implementation spends significant amounts of time on both the state space generation and the evaluation of the cOCL expression; thus, it is sensible to look into improvements in both areas. A more efficient cOCL evaluation might also reduce the state space generation time if fewer states need to be generated.

6 Related Work

We discuss related works focusing on temporal extensions for OCL first, followed by reviews of model checkers that verify whether a system, whose structure and behavior is described by (graphical) models, satisfies its specification. For an in-depth discussion on verification approaches in the context of MBE we refer the interested reader to [13].

Temporal Extensions. Distefano *et al.* [8] propose a CTL-based logic, called BOTL, to specify static and dynamic properties of object-oriented systems. But instead of extending OCL, they map OCL onto BOTL; hence, they provide formal semantics for a large part of OCL based on BOTL. Ziemann *et al.* [29] suggest an extension based on linear time logic, which is similar in nature to our CTL-based solution. Soden and Eichler [26] also present a linear time-based extension for OCL and define the operational semantics of MOF-conforming models with the Model Execution Framework for Eclipse (MXF) [27]. This allows them to describe a finite execution trace through a sequence of changes. Flake and Mueller [12] use state charts to describe the behavior of associated class diagrams and time-based traces to capture the execution of the system. They propose a UML Profile to specify state-oriented, real-time invariants, whose semantics are defined by a mapping to clocked CTL formulas. Bradfield *et al.* [5] embed OCL into the observational μ-calculus. They suggest the use of predefined templates with intuitive semantics, from which the underlying μ-calculus formula is automatically generated. Likwise, Kanso and Taha [16] introduce a temporal extension based on Dwyer *et al.*'s patterns for the specification of properties for finite state systems [10]. They define a scenario-based semantics for their extension, where each scenario is a finite sequence of events.

Verification Engines. Mullins and Oarga [22] present EOCL, an extension inspired by BOTL, that augments OCL with CTL operators. The operational semantics of EOCL are defined over object-oriented transition systems. They announce and describe SOCLe, a tool that translates class, state chart, and object diagrams into an abstract state machine and checks on-the-fly if the system satisfies a given EOCL specification.The GROOVE framework [17] verifies object-oriented systems modeled as attributed, type graphs with inheritance relations. It is similar to MocOCL in that it represents system states as graphs and the system's

behavior by graph transformations. But, in contrast, it uses standard CTL and LTL to formulate the system's specification. Recently, abstraction techniques have been implemented to handle infinite state spaces by over-approximating system behaviors [23]. Al-Lail *et al.* [20] describe systems with class diagrams and the operations' contracts, given by OCL pre- and postconditions, capture the behavior of the system. They use TOCL [29] to specify reachability and safety properties. Their model checker builds a *Snapshot Transition Model* that consists of snapshots, which represent a state of the system, and transitions, which run from source states that satisfy an operation's precondition to target states that satisfy the postcondition. With the USE Model Validator [28] they perform a depth-bounded search for sequences of snapshots that violate the specification and, if one is found, visualizes the violating sequence as a UML sequence diagram. Dingel et al. [21,30] verify UML–RT state machines symbolically using a CTL–extension without transforming to another model checker, but representing their models as Functional Finite State Machines. In contrast to MocOCL, OCL is not part of their language.

To the best of our knowledge, MocOCL is currently the only framework that (i) integrates its CTL-extension seamlessly into the formal semantics of OCL, (ii) implements the evaluation of CTL operators directly within the OCL evaluation engine, and thus (iii) performs the verification and result reporting directly at the modeling layer.

7 Conclusion and Future Work

In this paper, we present syntax and semantics of cOCL, our OCL extension with CTL-based temporal operators. Further, we describe the implementation and technical feasibility of our MocOCL model checker that verifies cOCL specifications of software systems, whose static structure is described by Ecore-conformant models and whose behavior is defined by a set of graph transformations. We conducted a first user study, where we invited colleagues to solve a set of verification tasks with our tool. The results of this user study are already incorporated into MocOCL and they improved, among others, the concrete syntax of cOCL.

A performance evaluation shows that our approach is able to verify models of various sizes. With increasing state space sizes, memory consumption becomes a major issue. This is, however, an inherent problem of model checking in general, which suffers from the state explosion problem and, for practical application, several tuning techniques can be applied. In our current prototype, we do not use such techniques yet. Thus, in future work, we plan to employ symbolic model checking and abstraction techniques to improve runtimes and memory consumption.

Besides technical issues we are also interested in improving the usability of our tool. The aim is (i) to further explore the intuitiveness of the combination of temporal operators and OCL expressions and (ii) the presentation of the evaluation result, in particular, with respect to the reconstructability of the cause. A larger user study is planned to improve future versions of the tool.

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. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
3. 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)
4. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: Model Checking of CTL-Extended OCL Specifications. Tech. Rep. BIG-TR-2014-2, E188 - Institut für Softwaretechnik und Interaktive Systeme; Technische Universität Wien (2014)
5. Bradfield, J.C., Filipe, J.K., Stevens, P.: Enriching OCL Using Observational Mu-Calculus. In: Fundamental Approaches to Software Engineering. LNCS, vol. 2306, pp. 203–217. Springer (2002)
6. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logics of Programs. LNCS, vol. 131, pp. 52–71. Springer (1981)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)
8. Distefano, D., Katoen, J.P., Rensink, A.: On a Temporal Logic for Object-Based Systems. In: Formal Methods for Open Object-Based Distributed Systems IV. IFIP Advances in Information and Communication Technology, vol. 49, pp. 305–325. Springer US (2000)
9. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering. pp. 411–420. ACM (1999)
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
12. Flake, S., Müller, W.: Formal semantics of static and temporal state-oriented OCL constraints. Software and System Modeling 2(3), 164–186 (2003)
13. 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)
14. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)
15. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T.L., Porres, I.: Model Checking Dynamic and Hierarchical UML State Machines. In: Models in Software Engineering. LNCS, vol. 4364, p. 15. Springer (2006)
16. Kanso, B., Taha, S.: Temporal Constraint Support for OCL. In: Software Language Engineering. LNCS, vol. 7745, pp. 83–103. Springer (2012)
17. Kastenberg, H., Rensink, A.: Model Checking Dynamic States in GROOVE. In: Model Checking Software. LNCS, vol. 3925, pp. 299–305. Springer (2006)
18. Kern, C., Greenstreet, M.R.: Formal Verification in Hardware Design: A Survey. ACM Transactions on Design Automation of Electronic Systems (TODAES) 4(2), 123–193 (Apr 1999)

19. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. In: Models in Software Engineering. LNCS, vol. 4364, pp. 42–51. Springer (2006)
20. 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)
21. Moffett, Y., Dingel, J., Beaulieu, A.: Verifying Protocol Conformance Using Software Model Checking for the Model-Driven Development of Embedded Systems. IEEE Software Engineering 39(9), 1307–13256 (2013)
22. 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)
23. Rensink, A., Zambon, E.: Neighbourhood Abstraction in GROOVE. ECEASST 32, 44–56 (2010)
24. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In: Object Modeling with the OCL, LNCS, vol. 2263, pp. 42–68. Springer (2002)
25. Selic, B.: What will it take? A view on adoption of model-based methods in practice. Software and Systems Modeling 11, 513–526 (2012)
26. Soden, M., Eichler, H.: Temporal Extensions of OCL Revisited. In: Model Driven Architecture - Foundations and Applications. LNCS, vol. 5562, pp. 190–205. Springer (2009)
27. Soden, M., Eichler, H.: Towards a model execution framework for Eclipse. In: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture. pp. 4:1–4:7. ACM (2009)
28. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design, Automation and Test in Europe. pp. 1341–1344. IEEE (2010)
29. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Perspectives of Systems Informatics. LNCS, vol. 2890, pp. 351–357. Springer (2003)
30. Zurowska, K., Dingel, J.: Model Checking of UML-RT Models Using Lazy Composition. In: Model-Driven Engineering Languages and Systems, LNCS, vol. 8107, pp. 304–319. Springer (2013)