OCL Extended with Temporal Logic

Paul Ziemann and Martin Gogolla

University of Bremen, Department of Computer Science
P.O. Box 330440, D-28334 Bremen, Germany
{ziemann|gogolla}@informatik.uni-bremen.de

Abstract. UML class diagrams have become a standard for modeling the static structure of object-oriented software systems. OCL can be used for formulating additional constraints that can not be expressed with the diagrams. In this paper, we extend OCL with temporal operators to formulate temporal constraints.

1 Introduction

UML class diagrams are popular for modeling the static structure of object-oriented software systems. Syntax and semantics of UML diagrams are semi-formally defined in [8]; OCL, a textual language similar to predicate logic which is used to formulate additional constraints, is also defined there in the same semi-formal way. A formal semantics for UML class diagrams and OCL was given in the current OCL 2.0 OMG submission [2]. OCL expressions used in invariants are evaluated in a single system state. OCL pre- and postconditions characterize operations by considering state transitions, i.e. state pairs.

Temporal logic, as an extension of predicate logic, has been used successfully in the field of software development (see [7] among other approaches). The basic idea of linear temporal logic is to consider not only single states or state pairs, but to care about arbitrary state sequences. By doing so, it is possible to characterize system development by specifying the allowed system state sequences.

In this paper, we present an extension of OCL with important elements of a linear temporal logic. Past and future temporal operators are introduced. Our extended version of OCL, which we call *TOCL* (*Temporal OCL*), allows software engineers to specify constraints on the temporal evolution of a system structure. Since the temporal elements are smoothly integrated in the common OCL syntax, TOCL is easy to use for an engineer familiar with OCL. Another motivation is that several high-level UML/OCL constructs could be reduced to constructs of lower level with additional TOCL constraints. We define the extension by building upon the formal definition of OCL presented in [2]. This paper is a polished version of [10] and a short version of [11].

There is already work to extend OCL with temporal logic in various directions. [9, 4] extend OCL with operators of a linear temporal logic. However, the paper does not give a formal foundation of the extension. In [5], the object-based temporal logic BOTL is defined, which is based on the branching temporal logic CTL and a subset of OCL. Inheritance and subtyping are not considered

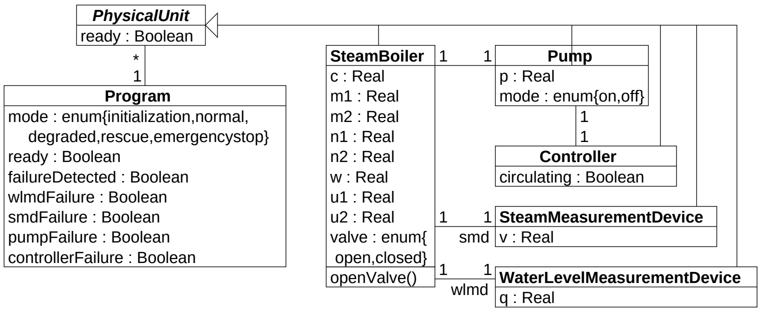

Fig. 1: Object model for the steam-boiler problem

in this approach. [6] presents an OCL extension, equally based on CTL, for specification of state-oriented constraints. This extension concerns system behavior modeled by statechart diagrams; but the development of attributes is not considered there. In [3], OCL is extended with temporal constructs based on the observational mu-calculus. The authors suggest using "templates" with user-friendly syntax which then have to be translated to $\mathcal{O}\mu$(OCL). However, we think our direct semantics in terms of set theory is useful as well due to its comprehensibility.

2 Basic Idea

To give an idea of the usefulness of a temporal OCL, we demonstrate how parts of the "Steam-boiler control specification problem" [1] can be specified using TOCL. The underlying object model is shown in Fig. 1.

The first invariant states that when a program is in initialization mode, it remains in this mode until all physical units are ready or a failure of the water level measurement device has occurred. We use the temporal operator 'always-until' in this example.

```
context Program inv:
  self.mode = #initialization implies
    always self.mode = #initialization
    until (PhysicalUnit.allInstances->forAll(pu | pu.ready)
           or self.wlmdFailure)
```

The next invariant requires that the program starts in the mode 'initialization'. Here we use the well known operation 'oclIsNew', which can be used in TOCL invariants and not only in postconditions as it is the case in OCL.

```
context Program inv:
  self.oclIsNew implies self.mode = #initialization
```

The following invariant applies the operator 'next' to specify that the mode changes from 'initialization' to 'emergencystop' if a failure of the water level measurement device is detected.

```
context Program inv:
  (self.mode = #initialization and self.wlmdFailure)
  implies next self.mode = #emergencystop
```

When the valve of a steam boiler is open, the water level measured by the water level measurement device (attribute q) will be lower or equal to the normal upper boundary of water level (attribute n2) sometime. Here, the operator 'sometime' is applied.

```
context SteamBoiler inv:
  self.valve=#open implies sometime self.wlmd.q <= n2
```

Of course, TOCL can be used in pre- and postconditions as well. The operation 'openValve()' causes the valve of the steam-boiler to be open until the water level sinks under the normal upper boundary n2. The operator 'always-until' is used in the postcondition.

```
context SteamBoiler::openValve()
  post: always valve = #open until wlmd.q <= n2
```

3 Object Models

In our context, an object model uses all those UML concepts that are essential for modeling structural aspects of a problem domain. An object model can be visualized by a UML class diagram. Instances of an object model (i.e. states of the modeled system) can be visualized by a UML object diagram. Object models are referred to by TOCL constraints and are therefore a prerequisite for the TOCL definition. We adopt the object model definition presented in [2] but extend it with *state sequences*. An *object model*

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \text{associates}, \text{roles}, \text{multiplicities}, \prec)$$

consists of a set of classes ($\textsc{Class}$) with each class c having attributes ($\textsc{Att}_c$) and operations ($\textsc{Op}_c$) assigned to it. Associations ($\textsc{Assoc}$) connect classes with each other. The functions 'associates', 'roles' and 'multiplicities' assign associated classes, role names and multiplicities to associations, respectively. $\prec$ is an irreflexive partial order on the set of classes, representing a generalization hierarchy. An object model specifies the possible states of a system. A *system state*

$$\sigma(\mathcal{M}) = (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Assoc}}),$$

also called a snapshot of a running system, consists of the existing objects ($\sigma_{\textsc{Class}}$) with its current attribute values ($\sigma_{\textsc{Att}}$) and links ($\sigma_{\textsc{Assoc}}$) connecting them. We write σ instead of $\sigma(\mathcal{M})$ if the model is clear from the context.

An OCL constraint is evaluated in a single system state. Since TOCL is intended for formulating constraints on the temporal development of a system, a single system state is not sufficient here. We therefore introduce infinite *state sequences* for a model $\mathcal{M}$, denoted as

$$\hat{\sigma}(\mathcal{M}) = \langle \sigma_0, \sigma_1, \dots \rangle.$$

The order of the states reflects a temporal relationship; that is, the system is in state σ_0 at the beginning, later in state σ_1, and so on. Again, we write $\hat{\sigma}$ instead of $\hat{\sigma}(\mathcal{M})$ if the model is clear from the context. For example, $\sigma_{0\,\mathrm{CLASS}}(c)$ is the set of objects of class c existing in the first state of the sequence. Finite state sequences, which we do not explicitly consider, can be seen as infinite ones with one state that is followed only by empty states, i.e. states without objects.

4 TOCL Types

TOCL is a strongly typed language. We adopt the type system of OCL defined in [2]. Each type t is mapped to its domain by a function I. Each operation on a type t is mapped to a function $I(\hat{\sigma}, i)$, where i is the so called reference index denoting the current state. Therefore, the semantics of an operation can depend on a state sequence and a reference index. However, most operations only depend on the current state or none state at all.

The types of (T)OCL can by divided into several groups. *Integer*, *Real*, *Boolean* and *String* with the expected domains and operations are the basic types. Enumeration types are user-defined; that is, the user specifies the name and a list of possible values for the type. Object types are derived from the object model. There is an object type for each class, having the same name as the corresponding class. The domain of an object type is an infinite set of objects of the class.

Collections of values can be described by the complex types *Set*(t), *Sequence*(t), *Bag*(t), and *Collection*(t). The parameter t denotes the type of the elements of the set, sequence, or bag (multi-set), respectively. Each type has a special undefined value $\perp$ contained in its domain.

There is a subtype relationship between certain types that is defined by a reflexive partial order $\leq$ on the set of types.

$\mathrm{allInstances}_t : \to \mathit{Set}(t)$ and $\mathrm{oclIsNew}_t : t \to \mathit{Boolean}$ are two of the operations defined for all object types. In OCL, 'oclIsNew' is only applicable in postconditions. In TOCL, it can be applied in invariants and preconditions as well. The result of '$\mathrm{allInstances}_t$' is the set of objects of type t existing in the current state. This includes instances of child classes of the corresponding class. The operation 'oclIsNew' can be used to check whether an object is new, that is, whether it exists in the current state and not in the preceding. In the first state of a sequence, all existing objects are new. Other operations exist to access attribute values or navigate along links connecting objects. For user-defined operations the user has to provide a OCL expression that specifies the semantics.

5 TOCL Expressions and Constraints

Due to space limitations, we define syntax and semantics of temporal expressions and constraints in this section only informally. The formal definition can be found in [11].

5.1 Temporal expressions

Syntax and semantics of OCL expressions is defined in [2] by giving six rules each. We do not repeat them here but add further rules for temporal expressions.

Expressions are evaluated in an environment consisting of a state sequence $\hat{\sigma}$, a reference index i denoting the current state, and variable assignment β. The variable assignment influences the evaluation of free variables, the choice of the current state effects the evaluation of object operations like navigation or attribute access. The current state together with the whole state sequence is necessary for the evaluation of temporal expressions.

In the following, we enumerate the temporal expressions and explain their semantics. Let e, e_1 and e_2 be boolean expressions, $a_1, \ldots, a_n$ expressions of types $t_1, \ldots, t_n$, and $\omega : t_1, \ldots, t_n \rightarrow t$ an operation.

'next e' is true if e is true "in the next state"; that is, if e is true with a reference index incremented by one. In all other cases the evaluation results in false. 'always e' is true if e is true in the current state and in all future states of the sequence, otherwise false. An expression 'sometime e' is true if e is true in the current state or in one of the future states. Otherwise it is false. 'always e_1 until e_2' is true if e_1 is true "from now on" until e_2 is true for the first time in future. The expression is also true if e_2 is never true and e_1 is true in all future states. Otherwise it is false. For an expression 'sometime e_1 before e_2' to be true, e_1 just has to be true in at least one future state. One of these has to be before the next future state e_2 is true in (if there is one). When evaluating 'ω@next$(a_1, \ldots, a_n)$', the argument expressions are evaluated in the current state but the operation ω is evaluated in the next state. This expressions is also written as '$a_1.\omega$@next$(a_2, \ldots, a_n)$'. If a_1 denotes a collection value, an arrow symbol is used instead of the period. Operations such as addition are denoted in infix notation.

The past expressions 'previous e', 'alwaysPast e', 'sometimePast e', 'always e_1 since e_2', 'sometime e_1 since e_2, and the modifier @pre are defined analogously. They behave like the future expressions flipped (with respect to the temporal ordering) across the current state, with the difference that the sequence of past states is bounded by the first state.

5.2 Constraints

A TOCL constraint can either be an invariant or an operation specification. We informally define the slightly modified semantics of invariants. In addition, we give an idea of how the semantics of pre- and postconditions could be defined.

Invariants. An invariant is a condition that must be satisfied in all system states. Let e be a boolean expression with free variables $v_1, \ldots, v_n$. An invariant **context** $v_1 : t_1, \ldots, v_n : t_n$ **inv**: e is *valid in a state sequence* $\hat{\sigma}$ if the following expression is true in the first state:

```
always(
  t1.allInstances->forAll(v1:t1 |
    ...
    tn.allInstances->forAll(vn:tn |
      e
    )...))
```

The expression e is extended to an expression that prepends the 'always' operator and quantifies the declared variables over the set of objects of the respective type existing in the respective state. The fact that an invariant is a condition that must *always* be satisfied is therefore made explicit here. Every OCL invariant is also a TOCL invariant with semantics as expected. If there is no variable declared explicitly but only a type is given as context, the variable 'self' of this type is implicitly declared.

Pre- and Postconditions. Pre- and postconditions are part of operation specifications. They are used to specify conditions to be satisfied before and after the execution of a user-defined operation, respectively.

For an operation specification to be valid in a state sequence, the postcondition must be satisfied in the poststate of all executions of the given operation if the precondition is satisfied in the corresponding prestate. In TOCL, the conditions can either be standard OCL expressions, or boolean past expressions (in preconditions), or boolean future expressions (in postconditions).

To define these semantics formally, a system state would have to hold information about invoked and terminated operations. Then, operation specifications could be reduced to invariants as it is done in [5]. We present this approach in [11].

6 Summary and Conclusions

In this paper we presented TOCL, an extension of OCL with elements of a linear temporal logic. We have started with a description of object models, modeling the static structure of a system. System states have been introduced as snapshots of a running system. Multiple states, which are ordered in time, form state sequences that constitute part of the environment for evaluating TOCL expressions.

We have proceeded with an outline of the TOCL type system. In the central part of the paper, temporal operators have been introduced to combine boolean expressions to new ones. Expressions have been used to form constraints, i.e. invariants and operation specifications (with pre- and postconditions).

There are some issues that are topics of further research. It could be examined to which extent TOCL is capable of describing properties of the various UML diagram types. While TOCL needs a class diagram providing the context, other diagram types (like certain statechart diagrams) could be explained by appropriate TOCL constraints, maybe in part or even completely.

References

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, volume 1165 of *LNCS*. Springer, 1996.

[2] Boldsoft, Rational Software Corporation, and IONA. Response to the UML 2.0 OCL RfP (ad/2000-09-03), June 2002. Internet: `http://www.klasse.nl/ocl/subm-draft-text.html`.

[3] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[4] Stefan Conrad and Klaus Turowski. Temporal OCL: Meeting Specification Demands for Business Components. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 10, pages 151–166. Idea Publishing Group, 2001.

[5] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a Temporal Logic for Object-Based Systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 305–326. Kluwer Acadeim Publishers, 2000. Report version: TR–CTIT–00–06, Faculty of Informatics, University of Twente.

[6] Stephan Flake and Wolfgang Mueller. A UML Profile for Real-Time Constraints with the OCL. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephan Cook, editors, *UML 2002 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 179–195. Springer, 2002.

[7] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[8] OMG. *OMG Unified Modeling Language Specification, Version 1.5, March 2003*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2003.

[9] Sita Ramakrishnan and John McGregor. Extending OCL to Support Temporal Operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*, 1999.

[10] Paul Ziemann and Martin Gogolla. An Extension of OCL with Temporal Logic. In Jan Jürjens, Maria Victoria Cengarle, Eduardo B. Fernanez, Bernhard Rumpe, and Robert Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 53–62. TUM, Institut für Informatik, September 2002. TUM-I0208.

[11] Paul Ziemann and Martin Gogolla. An OCL Extension for Formulating Temporal Constraints. Technical report, Universität Bremen, 2003.