Past- and Future-Oriented Time-Bounded Temporal Properties with OCL

Stephan Flake and Wolfgang Mueller

C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany

E-mail: {flake,wolfgang}@c-lab.de

Abstract

We present the syntax and semantics of a past- and future-oriented temporal extension of the Object Constraint Language (OCL). This extension supports designers to express time-bounded properties over a state-oriented UML model of a system under development. The semantics is formally defined over the system states of a mathematical object model. Additionally, we provide a mapping to Clocked Linear Temporal Logic (Clocked LTL) formulae, which is the basis for further application in appropriate model checking verification tools. We demonstrate the applicability of the approach by the example of a buffer within a production system.

1 Introduction

The Unified Modeling Language (UML) defines a number of diagrams to model different aspects of the structure and behavior of software systems [20]. For example, Class Diagrams are used to describe the static structure of a system, while UML State Diagrams model the (reactive) behavior of objects. In addition to the set of diagrams, the textual Object Constraint Language (OCL) is an integral part of UML to specify further restrictions over values of (parts of) a given UML model [19]. Significant parts of OCL have already been formally defined in [24] based on the set-theoretic definition of an *object model*. That work heavily influenced the formal semantics of the recently adopted OCL 2.0 proposal [19].

UML has already been applied in different domains, e.g., to model *time-critical* software-controlled systems such as embedded real-time systems [6]. For time-critical systems, correct time-constrained behavior is an essential requirement to meet. In this context, it is desirable to be able to identify improper behavior w.r.t. such *time-bounded temporal properties* already in early phases of development. Otherwise, overall goals like meeting project deadlines and adherence to estimated costs may fail due to the need of time-consuming and expensive re-designs at a later stage of development.

UML currently provides only limited support for the specification of *temporal properties* such as safety or liveness constraints [17] – let it be with or without explicit time. Different approaches have already introduced extensions to overcome this deficiency, e.g., extensions of UML Sequence Diagrams to enhance time-bounded specifications of *event-based* communication among objects [9, 5, 16]. In contrast, we focus on the specification of time-bounded *state-oriented* constraints to reason about the time-critical system execution.

In our previous work, we already introduced a future-oriented temporal extension of OCL [13]. We chose OCL for our specification approach, as it already supports operations for sets and sequences to extract and manipulate collections (in particular, collections of states). We can thus reuse existing UML concepts and keep the learning curve low for people that already know UML and OCL. The semantics of our temporal OCL extension is defined over *traces* of the referred UML user model. Traces are sequences of *system states* that keep all information necessary to evaluate OCL expressions.

For further application in a verification tool, we additionally defined a mapping to a *temporal logics* called Clocked Computation Tree Logic (CCTL) [27]. Temporal logics are frequently applied to formally specify required behavioral properties of a system under development. The most popular temporal logics used in the area of formal verification are Linear Temporal Logic (LTL) and the branching-time Computation Tree Logic (CTL) [21, 8]. Most temporal logics support future-oriented temporal operators, but past time operators can often be very useful to express required properties in an easier way [18]. Note that past time operators do not necessarily add expressive power to temporal logics that solely rely on future-oriented temporal operators [14]. Due to space limitations, we do not go into more details about different temporal logics here. Instead, we refer to [3, 15] for introductions to temporal logics and their appli-

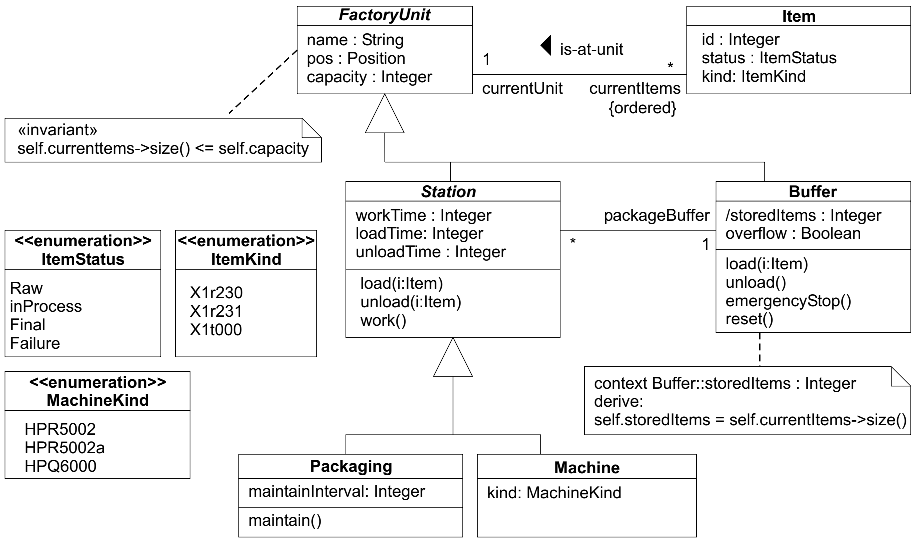

Figure 1. Parts of the UML Class Diagram of the Case Study

cation in formal verification. Furthermore, [22] provides a good overview of the history and application of past time temporal logics.

Compared to branching-time temporal logics, the semantics of *linear temporal logics* is often seen as being *more intuitive* for modelers that are not experts in formal methods, as linear temporal logics define the validity of a formula over a given *entire execution trace*. This combines well with the mental model of people that are used to think in terms of simulation-based tests and validations. In this article, we therefore focus on an LTL-based language as the semantic foundation for our temporal OCL extension. We provide a mapping of our temporal OCL extension to a time-bounded variant of linear temporal logics that we call *Clocked Linear Temporal Logics with Past* (Clocked LTL).

The remainder is structured as follows. The next section presents an example that is later used to demonstrate the applicability of the approach. Sections 3 and 4 then briefly outline the temporal logics Clocked LTL and standard OCL, respectively. Section 5 presents our state-oriented temporal extensions to OCL and a mapping to Clocked LTL. Section 6 shows some time-bounded constraints in the context of the example in both temporal OCL as well as corresponding Clocked LTL formulae. Section 7 briefly discusses related work. Section 8 closes with a conclusion.

2 Example

In this section we present a rather generic example of a buffer that is part of a more complex production system. A UML class diagram that presents the system structure and an object diagram that gives the initial setting are shown in Figures 1 and 2.

The buffer is used to store production items delivered by three preceding machines. It has limited space for items, e.g., 17 items can maximally be stored. The three machines cyclically output items with different periods, i.e., 5, 6, and 7 time units. Items are taken from the buffer by a rather fast packaging unit. However, the packaging unit has to be maintained in certain intervals, e.g., every 40 time units for the length of 10 time units. During maintenance the packaging unit cannot take any items from the buffer.

We can already specify with standard OCL that the capacity of the machines and buffer must always be regarded, such that no overflow occurs. The corresponding OCL invariant is

```
self.currentItems->size() <= self.capacity .
```

However, enhanced temporal properties cannot directly be expressed with UML or OCL means, e.g., that

- as long as no error occurs, the buffer takes items from the machines and eventually puts them into the packaging unit, and
- every overflow in the buffer is due to an error in the packaging unit (causality w.r.t. the past).

In Section 6, we will show that such properties can be expressed with our OCL extension.

3 Clocked Linear Temporal Logic with Past

Various variants of temporal logics with time-bounds exist. We here only mention Timed Linear Temporal Logic (TLTL) [29], RTCTL [8], and CCTL [27]. Most temporal

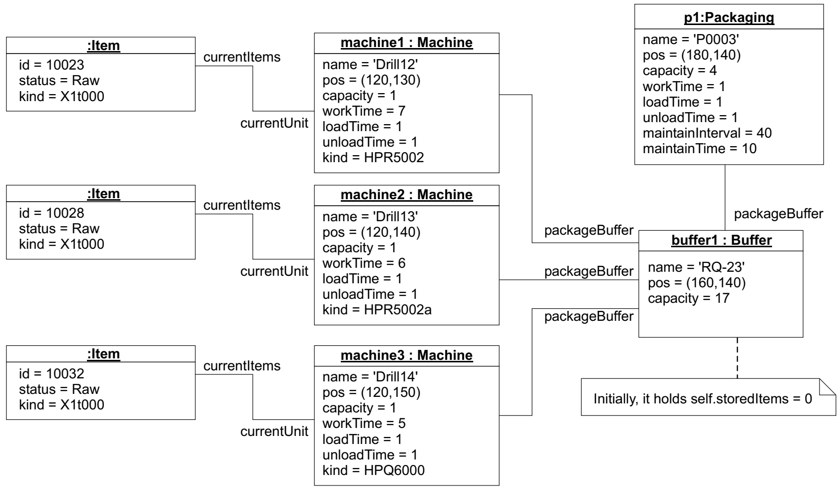

Figure 2. UML Object Diagram of the Initial Situation of the Case Study

logics focus on future-oriented temporal operators such as 'next' or 'eventually'.

In this article, we present a new variant called *Clocked Linear Temporal Logic (Clocked LTL)*. The syntax of Clocked LTL is recursively defined by the following grammar:

$$
\begin{array}{llllll}
\phi ::= & p & \mid \text{true} & \mid \text{false} & \mid (\phi) \\
 & \mid \neg\phi & \mid \phi \wedge \phi & \mid \phi \vee \phi & \mid \phi \rightarrow \phi \\
 & \mid \mathrm{X}_{[a]}\ \phi & \mid \mathrm{F}_{[a,b]}\ \phi & \mid \mathrm{G}_{[a,b]}\ \phi & \mid \phi\ \mathrm{U}_{[a,b]}\ \phi \\
 & \mid \mathrm{P}_{[a]}\ \phi & \mid \mathrm{F}_{\text{past [a,b]}}\ \phi & \mid \mathrm{G}_{\text{past [a,b]}}\ \phi & \mid \phi\ \mathrm{S}_{[a,b]}\ \phi
\end{array}
$$

where p is an element of a set Pr of propositions, $a \in \mathbb{N}_0$, and $b \in \mathbb{N}_0 \cup \{\infty\}$. The symbol ∞ is defined through: $\forall i \in \mathbb{N}_0 : i < \infty$, and it holds $i + \infty = \infty$ and $\infty - i = \infty$ and $i - \infty = 0$ (the latter rule is particularly necessary for well-defined time-bounded past temporal operators).

The letters are acronyms for the usual temporal logic operators. The future-oriented operators are X (for neXt), F (eventually), G (globally), and U (until). The past time-oriented operators are P (for previous), GP (globally in the past), F_{past} (eventually in the past), and S (since). The temporal operators F, G, U, G_{past}, F_{past}, and S are provided with interval time-bounds $[a, b]$. However, we also allow that these operators have a single time-bound only. In this case the lower bound is set to zero by default. It is also allowed to specify no timing annotation at all. In this case, the lower bound is zero and the upper bound is infinity by default. The X- and P-operators have a single time-bound $[a]$ only (here, $a \in \mathbb{N}$). If no time bound is specified, it is implicitly set to one. The operator precedence is categorized into five groups as follows (ordered from high to low): (1) $\neg$, (2) $\mathrm{G}_{[a,b]}$, $\mathrm{F}_{[a,b]}$, $\mathrm{X}_{[a]}$, $\mathrm{G}_{\text{past}[a,b]}$, $\mathrm{F}_{\text{past}[a,b]}$, $\mathrm{P}_{[a]}$, (3) $\wedge$, $\vee$, (4) $\rightarrow$, (5) $\mathrm{U}_{[\text{a,b}]}$, $\mathrm{S}_{[a,b]}$.

Of course, several additional operators known from the literature can additionally be supported, e.g., logical operators like equivalence and xor and temporal operators like 'before', 'weak until', and 'eventually since'.

Semantics. The validity of a Clocked LTL formula is defined over a *trace* of a model that is given as a time-annotated Kripke structure $\mathcal{K}$. Note here that the particular execution semantics of $\mathcal{K}$ are not essential for the definition of the temporal logics. We only require that a discrete time step (i.e., a time unit) passes between two subsequent states of a Kripke structure, such that we can speak of a trace v with time steps 0, 1, 2, 3, ...

It is thus sufficient here to define $\mathcal{K}$ as a tuple (Pr, S, Tr, L, I) where Pr is a set of atomic propositions, S is a set of states, $Tr \subseteq S \times S$ is a total transition relation, and $L : S \rightarrow 2^{Pr}$ is a labeling function, such that L labels each state of S with a set of propositions that are true in that state. Finally, I is a time labeling function that defines delay times in $\mathcal{K}$. For example, I can be a transition labeling $I : Tr \rightarrow 2^{\mathbb{N}}$ that defines delay times of transitions (cf. Interval Structures [28]). However, other labelings and slightly different execution semantics for $\mathcal{K}$ are possible.

A *trace* $v : \mathbb{N}_0 \rightarrow S$ over discrete time is an infinite sequence $g_0, g_1, \ldots$ of states $\in S$, where for all $i \in \mathbb{N}_0$ holds $(g_i, g_{i+1}) \in Tr$. The semantics of Clocked LTL formulas is defined by a satisfiability relation $\models$ over traces. We use $v \models_t \phi$ to denote that trace v satisfies formula ϕ at time t. The satisfiability relation that recursively defines the semantics of Clocked LTL formulae is shown in Table 1. In that table, ϕ and ψ denote arbitrary Clocked LTL (sub)formulae and $0 \leq a \leq b \in \mathbb{N}_0$. Note that we require $a > 0$ for the operators $\mathrm{X}_{[a]}$ and $\mathrm{P}_{[a]}$. The semantics of a Clocked LTL formula over an entire trace is then as follows.

Table 1. Description of Clocked LTL Operators

Formula	Description
$v \models_t p \ (p \in Pr)$	if $p \in L(v(t))$
$v \models_t \neg\phi$	if $v \models_t \phi$ is false
$v \models_t \phi \wedge \psi$	if $v \models_t \phi$ and $v \models_t \psi$
$v \models_t \phi \vee \psi$	if $v \models_t \phi$ or $v \models_t \psi$
$v \models_t \phi \rightarrow \psi$	if $v \models_t \phi$ implies $v \models_t \psi$
$v \models_t \mathrm{X}_{[a]}\ \phi$	if $v \models_{t+a} \phi$
$v \models_t \mathrm{F}_{[a,b]}\ \phi$	if there is an i with $t+a \leq i \leq t+b$ such that $v \models_i \phi$
$v \models_t \mathrm{G}_{[a,b]}\ \phi$	if for all i with $t+a \leq i \leq t+b$ holds $v \models_i \phi$
$v \models_t \phi\ \mathrm{U}_{[a,b]}\ \psi$	if there is an i with $t+a \leq i \leq t+b$ such that $v \models_t \psi$ and for all j, $t+a \leq j < i$ holds $v \models_j \phi$.
$v \models_t \mathrm{P}_{[a]}\ \phi$	if $t-a \geq 0$ and $v \models_{t-a} \phi$
$v \models_t \mathrm{F}_{\mathrm{past}[a,b]}\ \phi$	if $t-b \geq 0$ and there is an i with $t-b \leq i \leq t-a$ such that $v \models_i \phi$
$v \models_t \mathrm{G}_{\mathrm{past}[a,b]}\ \phi$	if $t-b \geq 0$ and for all i with $t-b \leq i \leq t-a$ holds $v \models_i \phi$
$v \models_t \phi\ \mathrm{S}_{[a,b]}\ \psi$	if $t-b \geq 0$ and there is an i with $t-b \leq i \leq t-a$ such that $v \models_i \psi$ and for all j, $i < j \leq t-a$ holds $v \models_j \phi$

Definition 1 *Let ϕ be a Clocked LTL formula and v be a trace. v satisfies ϕ (denoted by $v \models \phi$) iff $v \models_0 \phi$.*

For specification purposes, it is often necessary to distinguish whether a property expressed by a (C)LTL formula has to hold over *all* possible traces or only over at least one trace.

Definition 2 *Let $\mathcal{K}$ be a discrete-time Kripke Structure and ϕ be a Clocked LTL formula. $\mathcal{K}$ satisfies ϕ* on all traces *(denoted by $\mathcal{K} \models_A \phi$) iff for all possible traces v of $\mathcal{K}$ holds $v \models_0 \phi$. We say that that $\mathcal{K}$* can satisfy *ϕ (denoted by $\mathcal{K} \models_E \phi$) iff there is a trace v of $\mathcal{K}$ with $v \models_0 \phi$.*

Though temporal logics are powerful languages that can produce arbitrarily nested specifications, their full expressive power is not needed in practice. This led to property specification pattern systems [7] and related approaches that abstract from temporal logics. With our OCL extension, we also follow this idea. We nevertheless have to define a *mapping* to a temporal logic like Clocked LTL to be able to make use of appropriate automatic verification tools.

Note that our variant Clocked LTL is not more expressive than other time-bounded linear temporal logics from a theoretical viewpoint. It is therefore possible to map Clocked LTL to other temporal logics and thus use existing verification tools, e.g. SPIN[1]. The reason for defining Clocked LTL is that the syntax and semantics of existing logics do not exactly match our requirements. All temporal logics we found so far either do not consider timing intervals, do not consider past time operators, or are defined over continuous time. In contrast, Clocked LTL avoids cryptical symbols (like diamonds, squares, and circles), is based on discrete time for practical purposes, and supports future as well as past temporal operators in combination with timing intervals.

[1]http://spinroot.com/spin/whatisspin.html

4 Introduction to OCL

OCL is a declarative expression-based language to constrain values in the context of a given UML model. Evaluation of OCL expressions does not have side effects on the corresponding UML model. In the remainder, we will call this UML model the *referred user model*.

Each OCL expression has a type. Besides user-defined model types (e.g., classes or interfaces) and some predefined basic types (e.g., `Integer`, `Real`, or `Boolean`), OCL also has a notion of object collection types (i.e., sets, ordered sets, sequences, and bags). Collection types are homogeneous in the sense that all elements of a collection have a common type. In contrast, OCL 2.0 now also supports tuples that are sequences of a fixed number of elements that can be of different types. Moreover, a standard library is available that provides operations to access and manipulate values and objects.

OCL constraints can be visually applied as stereotyped notes that are attached to their corresponding class as shown in Figure 1 in Section 2. Alternatively, they can be formulated separately in pure textual form, but then the context class has to be provided. For example, the following invariant ensures that each instance of class `Machine` has one associated package buffer and that this buffer is the same for all instances of class `Machine`:

```
context Machine
inv: self.packageBuffer->size() = 1
     and
     Machine.allInstances().packageBuffer
                                  ->asSet()->size() = 1
```

The class name that follows the `context` keyword specifies the class for which the following expression should hold. The keyword `inv` indicates that this is an invariant that has to hold for each object of the context class at all times. The keyword `self` refers to each object of the context class. Attributes, operations, and associations can be accessed by dot notation, e.g., `self.myBuffer` results in a (possibly empty) set of instances of `Buffer`. The arrow notation indicates that a collection of objects is manipulated by one of the pre-defined OCL collection operations. For example, operation `notEmpty()` returns true, if the accessed set is not empty.

The predefined operation `allInstances()` is applied to class `Machine` to extract the set of currently existing `Machine` objects. The expression `Machine.allInstances().packageBuffer` then refers to the multiset (or: bag) of buffer objects when navigating from each `Machine` object via association `packageBuffer` to class `Buffer`. This multiset is casted to an ordinary set by applying the predefined operation `asSet()`. We finally require that the size of this set is equal to one, and we thus have specified that each machine object is associated with the same buffer object.

Note that it is also possible to formulate checks for activated State Diagram states that are associated with objects, using the operation `oclInState(stateName:OclState):Boolean`. The existing – though still informal – notion of states in OCL is especially important for our temporal OCL extension.

5 Temporal OCL Extension

The concrete syntax of OCL 2.0 is defined by an attributed grammar in EBNF (Extended Backus-Naur Form) with inherited and synthesized attributes as well as disambiguating rules. For each production rule, a mapping to the corresponding concept in the abstract syntax (i.e., the metamodel) is provided.

Based on this grammar and a UML Profile that introduces stereotypes for temporal expressions, we already introduced future-oriented temporal OCL expressions in [13]. The basic idea is to interpret a temporal OCL expression as a special form of operation call. An operation call in the abstract OCL syntax has a source, a referred operation, and operation arguments (see Figure 3). Corresponding attribute values have to be set and become part of the abstract syntax tree. The dedicated variable `ast` (**a**bstract **s**yntax **t**ree) is used to store these values. The type of `ast`

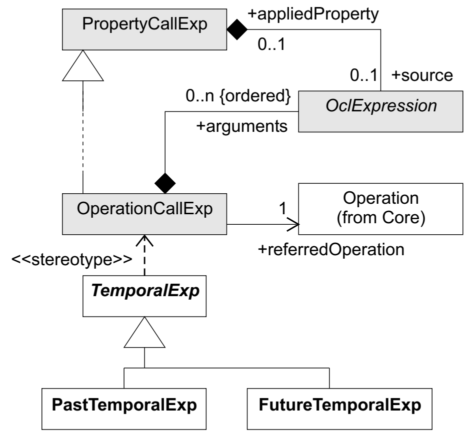

Figure 3. Temporal Expressions as Special Form of Operation Call Expressions

differs depending on the production rule and refers to a type of the OCL metamodel. In this case, `ast` is of type `PastTemporalExp`. The following new rule now gives the main production rule for *past temporal OCL expressions*. Note that we introduce a temporal operator '@' to distinguish temporal expressions from OCL's common dot and arrow notation for accessing attributes, operations, and associations.

```
PastTemporalExpCS ::= OclExpressionCS '@'
                      simpleNameCS '(' argumentsCS? ')'
Abstract Syntax Mapping:
   PastTemporalExpCS.ast : PastTemporalExp
Synthesized Attributes:
   PastTemporalExpCS.ast.source =   OclExpressionCS.ast
   PastTemporalExpCS.ast.arguments =    argumentsCS.ast
   PastTemporalExpCS.ast.referredOperation =
      OclExpressionCS.ast.type.lookupOperation(
                  simpleNameCS.ast,
                  if argumentsCS->notEmpty()
                    then argumentsCS.ast->collect(type)
                    else Sequence{}
                  endif )
Inherited Attributes:
   OclExpressionCS.env =          PastTemporalExpCS.env
   argumentsCS.env =              PastTemporalExpCS.env
Disambiguating Rules:
-- Operation name must be a past time temporal operator
   [1] Set{'pre'}->includes(simpleNameCS.ast)
-- The operation signature must be valid
   [2] not PastTemporalExpCS.ast.referredOperation.
                                         oclIsUndefined()
```

Thus, past time temporal OCL expressions map to the specific UML stereotype `PastTemporalExp` that inherits values from `OperationCallExp` on the metamodel level see Figure 3). Additional temporal operations can easily be introduced at a later point of time, as just the disambiguating rule [1] has to be extended in such cases.

5.1 Semantics

OCL 2.0 provides extensive semantic descriptions by both a metamodel-based as well as a formal mathematical approach. In the remainder, we focus on the *formal OCL semantics* that is based upon the notion of an set-theoretic *object model* [19]. An object model $\mathcal{M}$ is a tuple with a set $CLASS$ of classes, a set ATT of attributes, a set OP of operations, a set $ASSOC$ of associations, a generalization hierarchy $\prec$ over classes, and functions $associates$, $roles$, and $multiplicities$ that give for each $as \in ASSOC$ its dedicated classes, the classes' role names, and multiplicities, respectively.

In the remainder, we call an instantiation of an object model a *system*. A system changes over time, i.e., the (number of) objects, their attribute values, and other characteristics change during system execution. The information to evaluate OCL expressions is stored in *system states*, which represent snapshots of the running system. In OCL 2.0, a system state $\sigma(\mathcal{M})$ is formally defined as a triple with a set Σ_{CLASS} of currently existing objects, a set Σ_{ATT} of attribute values of the objects, and a set Σ_{ASSOC} of currently established links.

The object model and system state definition, however, lack descriptions of ordered sets, global OCL variable definitions, OCL messages, and states of UML State Diagrams. Especially the latter are needed for our temporal OCL semantics. We therefore extend the formal model and system states accordingly, such that the resulting *extended object model* $\mathcal{M}$ with

$$\begin{aligned}\mathcal{M} = \langle\ & CLASS, ATT, OP, SIG, SC, ASSOC,\\ & paramKind, isQuery, \prec, \prec_{sig},\\ & associates, roles, multiplicities\ \rangle\end{aligned}$$

additionally includes operation parameter kinds {*in, inout, out*}, a flag that indicates query operations, signal receptions for classes, State Diagrams[2], a formal definition of state configurations[3], and an extension of the formal descriptor of a class. Furthermore, the following information has to be added to system states to evaluate OCL expressions that make use of state-related and OCL message-related operations: for each object, the input queue of received signals and operation calls, the state configurations of all active objects, the currently executed operations, and for each currently executed operation, the messages sent so far. The resulting tuple of a *system state* over an extended object model $\mathcal{M}$ is

$$\begin{aligned}\sigma(\mathcal{M}) = \langle\ & \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \Sigma_{CONF},\\ & \Sigma_{currentOp}, \Sigma_{currentOpParam},\\ & \Sigma_{sentMsg}, \Sigma_{sentMsgParam},\\ & \Sigma_{inputQueue}, \Sigma_{inputQueueParam}\ \rangle\ .\end{aligned}$$

With those extensions, it is possible to define *execution traces* that capture all of those system changes that are relevant to evaluate OCL constraints [11]. In the simplest case, e.g., when (an implementation of) the system is executed on a single CPU, there is a clear temporal order of operations. But when (the implementation of) the system is distributed, we have a partial order between configurations of different objects. This problem can be treated in an ideal case by introducing a *global clock* that allows for a global view on the system. And additionally, we here assume that the time unit is chosen sufficiently small, such that only at most one OCL-relevant change per object may happen in a time step. This leads to a discretization of time.

Definition 3 *(Time-based Trace)*
A time-based trace *for an instantiation of an extended object model* $\mathcal{M}$ *is an (infinite) sequence of system states,* $trace(\mathcal{M}) \stackrel{def}{=} \langle\langle\ \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \ldots, \sigma(\mathcal{M})_{[i]}, \ldots\ \rangle\rangle$, *where each* $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, *represents the system state* i *time units after start of execution. In particular,* $\sigma(\mathcal{M})_{[0]}$ *denotes the initial system state.*

We may also apply the annotation $[i]$ to the components of the system state. In particular, we denote the state configuration of an active object $\underline{oid}$ over a system state $\sigma(\mathcal{M})_{[i]}$ by $s_{\underline{oid}[i]}$.

We here give an interpretation for the past temporal operation `pre(a,b)`, while the semantics of the future-oriented operation `post(a,b)` can be found in [13]. Assume that a temporal OCL expression for an object $\underline{oid}$ of a class $c \in CLASS$ is to be evaluated over a system state $\sigma(\mathcal{M})_{[t]}$ at time t of a trace $trace(\mathcal{M})$. The semantics of operation `pre(a,b)` is then defined as follows.[4]

$$\begin{aligned}& I[[pre : OclAny \times Integer \times OclAny\\ &\qquad \rightarrow Sequence(Set(OclState))]](\underline{oid}, a, b)\\ &\stackrel{def}{=} \begin{cases} \langle\langle s_{\underline{oid}\ [t-b]}, \ldots, s_{\underline{oid}\ [t-a]}\rangle\rangle, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c}\\ & \wedge\ a \geq 0\ \wedge\ b \geq a\\ & \wedge\ t - b \geq 0,\\ \bot, & \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c}\\ & \vee\ a < 0\ \vee\ a = \bot\\ & \vee\ b < a\ \vee\ b = \bot\ . \end{cases}\end{aligned}$$

[2]Note that no specific execution semantics for state diagrams have to be assumed here.

[3]UML only informally defines *active state configurations*. This results in some shortcomings, e.g., it is not considered that final states can be part of state configurations.

[4]For the matter of brevity, we omitted the additional variable assignment β in this definition. Function β determines values for OCL-specific variables, such as iterator variables and local variables of so-called `let`-expressions [19, Section A.3.1.2].

Symbol $\perp$ represents the predefined OCL value `OclUndefined`, i.e., a third logical built-in value of OCL that is used to indicate erroneous expressions. $\Sigma_{ACTIVE,c} \subseteq \Sigma_{CLASS}$ is the set of all currently existing objects of a so-called *active* class c – we have to consider here that only such kinds of objects have a notion of state configurations. Recall that b is either a non-negative natural number or `'inf'`. We interpret `'inf'` in this context as ∞. For the symbol ∞, it holds that $\forall i \in \mathbb{N}_0 : i < \infty \ \wedge \ i + \infty = \infty \ \wedge \ i - \infty = 0$.

5.2 Trace Literal Expressions

As we want to reason about time-based traces obtained by `@pre`, we need a new mechanism in OCL to explicitly specify traces with annotated timing intervals by means of literals. The timing intervals denote for how long each state configuration may be activated. Based on the OCL 2.0 metamodel, we define stereotypes `TraceLiteralExp` and `TraceLiteralPart` as illustrated in Figure 4. The following restrictions apply, leaving out the corresponding formal well-formedness rules for reasons of brevity.

1. The collection kind of stereotype `TraceLiteralExp` is `CollectionKind::Sequence`.

2. The type associated with a `TraceLiteralPart` must be `Set(OclState)`. Note that we do not require explicit specification of a set when a state configuration can already be specified by one state only. In this case, type `OclState` is implicitly casted to `Set(OclState)`.

3. Each `TraceLiteralPart` has a lower bound and an upper bound.

4. Lower bounds must evaluate to non-negative Integer values.

5. Upper bounds must evaluate to non-negative Integer values or to the String `'inf'` (for *infinity*). In the first case, the upper bound value must be greater or equal to the corresponding lower bound value.

Similar to the grammar rule for `PastTemporalExpCS`, only some additional grammar rules have to be added to the concrete OCL 2.0 syntax, such that modelers can specify trace literal expressions with timing bounds in OCL.

Finally, we define a new boolean operation on sequences called `includesSequence(seq:Sequence(T))`. Basically, this operation is a more general form of the already existing OCL collection operation `subSequence()`. This new operation returns true if the argument `seq` is included in the sequence to which the operation is applied. The abstract parameter type `T` is a placeholder for the element type of `seq`. It is required that this type must conform

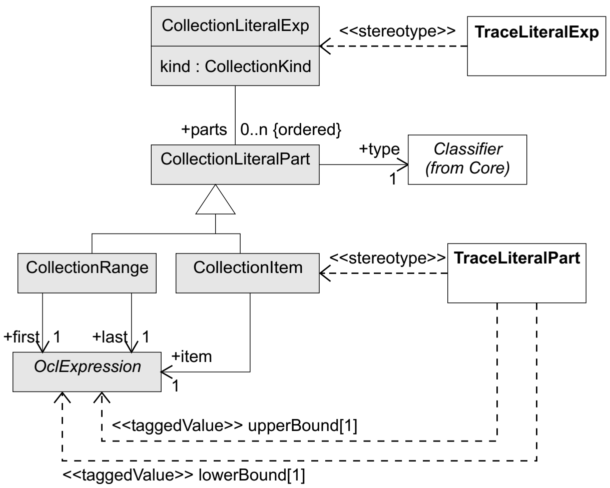

Figure 4. UML Stereotypes for Trace Literal Expressions

to the element type of the sequence to which the operation `includesSequence()` is applied. In particular, this allows to investigate whether a required sequence of state configurations (that is specified by means of a trace literal expression) has appeared in a trace. An example is given in the next section.

5.3 Mapping to Clocked LTL

Due to space limitations, we here focus on the mapping of instances of `PastTemporalExpCS` to Clocked LTL formulae. However, a corresponding mapping of future-oriented temporal OCL expressions can easily be obtained a very similar way.

By definition, OCL invariants for a given class must be true for all its instances at any time [19, Section 7.3.3]. In the context of time-based traces, this means that the invariant must be true on all traces at each position. Consequently, a corresponding Clocked LTL formula ϕ must hold for all traces of the model, i.e., $\mathcal{K} \models_A \phi$, and ϕ has to start with the `G` operator (globally).

Table 2 lists the main predefined OCL collection operations that can be directly applied to past time temporal OCL expressions. In each case, we give a mapping to corresponding Clocked LTL expressions. In that table, `expr` denotes a Boolean OCL expression. `cltlExpr` is the equivalent Boolean expression in Clocked LTL syntax.[5] `cfg` denotes a state configuration of a UML State Diagram (i.e., a set of activated states) and `cltlCfg` is the corresponding set of states in Clocked LTL syntax. `c` is an iterator variable for state configurations.

[5]We here assume that there is a mapping available from UML State Diagram states to the states of a Kripke Structure $\mathcal{K}$.

Table 2. Mapping Past Temporal OCL Expressions to Clocked LTL Formulae (at time t)

Temporal OCL Expression	Clocked LTL Formula
obj@pre(a,b)→includes(cfg)	$v \models_t \mathrm{F}_{\mathrm{past}[a,b]}(cltlCfg)$
obj@pre(a,b)→excludes(cfg)	$v \models_t \neg\mathrm{F}_{\mathrm{past}[a,b]}(cltlCfg)$
obj@pre(a,b)→exists(c | expr)	$v \models_t \mathrm{F}_{\mathrm{past}[a,b]}(cltlExpr)$
obj@pre(a,b)→forAll(c | expr)	$v \models_t \mathrm{G}_{\mathrm{past}[a,b]}(cltlExpr)$
obj@pre(a,b)→at(i:Integer | expr)	$v \models_t \mathrm{P}_{[b-i]}(cltlExpr)$

Let $e_1, e_2, \ldots, e_n$ be the parts of a trace literal expression with timing intervals $[a_i, b_i]$, $1 \leq i \leq n-1$. The past temporal OCL expression

```
obj@pre(a,b)->includesSequence(
            Sequence{e1[a1,b1], e2[a2,b2], ..., en})
```

maps to Clocked LTL as follows :

$$\mathrm{F}_{\mathrm{past}[a,b]}(e_1\,\mathrm{U}_{[a_1,b_1]}\,(e_2\,\mathrm{U}_{[a_2,b_2]}\,(\ldots\,(e_{n-1}\mathrm{U}_{[a_{n-1},b_{n-1}]}e_n)\ldots))).$$

Though we have presented the mapping by some examples here, it should be clear that more complex formulae are easily combined from the above, in particular with the logical OCL and Clocked LTL connectives `and`, `or`, and `implies`.

Note that we only investigate models with ‘persistent’ active objects, i.e., objects must exist from the initial system state onwards for the complete execution time. This is due to the formal model of Kripke structures and Clocked LTL formulae that do not support specification means for dynamic object creation and deletion. While this is sufficient for our particular application domain, this limitation should be overcome in the future for the benefit of a more general application. As a next step, we therefore intend to extend Kripke Structures by additional components and introduce new modalities to Clocked LTL.

6 Temporal OCL and Clocked LTL Examples

In this section we show some past-oriented temporal OCL constraints for requirements in the context of the buffer example presented in Section 2.

Figure 5 illustrates the behavior of the buffer by means of a State Diagram that is associated with the active class `Buffer`. The State Diagram comprises two orthogonal sections that work concurrently. One section is for taking items

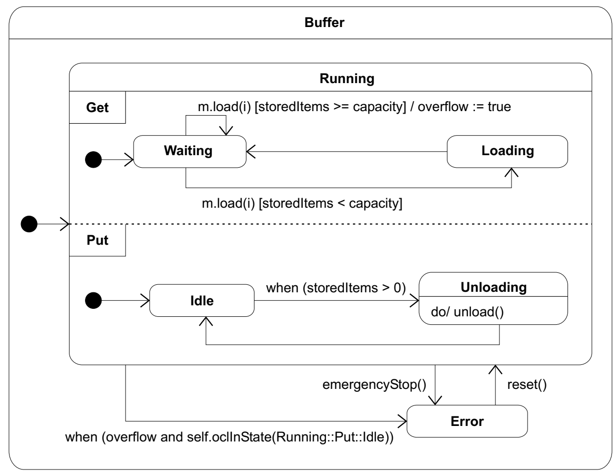

Figure 5. UML State Diagram for class Buffer

from the machines, the other section is for delivering items to the packaging unit. We do not show the State Diagrams of the remaining active classes for the sake of brevity, but note that they are modeled in a very similar way. For example, the State Diagram for the class `Packaging` comprises the simple states `Waiting`, `Loading`, `Maintaining`, and `Error`.

We require that every overflow in the buffer is due to an error in the packaging unit (i.e., a causality w.r.t. the past). This guarantees that the packaging unit is working sufficiently fast under usual conditions. Or in other words, the maintenance times do not interrupt the packaging unit for too long w.r.t. the speed of the machine outputs.

```
context Buffer
inv: self.oclInState(Error) implies
                       packaging@pre()->includes(Error)
```

This requirement can even be strengthened by an additional timing bound, e.g., `packaging@pre(1,10)`. Recall here that there has to be a time-based semantics for the execution of State Diagrams, which is not in the scope of standard UML.

Assuming that there is a mapping of such a time-based semantics to discrete-time Kripke Structures (cf. Definition 2), the following corresponding CLTL formula must hold:

$$\begin{aligned} &\mathrm{G}\ (\ (\text{buffer.state} = \text{buffer.error}) \\ &\quad \rightarrow \mathrm{F}_{\text{past}[1,\infty]}(\text{packaging.state} = \text{packaging.error})\ ) \end{aligned}$$

Although it is also possible to specify this requirement with future-oriented temporal OCL or LTL operators, the presented solution is a much more natural way of specifying a past-oriented causality.

7 Related Work

Temporal OCL extensions have already been proposed by other authors. After early approaches that directly add temporal logic formulas to OCL [23], more elaborated works consider future and/or past temporal operations [30, 4, 1]. However, all of these works do not consider timing bounds.

Some approaches already include timing bounds for property specifications, but they either use completely different notations [25] or introduce time-bounded OCL operations for *event*-based specifications [2]. We refer to [10] for a detailed discussion of temporal OCL extensions.

In contrast, we present a *consistent* OCL extension that reuses OCL language concepts like predefined collection types and corresponding operations. Additionally, we build upon the semantics adopted in the OCL 2.0 specification. In our work, we focus on *state-oriented* temporal OCL expressions rather than event-based specifications. Note here that the OCL standard already considers states of UML State Diagrams, as it is possible to check for activated states with the predefined operation `oclInState(stateName:OclState)`. However, some effort is needed to semantically integrate State Diagram states with the underlying formal model of OCL as explained in Section 5.1.

8 Conclusion

Together with our previous work, we now have an OCL extension that allows for the specification of past- and future-oriented state-oriented time-bounded constraints on the basis of the latest OCL 2.0 metamodel proposal. Our approach is still the only one that extends OCL by using the UML extension mechanism of profiles, i.e., stereotypes, tagged values, and constraints. The approach demonstrates that an OCL extension by means of a UML Profile towards temporal time-bounded constraints can be seamlessly applied on the abstract syntax layer M2. Nevertheless, extensions have to be made on the M1 layer as well in order to enable modelers to use OCL extensions like the temporal one we have proposed here.

We applied our temporal OCL extensions in the domain of modeling production automation systems and presented a UML Profile for a corresponding notation called MFERT in [12]. A semantics is given to the MFERT Profile by a mapping to synchronous time-annotated finite state machines (extended Interval Structures [28]). Our temporal OCL expressions then have a semantics for MFERT models, as their mapping to Clocked LTL formulae automatically establishes a formal relation between the two parts. This provides a sound basis for formal verification by Real-Time Model Checking. In this context, the RAVEN model checker has already been used to investigate finite Clocked LTL formulae in a simulation-based verification approach [26].

Concerning future research, we want to develop further temporal OCL extensions on arbitrary objects, as temporal requirements over *passive objects* cannot yet be expressed with our approach. Such requirements can only indirectly be specified through the different states of associated active objects. We therefore want to extend OCL towards specification of temporal expressions also w.r.t. attribute values and established links between objects. It is then possible to specify temporal restrictions over active as well as passive objects.

For example, in the context of the buffer example one might want to require that each item must not remain in the buffer for more than 120 time units. A possible solution would be to require for each `Item` object that the association `self.currentUnit` is of type `Buffer` for not more than 120 time units. A corresponding temporal OCL expression could be

```
context Item inv:
  self.currentUnit.oclIsTypeOf(Buffer)
  implies
  self.currentUnit@post(1,120).oclIsTypeOf(Packaging)
```

References

[1] J. Bradfield, J. Küster Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France*, volume 2306 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2002.

[2] M. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 389–408. Springer, July 2002.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT PRESS, 1999.

[4] S. Conrad and K. Turowski. Temporal OCL: Meeting Specifications Demands for Business Components. In *Unified*

Modeling Language: Systems Analysis, Design, and Development Issues. IDEA Group Publishing, 2001.

[5] W. Damm and J. Klose. Verification of a Radio-based Signaling System Using the Statemate Verification Environment. *Formal Methods of System Design*, 19(2):121–141, 2001.

[6] B. Douglass. *Doing Hard Time: Developing Real Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 2000.

[7] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. In *21st International Conference on Software Engineering, Los Angeles, California*, May 1999.

[8] E. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier Science Publisher, Amsterdam, 1990.

[9] T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz. Timed Sequence Diagrams and Tool-Based Analysis – A Case Study. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 1999*, volume 1723 of *Lecture Notes in Computer Science*, pages 645–660. Springer, 1999.

[10] S. Flake. Temporal OCL Extensions for Specification of Real-Time Constraints. In S. Graf, O. Haugen, I. Ober, and B. Selic, editors, *UML 2003 Workshop "Specification and Validation of UML models for Real Time and Embedded Systems" (SVERTS'03), San Francisco, CA, USA*, 2003.

[11] S. Flake. Towards the Completion of the Formal Semantics of OCL 2.0. In *27th Australasian Computer Science Conference (ACSC 2004), Dunedin, New Zealand, January 2004*, volume 26 of *Australian Computer Science Communications*, pages 73–82. Australian Computer Science Society, Sydney, Australia, 2004.

[12] S. Flake and W. Mueller. A UML Profile for Real-Time Constraints with the OCL. In *UML 2002 - The Unified Modeling Language. 5th International Conference, Dresden, Germany*, volume 2460 of *LNCS*, pages 179–195. Springer, 2002.

[13] S. Flake and W. Müller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Software and Systems Modeling (SoSyM), Springer*, 2(3):164–186, October 2003.

[14] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the Temporal Analysis of Fairness. In *7th ASM Symposium on Principles of Programming Languages (POPL'80)*, pages 163–173. ACM Press, January 1980.

[15] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK, 2000.

[16] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. In *7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002*, Lecture Notes in Computer Science. Springer, 2002.

[17] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 1977.

[18] O. Lichtenstein, A. Pnueli, and L. Zuck. The Glory of the Past. In R. Parikh, editor, *Conference on Logic of Programs, Brooklyn, NY, June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[19] OMG, Object Management Group. UML 2.0 OCL Final Adopted Specification. OMG Document ptc/03-10-14, October 2003. ftp://ftp.omg.org/pub/docs/ptc/03-10-14.pdf.

[20] OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003.

[21] A. Pnueli. A Temporal Logic of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1980.

[22] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical Model Checking of LTL with Past. In *1st Int. Workshop on Automated Technology for Verification and Analysis*, National Taiwan University, Taipei, Taiwan, December 2003.

[23] S. Ramakrishnan and J. McGregor. Extending OCL to Support Temporal Operators. In *Proc. of the 21st International Conference on Software Engineering (ICSE99), Workshop on Testing Distributed Component-Based Systems*, Los Angeles, May 1999.

[24] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Bremen, Germany, 2001.

[25] E. Roubtsova, J. van Katwijk, W. Toetenel, and R. de Rooij. Real-Time Systems: Specification of Properties in UML. In *ASCI 2001 Conference*, pages 188–195, Het Heijderbos, Heijen, The Netherlands, May 2001.

[26] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-Guided Property Checking Based on Multivalued AR-Automata. In *Design, Automation and Test in Europe (DATE'01), Munich, Germany*, pages 742–748. IEEE Computer Society Press, 2001.

[27] J. Ruf and T. Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME97)*, pages 146–166, Montreal, Canada, October 1997. IFIP WG 10.5, Chapman and Hall.

[28] J. Ruf and T. Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Correct Hardware Design and Verification Methods (CHARME 99)*, pages 265–279. IFIP WG 10.5, Springer, September 1999.

[29] Y. Zhang. A Foundation for the Design and Analysis of Robotic Systems and Behaviors. Technical Report 94-26, Department of Computer Science, University of British Columbia, Vancouver, Canada, 1994. PhD Thesis.

[30] P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML*, pages 53–62. Technische Universität München, Institut für Informatik, Munich, Germany, 2002.