OCL meets CTL: Towards CTL-Extended OCL Model Checking*

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

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

Abstract. In software modeling, the Object Constraint Language (OCL) is an important tool to specify properties that a model has to satisfy. The design of OCL reflects the structure of MOF-based modeling languages like UML and the tight integration results in an intuitive usability. However, OCL allows to express properties in the context of the current state of an instance model only but not with respect to its evolution.
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 explain syntax and semantics of our OCL extension and provide a prototypical implementation of our MocOCL model checker.

1 Introduction

In software and hardware verification, model checking [3] 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 is often expressed in terms of a temporal logic formula. Common choices are the computation tree logic (CTL) and the linear temporal logic (LTL) that are used to express constraints over the lifetime of the system.

In the context of model-based engineering (MBE), software models are the core artifacts to specify and develop a system. Obviously, the correctness of the models is a prerequisite for the correctness of the system that is presented to the end user [19]. Consequently, formal verification techniques find their way into the MBE processes to help detect and avoid errors in the models. Due to its fully automatic verification capabilities, model checking has been shown to be of particular importance. Recent works and tools, for example, Hugo [14], Groove [13], and Proco [11] to name but a few, show that various kinds of software models can be suitably verified with model checking.

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

[3] Usually, a specification consists of a set of properties that should hold in a system. We will, however, often use these terms interchangeably.

Many approaches, however, require the modeler to express the properties of a specification in the language of the target model checker. In our opinion this is a drawback as (1) the modeler needs to leave her usual working environment and (2) the properties are not expressed on the modeling layer. Typically, a modeling environment provides some language to express constraints that a model has to satisfy. For example, the Object Constraint Language (OCL) [10] is a widely adopted language to express invariants, and pre- and postconditions over a model. But OCL only considers a single snapshot of the model, not its evolution during the execution of the system.

In this paper we thus present a CTL-based temporal extension to OCL and, in addition, provide a working implementation of a model checker to verify CTL-extended OCL constraints. Hence, our contribution is twofold. First, we extend syntax and semantics of OCL with CTL operators. Second, we integrate the syntactical extension into the Eclipse OCL Workbench and implement a model checker to evaluate CTL-extended OCL constraints in Eclipse. In the following, we assume that the static structure of the system is represented by an Ecore model and the system's behavior is described by a set of model transformations. The specification is expressed as a CTL-extended OCL constraint.

The structure of this paper is as follows. In Section 2 we present the syntax and semantics of our CTL-based OCL extension. Then, in Section 3, we discuss the overall idea by means of a motivating example and describe the implementation of our verification framework. Finally, after showcasing a first case study (Section 4), we close the paper with an overview of related approaches (Section 5), and conclude in Section 6 with a critical discussion and an outlook on future work. Due to space limitations we assume familiarity with model checking and CTL and kindly refer to standard literature [3] for an introduction.

2 A Temporal Extension of OCL

In the following, we formally introduce the syntax and semantics of OCL enriched with standard CTL operators. We integrate our extension, named cOCL, into the formal semantics of OCL [10,18] *without* modifying the existing definitions. Due to space limitations we do not reproduce the existing definitions here and kindly refer to the work of Richters *et al.* [18] for the details on the syntax and semantics of OCL.

Definition 1 (Syntax). *The expressions of cOCL are defined as follows:*

1. *Each OCL expression of Definition 1 in [18] is in cOCL;*
2. *if* $\phi, \psi \in Expr_{Bool}$ *then* $\mathsf{AX}\phi$, $\mathsf{EX}\phi$, $\mathsf{A}\phi\mathsf{W}\psi$, $\mathsf{E}\phi\mathsf{W}\psi$, $\mathsf{A}\phi\mathsf{U}\psi$, $\mathsf{E}\phi\mathsf{U}\psi \in Expr_{Bool}$ *in cOCL, where* $Expr_{Bool}$ *are expressions of type Boolean.*

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 the *eventually* and *globally* operators as equivalences: $\mathsf{EF}\varphi \equiv \mathsf{E}\,true\,\mathsf{U}\,\varphi$ and $\mathsf{AF}\varphi \equiv \mathsf{A}\,true\,\mathsf{U}\,\varphi$, and $\mathsf{EG}\varphi \equiv \mathsf{E}\,\varphi\,\mathsf{W}\,false$ and $\mathsf{AG}\varphi \equiv \mathsf{A}\,\varphi\,\mathsf{W}\,false$.

Definition 2 (State space). *The state space $\mathcal{K}_M = (\mathcal{S}, \iota, \mathcal{T}, \mathcal{B}, \mathcal{E})$ of a model M consists of a set of states $\mathcal{S}$, a single initial state $\iota \in \mathcal{S}$, a transition relation $T \subseteq \mathcal{E} \times \mathcal{E}$, a set of variable assignments $\mathcal{B}$, and the environment relation $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{B}$. An environment $\tau \in \mathcal{E}$ is a pair (σ, β), where $\sigma \in \mathcal{S}$ is a state and $\beta \in \mathcal{B}$ a variable assignment.*

For each state $\sigma \in \mathcal{S}$ the set of objects, associations, and attributes of M are accessible via $\sigma|_{class}$, $\sigma|_{assoc}$, and $\sigma|_{attrs}$ [18]. A variable assignment is a function $\beta : Var_t \rightarrow Val_t$ that, given a variable name, returns the current value of the associated variable, where t is the type of the associated variable. The concept of an environment $\tau = (\sigma, \beta)$ has been introduced by Richters and Gogolla [18].

Definition 3 (Path). *Let $\mathcal{K}_M = (\mathcal{S}, \iota, \mathcal{T}, \mathcal{B}, \mathcal{E})$ be the state space of a model M. A* path *π is a finite or infinite sequence of environments $(\tau_1, \tau_2, \ldots)$ with $\tau_i \in \mathcal{E}$ such that $(\tau_i, \tau_{i+1}) \in \mathcal{T}$. For a path $\pi = (\tau_1, \tau_2, \ldots)$, we define the projection function $\pi(i) = \tau_i$. The length of a path $|\pi| = n$ for finite paths $\pi = (\tau_1, \ldots, \tau_n)$, and $|\pi| = \infty$ for infinite paths $\pi = (\tau_1, \tau_2, \ldots)$.*

We are now able to describe the semantics of cOCL as follows.

Definition 4 (Semantics). *Let $\mathcal{K}_M$ be a state space of model M. The semantics of a cOCL expression is defined by the rules i.–vi. of Definition 2 from [18] plus the following rules for the temporal extension.*

vii. $I[\![A\,\phi\,U\,\psi]\!](\tau) = \text{true} \Leftrightarrow \forall\,\text{paths } \pi \text{ with } \pi(0) = \tau : \exists n \in \mathbb{N}, n \leq |\pi| : I[\![\psi]\!](\pi(n)) = \text{true} \wedge \forall\, 0 \leq i < n : I[\![\phi]\!](\tau_i) = \text{true}$

viii. $I[\![E\,\phi\,U\,\psi]\!](\tau) = \text{true} \Leftrightarrow \exists\,\text{path } \pi \text{ with } \pi(0) = \tau : \exists n \in \mathbb{N}, n \leq |\pi| : I[\![\psi]\!](\pi(n)) = \text{true} \wedge \forall\, 0 \leq i < n : I[\![\phi]\!](\tau_i) = \text{true}$

ix. $I[\![A\,\phi\,W\,\psi]\!](\tau) = \text{true} \Leftrightarrow \forall\,\text{paths } \pi \text{ with } \pi(0) = \tau : \forall n \in \mathbb{N}, n \leq |\pi| : I[\![\phi]\!](\pi(n)) = \text{false} \rightarrow \exists i \in \mathbb{N}, i \leq n : I[\![\psi]\!](\pi(i)) = \text{true}$

x. $I[\![E\,\phi\,W\,\psi]\!](\tau) = \text{true} \Leftrightarrow \exists\,\text{path } \pi \text{ with } \pi(0) = \tau : \forall n \in \mathbb{N}, n \leq |\pi| : I[\![\phi]\!](\pi(n)) = \text{false} \rightarrow \exists i \in \mathbb{N}, i \leq n : I[\![\psi]\!](\pi(i)) = \text{true}$

xi. $I[\![E\,X\,\phi]\!](\tau) = \text{true} \Leftrightarrow \exists\,\text{path } \pi \text{ with } \pi(0) = \tau, |\pi| \geq 1 : I[\![\phi]\!](\pi(1)) = \text{true}$

xii. $I[\![A\,X\,\phi]\!](\tau) = \text{true} \Leftrightarrow \forall\,\text{paths } \pi \text{ with } \pi(0) = \tau, |\pi| \geq 1 : I[\![\phi]\!](\pi(1)) = \text{true}$

The semantics of the *eventually* and *globally* operators follow directly from the above definitions. We define a cOCL expression *satisfiable* as follows.

Definition 5 (Satisfiability). *A cOCL expression ϕ is* satisfiable *w.r.t. a state space $\mathcal{K}_M$ iff $I[\![\phi]\!](\iota)$ is* true *w.r.t. $\mathcal{K}_M$.*

3 A framework to integrate CTL and OCL

In the following we describe our verification framework that accepts MOF-based software models and cOCL specifications as input. This allows us to embed model checking support seamlessly into the MBE workflow. We introduce the general idea based on the (in)famous dining philosophers problem.

3.1 Basic Idea

Consider the model depicted in Fig. 1. The root node of type Table contains an arbitrary number of instances of type Philosopher, Plate, and Fork. Each philosopher is associated with exactly one plate. Each fork is assigned to two adjacent plates such that philosophers need to share their forks.

In order to eat, the philosophers need to pick up both the left and the right fork of their plate. When they are done eating, they release their forks and put them back on the table. We model the dynamic behavior of the *dining philosophers system* with graph transformations [7], but any other model transformation formalism would work equally well. Figure 2 shows the rules in storyboard notation. Rule (a) of Figure 2 creates an association to assign the left fork to a philosopher whenever the philosopher has not picked up the left fork yet (forbid#1) and the philosopher to his right has not picked up the fork either (forbid#2). The rule to pick up the right fork works analogously. Rule (b) in Fig. 2 releases simultaneously the forks that a philosopher has picked up and deletes each of the associations between the philosopher and the left and right forks.

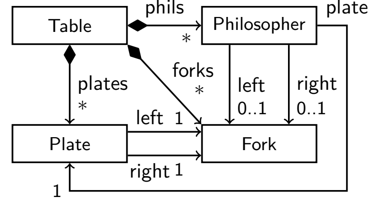
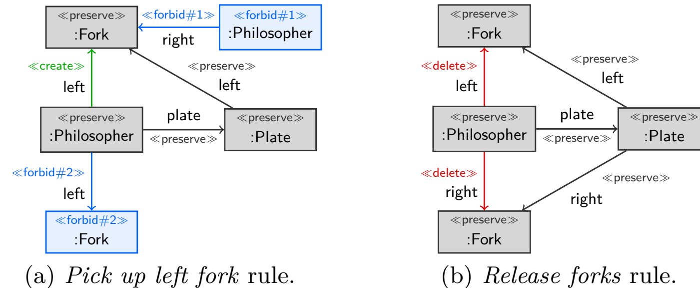


Fig. 1. Dining philosophers model

To check a cOCL property, we first need to create the state space of the dining philosophers system. The state space is obtained by recursively applying all matching graph transformation rules starting on a given initial model. For example, consider an initial model with one table, three philosophers, three plates, and three forks, where none of the philosophers has neither picked up a left fork nor a right fork. The initial state has six successor states, as both rules for picking up a left fork and a right fork can be applied to each philosopher. Overall, the resulting state space consists of 27 states.

3.2 Implementation

Our verification framework consists of two parts, a concrete syntax extension for our CTL-based OCL extension that we presented in Section 2, and the MocOCL model checker that verifies cOCL specifications.



(a) *Pick up left fork* rule. (b) *Release forks* rule.

Fig. 2. Graph transformation rules for the dining philosophers example.

The *concrete syntax* enhances the readability of cOCL expressions. It allows 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 quantifiers for temporal operators become **always** and **sometimes**.

The type definitions $Sequence(t)$, $Set(t)$, and $Bag(t)$ and the function definitions mkSequence_t, mkSet_t, and mkBag_t that we use in the following definitions are those introduced by Richters and Gogolla [18]. For example, $I(Sequence(t))$ defines the set of all possible sequences of type t. We define $I(Collection(t)) = (I(Sequence(t)) \cup I(Set(t)) \cup I(Bag(t)))$. The prototypical implementation of the *MocOCL model checker* performs the following steps. Given an initial environment, a set of model transformations, and a cOCL specification, MocOCL first generates the state space, then evaluates the cOCL properties, and finally reports to the modeler useful information on the reason of a specific evaluation 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 $\mathbf{stategen}_{\mathcal{R}}\colon \mathcal{S} \to P(\mathcal{S} \times \mathcal{M})$[4] handles the generation of the state space. 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 $m : \sigma_{Class} \to \sigma_{Class} \cup \{\bot\}$ is a morphism 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 $\sigma_s \overset{r}{\Rightarrow} \sigma_t$ to denote that σ_s is rewritten to σ_t by rule $r \in \mathcal{R}$. The state space generation function is then defined as $\mathbf{stategen}_{\mathcal{R}}(\sigma_s) = \bigcup_{r\in\mathcal{R}}\{(\sigma_t, m) | \sigma_s \overset{r}{\Rightarrow} \sigma_t \wedge \exists m \in \mathcal{M} : \forall c \in \sigma_s|_{class} : m(c) \in \sigma_t|_{class} \vee m(c) = \bot\}$. 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 $\mathbf{succ}((\sigma_s, \beta_s)) := \{(\sigma_t, \beta_t) | (\sigma_t, m) \in \text{stategen}_{\mathcal{R}}(\sigma_s), \beta_t = \text{mapvar}(\beta_s, m)\}$. The $\mathbf{mapvar} : \mathcal{B} \times \mathcal{M} \to \mathcal{B}$ function takes a variable assignment β_s of state σ_s and a mapping $m \in \mathcal{M}$ and updates β_s with respect to m resulting in a variable assignment β_t for the successor state σ_t. It is defined by

$$\text{mapvar}(\beta(v), m) : v \mapsto \begin{cases} \text{mapcol}(\beta(v), m) & \text{if } \exists t : \beta(v) \in I(Collection(t)) \\ m(\beta(v)) & \text{if } \beta(v) \in Dom(m), \text{i.e., } \beta(v) \in \sigma_{class} \\ \beta(v) & \text{otherwise.} \end{cases}$$

A collection is mapped by the $\mathbf{mapcol} : Collection(t) \times \mathcal{M} \to Collection(t)$ function, which applies **mapvar** recursively to all elements of the collection:

$$\text{mapcol}(X, m) = \begin{cases} \text{mkSequence}_t(\text{mapvar}(x, m) | x \in X) & X \in Sequence(t) \\ \text{mkSet}_t(\text{mapvar}(x, m) | e \in X) & X \in Set(t) \\ \text{mkBag}_t(\text{mapvar}(x, m) | e \in X) & X \in Bag(t) \end{cases}$$

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

This implementation gives us a state space $\mathcal{K}_M = (\mathcal{S}, \iota, \mathcal{T}, \mathcal{B}, \mathcal{E})$ with initial state $\iota \in \mathcal{G}$ and $(\tau_s, \tau_t) \in \mathcal{T} \Leftrightarrow \tau_t \in \text{succ}(\tau_s)$, $\mathcal{E}$ being the transitive closure of applying the **succ** function to the initial environment (ι, β_ι), and $\mathcal{S}$ and $\mathcal{B}$ being all states and variable assignments occurring in an environment. Currently, we use HENSHIN's graph rewrite engine [1] to generate the state space.

The *evaluation* of cOCL expressions of the form $(A|E)\,\phi\,(U|W)\,\psi$ is shown in Fig. 3. The algorithm constructs the sets Φ and Ψ that contain all states where φ and ψ hold, 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. Then, $I[\![A\,\phi\,U\psi]\!](\tau)$ holds if η is empty, and Φ contains neither cycle (cycle $\Leftrightarrow$ $\Delta \neq \Phi$) nor deadlock; $I[\![E\,\phi\,U\psi]\!](\tau)$ holds if Ψ is not empty; $I[\![A\,\phi\,W\psi]\!](\tau)$ holds if η is empty; and $I[\![E\,\phi\,W\psi]\!](\tau)$ holds if Ψ is not empty or Φ contains a cycle. Expressions $(A|E)\,X\phi$ are implemented as $I[\![(A|E)\,X\phi]\!]((\sigma,\beta)) = (\forall|\exists) n \in \text{succ}(\sigma,\beta) : I[\![\phi]\!](n) = \text{true}$ where we check if all (at least one) successor of the current state satisfies expression φ.

```
ω = {τ_ι};  Φ = Ψ = η = Δ = Δ_l = ∅
while ω ≠ ∅
   pick τ = (σ, β) ∈ ω
   ω := ω \ {τ}
   if I⟦ψ⟧(τ) or I⟦φ⟧(τ) then
      if I⟦ψ⟧(τ) then
         Ψ := Ψ ∪ {τ}
      else
         Φ := Φ ∪ {τ}
         ω := ω ∪ succ(τ) \ (Φ ∪ Ψ ∪ η)
      end if
   else
      η := η ∪ {τ}
   end if
end while
repeat
   Δ_l := Δ
   Δ := {τ ∈ Φ | succ(τ) ∩ (Φ \ Δ_l) = ∅}
until Δ = Δ_l
```

Fig. 3. Algorithm pseudo code

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 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 cause generation need not be deterministic, as is the case, for example, if both φ and ψ evaluate to **true**. In case of cOCL expressions the report generation becomes expensive fast. For example, the number of generated sub-causes for a counter-example trace of a $\mathsf{EF}\varphi$ formula, where φ is a propositional formula without set operations, has as upper bound $\mathcal{O}(|\mathcal{K}_M| \times |\varphi|)$ the size of the state space times the size of the formula φ.

4 A First Showcase

In this section, we illustrate how cOCL expressions can be used to express properties of a system. For the purpose of comparison, we also specify the properties as CTL formulas as used in Groove [17], which generates the state space similar to our model checker, but uses graph transformations to express properties of the system. Such a graph transformation-based expression evaluates to true if the graph transformation can be applied in the current state. Graph transformations are the only link between the temporal formula to be verified and the model of the system. Thus, it is neither possible to store and compare variables nor to iterate over multiple state space elements.

In the following, we formulate two properties based on the dining philosophers example introduced in the previous section. The statement "every philosopher should always be able to eat at some point in the future" is specified as follows:

```
self.philosophers->forAll(p | always globally always eventually
(p.left <> null and p.right <> null))
```

Note that OCL allows us to quantify over all philosopher objects and with our extension we are able to say that in all reachable branches of the state space eventually a philosopher has to have a right and a left fork.

In Groove, every philosopher needs an ID to be traced. Additionally, there are no parametrized properties and thus a new graph transformation has to be specified for each philosopher. Then the graph transformation rules are specified as in Fig. 4 and the formula is specified in Groove as

```
(A G A F phil1full) & (A G A F phil2full) &
(A G A F phil3full)
```

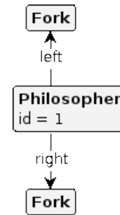


Fig. 4. Rule *phil1full*

The rule *philXfull* (see Fig. 5) is applicable when, and thus specifies the property that, the philosopher with ID X has forks in both hands.

The second example shows that in some cases, there can be an even more extreme blowup in necessary rule count. We want to check the property that a single fork is owned by exactly two adjacent philosophers. This is expressed in MocOCL as

```
self.forks->forAll(f | self.philosophers->
select(p | (sometimes eventually p.left = f or
p.right = f))->size() = 2)
```

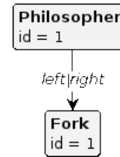


Fig. 5. Rule *phil1fork1*

In Groove, the statement `p.left = f or p.right = f` has to be specified for every fork and every philosopher. Thus, in the case of three philosophers nine graph transformation rules *philXforkY* have to be defined; in general, n^2 graph transformation rules for n philosophers are necessary. Additionally, to specify that for every fork, exactly two philosophers may use the fork, the specification of `((phil1forkX & phil1forkY) & !(phil1forkZ))` is required for every combination of X, Y, and Z. Consequently, the property in Groove grows rather large, even if we consider only three philosophers:

```
 (!((E F phil1fork1) & (E F phil1fork2) & (E F phil1fork3)) &
(((E F phil1fork1) & ((E F phil1fork2) | (E F phil1fork3))) |
((E F phil1fork2) & (E F phil1fork3)))) & (!((E F phil2fork1)
& (E F phil2fork2) & (E F phil2fork3)) & (((E F phil2fork1) &
((E F phil2fork2) | (E F phil2fork3))) | ((E F phil2fork2) & (E
F phil2fork3)))) & (!((E F phil3fork1) & (E F phil3fork2) & (E
F phil3fork3)) & (((E F phil3fork1) & ((E F phil3fork2) | (E F
phil3fork3))) | ((E F phil3fork2) & (E F phil3fork3))))
```

These small examples already illustrate the benefits of combining OCL with temporal logics. Since our implementation is currently a prototypical proof of concept, we do not achieve the same performance as Groove. Here, we identified the following problems and possible solutions to overcome these issues. First, the state space generation is slow and can be improved with more sophisticated isomorphism checks and retrieval of mapping information. Second, there is room for improvement in the evaluation of consecutive temporal operators, where we plan to apply more advanced evaluation algorithms. And third, the OCL engine imposes a bottleneck, and we are in the process of investigating if alternative implementations of OCL are better suited for our demands. Overall, we are convinced that it is possible to find solutions to these issues.

5 Related Work

One of the first to suggest a temporal extension to OCL were Conrad and Turowski [4]. For describing the interactions of software components in a *design by contract* manner they introduce an OCL extension that adds past and future temporal operators. Their work targets only the specification of the component's correct interaction, not the verification. Distefano *et al.* [5] propose a CTL-based logic, called BOTL, to specify static and dynamic properties of object-oriented systems without inheritance and sub-typing. Instead of extending OCL, they map OCL onto BOTL, thus providing a formal semantics for a large part of OCL based on BOTL. Ziemann *et al.* [21] aim to extend the semantics of the OCL standard with their proposed LTL-based extension, similar to our CTL solution. They need to extend the environment τ by an additional index i that points to the *current state*. Further, they analyze only finite sequences of states. We are not aware of any implentation of this approach. Soden and Eichler [20] also present an LTL based extension for OCL. They extend the semantics of the OCL standard and, like Ziemann *et al.*, they introduce an additional index into the evaluation environment that captures the current time instant. They suggest to define the operational semantics of MOF-conforming models with the M3Action framework. This allows them to define a finite execution trace by a sequence of changes from which the actual states are derived by applying the changes in succession to the initial model up to the current state. Flake and Mueller [8] aim at a tight integration of UML class diagrams, state charts, and OCL, where the state charts describe the behavior of the associated class diagrams. They use time-based traces to capture the evolution of the system

and propose a UML Profile to specify state-oriented, real-time invariants whose semantics are defined by a mapping to clocked CTL formulas. In regard to expressiveness, Bradfield *et al.* [2] propose the richest extension by embedding OCL into the observational μ-calculus. As noted by the authors this expressiveness comes at the price of complexity inherent to specifications using the μ-calculus. They thus suggest the use of predefined templates with concise semantics, but which hide the complexity of the underlying μ-calculus formula that is automatically generated from the template. We are not aware of any implementations realizing the above approaches.

Mullins and Oarga [16] present an extension to OCL, called EOCL, that augments OCL with CTL operators. They define EOCL's operational semantics over object-oriented transition systems. The SOCLe tool translates class, state chart, and object diagrams into an abstract state machine and it checks on-the-fly if the system satisfies the specification given as an EOCL expression. Kyas *et al.* [15] present a prototype that verifies OCL properties over simplified UML class diagrams whose behavior is described by state machines. In contrast to all other approaches presented thus far they do not extend OCL with temporal operators but rather translate class diagrams, state machines, and OCL specifications into the input format of the PVS theorem prover. With PVS they are able to prove OCL properties of infinite state systems. Similar to Bradfield *et al.*'s proposed templates, Kanso and Taha [12] introduce a temporal extension based on Dwyer *et al.*'s patterns for the specification of properties for finite state systems [6]. Although these patterns are not as expressive as their CTL, LTL, or μ-calculus counterparts, they greatly simplify the property specification process. Kanso and Taha define a scenario-based semantics for their extension, where each scenario is a finite sequence of events.

It is thus the combination of our two contributions, a CTL-based extension of OCL, whose formal syntax and semantics extend the OCL standard without modifying existing definitions, and the implementation of the MocOCL model checker, that distinguishes our approach from previous works.

6 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[5] that verifies cOCL specifications. With first showcases we illustrate that the combination of OCL and CTL expressions allows for compact formulations of specifications.

In future work, we plan to improve our implementation in terms of efficiency and usability. Work on a detailed performance analysis is currently in progress. For dealing with the state explosion problem, symbolic model checking techniques will be considered. Finally, we would like to apply our approach in the context of larger V&V environments as for example in USE [9].

[5] MocOCL is available at `www.modelevolution.org/prototypes/cocl`

References

1. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *MoDELS*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.
2. J. C. Bradfield, J. K. Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In *FASE*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.
3. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
4. S. Conrad and K. Turowski. Temporal OCL Meeting Specification Demands for Business Components. In *UML'01*, volume 2185 of *LNCS*, pages 151–165. Springer, 2001.
5. D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-Based Systems. In *FMOODS*, volume 177 of *IFIP Conf. Proc.*, pages 285–304. Springer, 2000.
6. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE*, pages 411–420. ACM, 1999.
7. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
8. S. Flake and W. Müller. Formal semantics of static and temporal state-oriented ocl constraints. *Software and System Modeling*, 2(3):164–186, 2003.
9. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.
10. O. M. Group. Object Constraint Language (OCL) V2.2. `http://www.omg.org/spec/OCL/2.2/`, February 2010.
11. T. Jussila, J. Dubrovin, T. Junttila, T. L. Latvala, and I. Porres. Model Checking Dynamic and Hierarchical UML State Machines. In *Proc. MoDeV^{2}a: Model Development, Validation and Verification*, pages 94–110, 2006.
12. B. Kanso and S. Taha. Temporal Constraint Support for OCL. In *SLE*, volume 7745 of *LNCS*, pages 83–103. Springer, 2012.
13. H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In *SPIN*, volume 3925 of *LNCS*, pages 299–305. Springer, 2006.
14. A. Knapp and J. Wuttke. Model Checking of UML 2.0 Interactions. In *MoDELS Workshops*, volume 4364 of *LNCS*, pages 42–51. Springer, 2006.
15. M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML Models and OCL Constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.
16. J. Mullins and R. Oarga. Model Checking of Extended OCL Constraints on UML Models in SOCLe. In *FMOODS*, volume 4468 of *LNCS*, pages 59–75. Springer, 2007.
17. A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.
18. M. Richters and M. Gogolla. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 42–68. Springer, 2002.
19. B. Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.
20. M. Soden and H. Eichler. Temporal Extensions of OCL Revisited. In *ECMDA-FA*, volume 5562 of *LNCS*, pages 190–205. Springer, 2009.
21. P. Ziemann and M. Gogolla. OCL Extended with Temporal Logic. In *Ershov Memorial Conference*, volume 2890 of *LNCS*, pages 351–357. Springer, 2003.