Specification of Real-Time Properties for UML Models

Stephan Flake and Wolfgang Mueller
C-LAB, Paderborn University
Fuerstenallee 11, 33102 Paderborn, Germany
{*flake, wolfgang*}*@c-lab.de*

Abstract

The Unified Modeling Language (UML) has received wide acceptance as a standard language in the field of software specification by means of different diagram types. In a recent version of UML, the textual Object Constraint Language (OCL) was introduced to support specification of constraints for UML models. But OCL currently does not provide sufficient means to specify constraints over the dynamic behavior of a model.

This article presents an OCL extension that is consistent with current OCL and enables modelers to specify state-related time-bounded constraints. We consider the case study of a flexible manufacturing system and identify typical real-time constraints. The constraints are presented in our temporal OCL extension as well as in temporal logic formulae. For general application, we define a semantics of our OCL extension by means of a time-bounded temporal logic based on Computational Tree Logic (CTL).

1. Introduction

Formal verification methods like equivalence and model checking have been well accepted through past years for different kinds of applications. In particular, model checking has received a wide industrial acceptance for electronic system and protocol verification. Model checking needs a system description and an additional property specification as input. Properties are typically specified by formulae in temporal logics, mostly in a future-oriented branching-time logic called Computational Tree Logic (CTL). Though already frequently applied, it often turns out that modelers and programmers are not familiar with formal methods and regard it as a task too cumbersome to specify and understand such properties in temporal logics.

On the other hand, the Unified Modeling Language (UML) is well accepted in research and industry for a wide spectrum of applications. With the wide acceptance of UML, the Object Constraint Language (OCL) has also received a considerable visibility. OCL provides means for the specification of constraints in the context of UML, focusing on class diagrams and on guards in behavioral diagrams, but it currently lacks sufficient means to specify constraints over the dynamic behavior of such diagrams, i.e., the evolution of states and state transitions as well as timing constraints. However, it is essential to be able to specify such constraints for real-time systems to guarantee correct system behavior.

We present an OCL extension that overcomes this limitation and keeps compliant with the syntax and semantics of the current version of OCL (Version 1.4). Though we present our approach as a constraint specification over the state space of UML Statechart diagrams, it is also well applicable for other state-oriented means like activity diagrams. With this approach, it is possible to replace cryptic CTL specifications by more meaningful extended OCL specifications, which are better tailored to the mental model of programmers. Our approach introduces new OCL types for certain state collections and operations for their manipulation. We see this OCL extension as a real improvement towards the specification of general real-time systems.

The remainder of this article is structured as follows. In the next section, we give a brief overview of related works w.r.t. formal verification and OCL extensions. Section 3 presents a manufacturing case study with automated guided vehicles. It is used as a running example throughout this article. Section 4 briefly covers real-time model checking with an introduction to modeling with I/O-interval structures and property specification with time-bounded CTL. Section 5 regards UML with an emphasis on Statechart diagrams and the concepts of current OCL. Section 6 introduces our OCL extensions and provides a semantics based on the temporal logic of time-bounded CTL. Section 7 briefly outlines the current state of our implementation, before Section 8 summarizes and concludes this paper.

2. Related work

There currently exist only a few approaches that apply OCL in the context of formal verification. The KeY project aims to facilitate the use of formal verification for software specifications [1]. As OCL currently has no formal semantics, this approach translates OCL constraints to dynamic logic (DL), an extension of Hoare logic. DL is used as input for formal verification. In this approach, OCL is applied without modifications to specify constraints on design patterns.

Two other approaches consider temporal extensions for OCL. Distefano et al. [3] define BOTL (Object-Based Temporal Logic) in order to facilitate the specification of static and dynamic properties. BOTL is based on a combination of CTL and a subset of OCL. Syntactically, BOTL looks very similar to the common formulae in CTL. Another temporal extension of OCL is defined by Ramakrishnan et al. [6, 7]. They extend OCL by additional rules with unary and binary temporal operators, e.g., `always` and `never`. Unfortunately, the resulting syntax does not combine well with current OCL concepts. Again, temporal expressions appear to be similar to temporal logic formulae.

In contrast to these approaches, we introduce extensions to OCL that concern the dynamic behavior of UML models. The extensions are performed with only minor modifications on the language metalevel, so that the use of current OCL is not affected in any way. In order to seamlessly integrate into existing OCL, our work is based on an OCL metamodel presented by Baar and Hähnle in [2]. We mainly have selected their model since it clearly separates metalevel and instance level for OCL's root metaclass `OclType`. However, we think that an adaptation to another OCL metamodel such as [8] is possible without any problems. Our extensions provide a seamless integration for the specification of state-oriented constraints, as they are required when OCL is used in combination with behavioral UML diagrams like Statecharts. Though our extensions are kept compliant with OCL syntax, they have a direct correspondence to temporal tree logic formulae for easy code generation in a formal verification framework. Moreover, our work also covers the specification of real-time constraints, as we apply Clocked CTL (CCTL), a time-oriented extension of CTL [10].

3. Flexible manufacturing case study

We apply the Holonic Manufacturing System (HMS) case study as a running example throughout the following sections. The HMS case study was introduced by the IMS Initiative [14]. It is composed of a set of different manufacturing stations and a transport system as it is illustrated by the virtual 3D model in Figure 1.

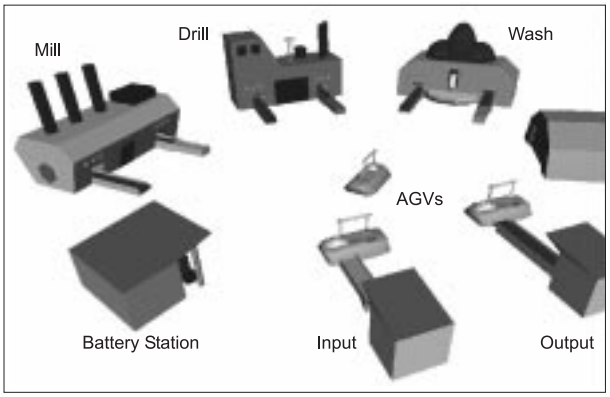

Figure 1. 3D Model of the Case Study

The different manufacturing stations transform items, e.g., by milling, drilling, or washing. Additional input and output storages are for primary system input and output. The transport system consists of a set of automated guided vehicles (AGVs), i.e., autonomous vehicles that carry items between stations. We assume that stations have an input buffer for incoming items and that each AGV can take only one item at a time. The whole system is basically characterized by the following application flow.
Once having located an item at its output buffer, a station

1. broadcasts a request for delivery to all AGVs
2. receives replies from each idle AGV h_i
3. selects one AGV h_i
4. notifies AGV h_i for its acceptance, and notifies all other AGVs for their rejection.

On the other hand, each AGV h_i

1. is idle until it receives a request for delivery from a station s_j
2. sends a reply to s_j on request of s_j
3. moves to s_j on notification of acceptance from s_j
4. takes an item from the output buffer of s_j
5. asks the next destination station s_k ($s_k \neq s_j$) for permission to deliver the item
6. moves to s_k on notification of acceptance from s_k
7. unloads the item at the input buffer of s_k
8. moves to a parking position and returns to step 1.

4. Real-time model checking with RAVEN

In this section, we outline a temporal logic that is used in our OCL extension to provide a formal semantics. This logic is introduced for formal verification with the RAVEN model checker. In RAVEN, a model is given by a time-annotated state transition system, i.e., a set of I/O-interval

Table 1. Semi-formal Description of CCTL Operators

Formula	Denotation	Description
$g_0 \models p$ ($p \in P$)	Proposition	g_0 is valid in p, if $p \in L(s_0)$
$g_0 \models \neg\phi$	Negation	g_0 is satisfied by $\neg\phi$ if $g_0 \models \phi$ is false.
$g_0 \models (\phi \wedge \psi)$	Concatenation	$g_0 \models \phi$ and $g_0 \models \psi$
$g_0 \models \mathbf{EX}_{[a]}\ \phi$	Next	There exists a run $r = (g_0, \ldots)$ such that $g_a \models \phi$
$g_0 \models \mathbf{EF}_{[a,b]}\ \phi$	Eventually	There exists a run $r = (g_0, \ldots)$ and $a \leq i \leq b$ s.t. $g_i \models \phi$
$g_0 \models \mathbf{EG}_{[a,b]}\ \phi$	Globally	There exists a run $r = (g_0, \ldots)$ s.t. for all $a \leq i \leq b$ holds $g_i \models \phi$
$g_0 \models \mathbf{E}(\phi\ \underline{\mathbf{U}}_{[a,b]}\ \psi)$	Strong Until	There exists a run $r = (g_0, \ldots)$ and an $a \leq i \leq b$ s.t. $g_i \models \psi$ and for all $j < i$ holds $g_j \models \phi$
$g_0 \models \mathbf{E}(\phi\ \mathbf{U}_{[a,b]}\ \psi)$	Weak Until	There exists a run $r = (g_0, \ldots)$ and and either (a) there exists an $a \leq i \leq b$ s.t. $g_i \models \psi$ and for all $j < i$ holds $g_j \models \phi$, or (b) for all $i \leq b$ holds $g_i \models \phi$

structures [9]. Interval structures are based on Kripke structures with [min,max]-time intervals at their state transitions. We here only briefly sketch the basics of interval structures. For a more detailed introduction to I/O-interval structures, the reader is referred to [10].

An interval structure $\Im$ is a tuple (P, S, T, L, I) with a set of propositions P, a set of states S, a transition relation T between states such that every state has a successor state, a state labeling function $L : S \rightarrow \wp(P)$, and a transition labeling function $I : T \rightarrow \wp(\mathbb{N})$. I/O-interval structures are interval structures that have read-only access to states of other interval structures. We assume that each interval structure has exactly one clock for measuring time. The clock is reset to zero, if a new state is entered. A state may be left, if the actual clock value corresponds to a delay time labeled at an outgoing transition. The state must be left where the maximal delay time of all outgoing transitions is reached. A *clocked state*[1] $g = (s, v)$ of an interval structure $\Im$ is a state s associated with a clock value v. The set of all valid clocked states in $\Im$ is called G.

Clocked CTL (CCTL) is a time-bounded temporal logic [11]. In contrast to classical CTL, the temporal operators **F** (i.e., eventually), **G** (globally), and **U** (until) are provided with interval time-bounds $[a, b]$, $a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\}$. The symbol ∞ is defined through: $\forall i \in \mathbb{N}_0 : i < \infty$. These temporal operators can also have a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is zero and the upper bound is infinity by default. The **X**-operator (i.e., next) can have a single time-bound $[a]$ only ($a \in \mathbb{N}$). If no time bound is specified, it is implicitly set to one.

The semantics of CCTL is defined as a validation relation "$\models$", using the notion of *runs*, which represent possible sequences of clocked states that occur during execution of $\Im$. Any arbitrary clocked state g_0 may be the starting point of a run. Table 1 shows some sample semi-formal descriptions of the validation relation for a given interval structure $\Im$ and a clocked state $g_0 = (s_0, v_0) \in G$. Note that ϕ and ψ denote arbitrary CCTL (sub)formulae.

The semantics for temporal operators with path quantifier **A** (i.e., regarding *all* possible runs) can easily be derived, e.g., $\mathbf{AX}_{[a]}\phi$ is equivalent to $\neg\mathbf{EX}_{[a]}\neg\phi$. Another example is $\mathbf{AF}_{[a,b]}\phi$, which is equivalent to $\neg\mathbf{EG}_{[a,b]}\phi$.

In the context of RAVEN, I/O-interval structures and a set of CCTL formulae are specified by means of the textual RAVEN Input Language (RIL). A RIL specification contains (a) a set of global definitions, e.g., fixed time bounds or frequently used formulae, (b) the specification of parallel running modules, i.e., a textual specification of I/O-interval structures, and (c) a set of CCTL formulae, representing required properties of the model. The following code is a fragment of the RIL model for our running example, regarding a part of a station input buffer.

```
MODULE acceptor
SIGNAL
  state : {waitingForOrder,rejecting,accepting,failed}
INPUTS announced := global_newOrderForMachine
       loadFail  := global_loadFailure
       idle      := (loader.state = loader.idle)
DEFINE rejectOrder := (state = rejecting)
       acceptOrder := (state = accepting)
INIT   state = waitingForOrder
TRANS
|- state=waitingForOrder
  --  loadFail                    --> state:=failed
  -- !loadFail &  idle & announced --> state:=accepting
  -- !loadFail & !idle & announced --> state:=rejecting
                              !-> state:=waitingForOrder
|- state=rejecting
  -- loadFail             --> state:=failed
                          !-> state:=waitingForDelivery
... // some more transitions omitted
```

[1] Clocked states are originally called *configurations*, but we are going to use this term in a different context in the following sections.

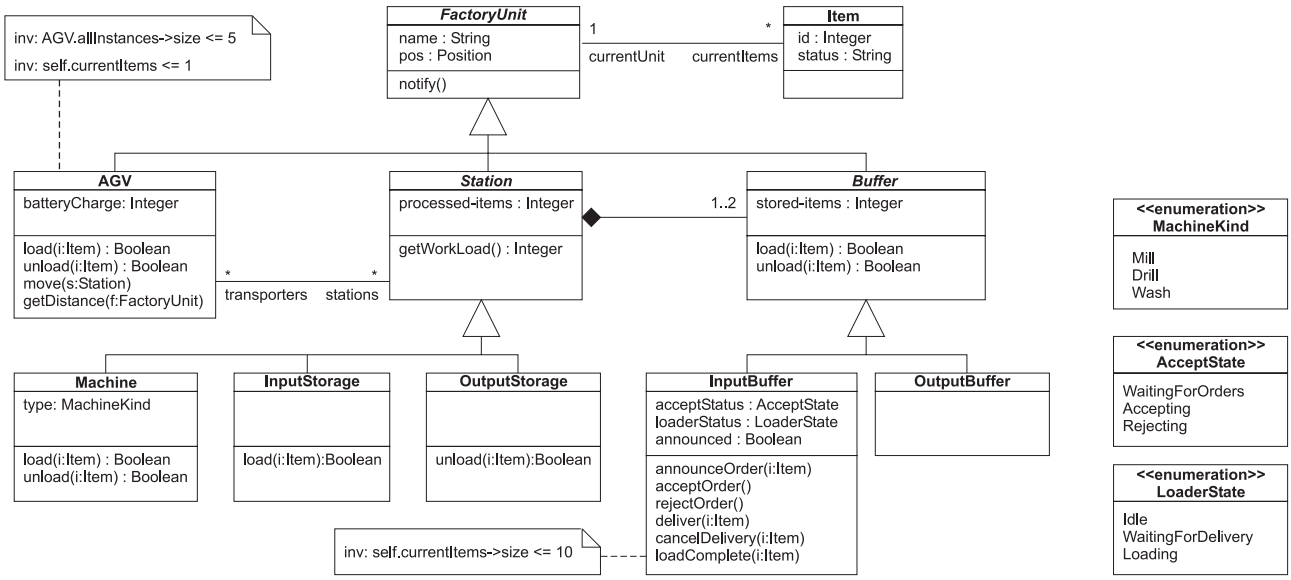

Figure 2. Class Diagram of the Case Study

For property specification, consider the following example. One requirement in our case study is that the input buffer of a station must not be blocked for too long in order to guarantee sufficient continuous workload, i.e., each accepted delivery request must be followed by actually loading an item at the input buffer within 100 time units after acceptance. Due to the dependency on other modules, in particular the AGVs, it is not obvious whether the model satisfies this property. Therefore, a corresponding CCTL formula has to be specified:

```
AG((acceptor.state = acceptor.accepting)
  -> AF[100]((loader.state = loader.waitingForDelivery)
             & AX(loader.state = loader.loading)
            )
  )
```

If RAVEN evaluates a CCTL formula to be incorrect, a counter example execution run can be generated. Execution runs are given by time-annotated sequences of state changes. RAVEN invokes a built-in waveform browser that lists all variables and their states over time.

5. UML

UML (Unified Modeling Language) is a widely accepted OMG standard for graphical design capture and representation. UML has a very rich notational framework which is deeply embedded in and tied to object oriented methodology. UML is most useful for communication amongst designers and design teams to understand and explain designers' intent. UML expresses models through a rich set of diagrams, i.e., class, package, deployment, use case, collaboration, sequence, activity, and state diagrams.

Class diagrams describe the static structure of a system. As an example, Figure 2 gives a class diagram for the previously introduced case study. Classes are given by rectangular boxes with variables and operations in their lower section. Generalizations are given as vertices with white triangles, while diamonds represent aggregation relationships, and simple vertices denote associations. OCL constraints, in particular invariants, are associated by a dotted line with the corresponding class.

Figure 3 illustrates examples for behavior-oriented UML diagrams: Statechart diagrams, activity diagrams, and sequence diagrams.

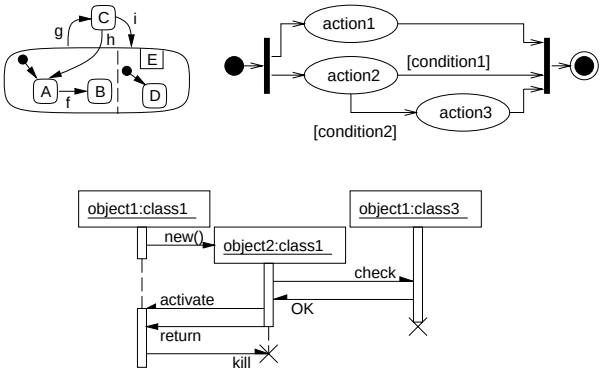

Figure 3. Sample UML Diagrams: Statechart, Activity Diagram, Sequence Diagram

Figure 4 gives the corresponding Statechart of a subbehavior of a station input buffer. The figure shows one superstate (`InputBuffer`) with two concurrent substate definitions (`Acceptor` and `Loader`) where the black circles denote the initial states. Directed vertices define state transitions and are annotated by conditions.

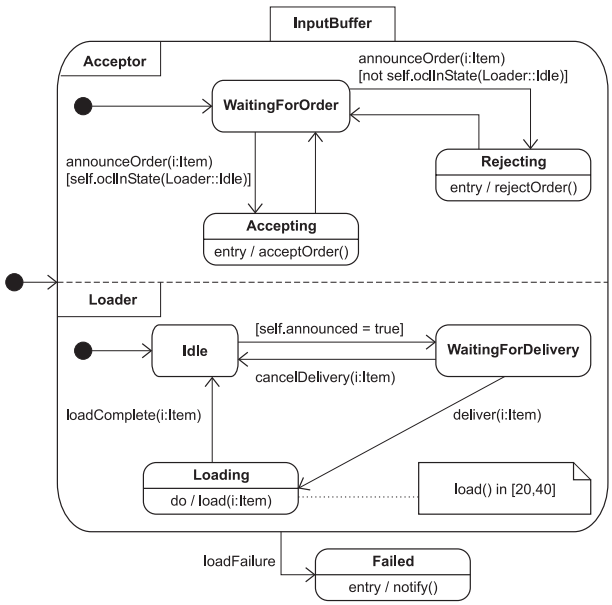

Figure 4. Statechart Diagram: Input Buffer

Based on the case study presented in Section 3, we assume that AGVs, stations, and the input and output storages are all modeled by UML class diagrams and that their behavior is given by Statecharts. We focus here on the subaspects of the behavioral specification of an input buffer that is in charge of delivery requests. Thus, the corresponding Statechart is separated into two parallel substates, one for handling messages from other stations which request a notification acceptance for delivery (`Acceptor`), the other one for performing the actual loading process after the acceptance of a delivery (`Loader`), as it is shown in Figure 4.

To model behavior over time, we are using text annotations. In our example, there is a time interval assigned to state `Loading`; ”load() in [20,40]” specifies that loading takes between 20 to 40 time units. If the buffer fails for some reason, e.g., a sensor is sending a failure signal, the buffer enters a failure state, notifies the AGVs and other stations, and gives an error report.

Object Constraint Language. The Object Constraint Language (OCL) is part of the UML since Version 1.3. It is a language to express restrictions on a system under development and is applied as textual annotations within the different UML diagram types. With OCL, modelers can express invariants on classes as well as pre- and postconditions for operations. Boolean OCL expressions can be applied in behavioral diagrams (i.e., Statecharts and activity diagrams) as transition conditions. OCL has a simple nonsymbolic syntax and claims to be precise and unambiguous, but still easy understandable by designers in the area of object-oriented technology [13]. OCL has a number of core concepts, e.g., it is declarative without side effects and has a set of predefined built-in types dedicated to deal with object collections. As an example, consider class InputBuffer in Figure 2. Assume that technical constraints require that the input buffer cannot keep more than 10 items at a time. Consequently, the number of items in the input buffer has to be restricted by the following OCL constraint:

```
context InputBuffer inv:
  self.currentItems->size <= 10
```

We briefly explain how to read this invariant. The dot ”.” is used to access properties of an object. In this example, it is used to navigate within the class diagram and yield those objects associated to the object on the left via the association name on the right. In this case, we retrieve the set of all instances of class Item currently associated to InputBuffer. The arrow ”→” indicates that the expression to its left represents a collection of objects. OCL distinguishes between three kinds of collections: sets, multisets resp. bags, and sequences. The operation to the right of the arrow is applied to this collection. In our example, operation `size()` returns the number of elements of the previously determined set of items.

6. Real-time OCL extensions

Our OCL extensions introduce temporal OCL operations for state-oriented behavior. For a seemless integration into the concepts of existing OCL, we consider the OCL type metamodel of Baar and Hähnle [2], introduce some new operations for `OclState` and `OclAny`, and add two new basic types `OclConfiguration` and `OclPath` to their metamodel. In the following, we only outline the basic concepts of our extensions, as we focus in this article on the application of the OCL extensions to specify typical real-time constraints. More details about our metamodel extension and the new operations can be found in [4].

6.1. States, configurations, and paths

Current OCL already supports the retrieval of states from Statechart diagrams. States are regarded to be of type `OclState`. However, this type is only marginally investigated in the OCL standard and thus needs to be elaborated

with respect to its combined usage with UML Statechart diagrams and the underlying formal model of state machines[2]. We therefore introduce new properties for OclState and briefly describe their semantics.

```
OclBasicType OclState <supertype> OclAny {
  stateType       : enum{composite,simple};
  isConcurrent    : Boolean;
  isRegion        : Boolean;
  parentState  () : OclState;
  subStates    () : Set(OclState);
  isActive     () : Boolean;
  notActive    () : Boolean;
  anySubState  () : OclState;
}
```

In the following, s denotes an instance of type OclState. We first introduce an enumeration attribute stateType with literals composite and simple that indicates whether s contains substates or not. A boolean attribute isConcurrent indicates whether s contains concurrent substates (called *regions*); a boolean attribute isRegion checks whether s is a substate of a concurrent state. We define the operations parentState() and subStates() which return the direct parent state and the set of direct substates, respectively. isActive() evaluates to true if s is currently active and its dual notActive() becomes true if it is not active. We also introduce an operation anySubState() which returns a non-deterministically selected substate.

Compliant with common OCL practice[3], we take some implicit presumptions for the remainder of this article. We assume that there is at most one Statechart (resp. one state machine) associated to each class. In order to be directly accessible from OCL, all simple and composite states of a state machine have to be available as instances of OclState and their properties are set according to their state machine specification.

6.1.1. Configurations

Currently, the only possibility to retrieve information about states in OCL is given by the boolean operation oclInState() of type OclAny. This is not sufficient since in concurrent Statechart diagrams, an overall state can only be uniquely described by tuples of substates and thus needs additional operations on them. We refer to such a tuple as a *configuration*. More precisely, we consider a configuration as a set of simple states that uniquely and completely describe an overall state of a given Statechart diagram.

The concurrent Statechart shown in Figure 4 has the top level state InputBuffer, which also denotes the class this Statechart belongs to. The initial configuration is Set{Acceptor::WaitingForOrder, Loader::Idle}.

[2]see [5], Section 2.12

[3]see [5], Section 6.5.10: States are already directly accessible in OCL expressions.

Given this initial configuration, the expression oclInState(Acceptor::WaitingForOrder) is true in OCL, although Acceptor::WaitingForOrder is not a complete configuration of InputBuffer. When investigating complete configurations on the level of simple states, we currently have to write

```
InputBuffer.oclInState(Acceptor::WaitingForOrder)
              and InputBuffer.oclInState(Loader::Idle)
```

since only the operation oclInState() is available. It is easy to see that such specifications are not easily manageable for complex Statecharts. To overcome this, we introduce the new basic type OclConfiguration and extend OCL's root metatype OclAny with the operation config(). This operation returns the set of all possible valid configurations, i.e., all sets of *simple* states that are required to uniquely cover complete configurations. For instance, InputBuffer.config() returns a set of $3 * 3 = 9$ elements:

```
{ Acceptor::WaitingForOrder, Acceptor::Rejecting,
              Acceptor::Accepting }
                        ×
   { Loader::Idle, Loader::WaitingForDelivery,
                Loader::Loading }
```

With config(), it is possible to check configurations for their validity. For instance,

```
context InputBuffer inv:
  self.config()->includes(c:OclConfiguration |
          c = Set{Acceptor:Accepting,Loader::Loading})
```

checks if Set{Acceptor::Accepting,Loader::Loading} is a valid configuration for InputBuffer.

6.1.2. OclConfiguration operations

As we interpret OclConfiguration as a new built-in type for a representation of specific sets of OclStates, most operations from OCL's collection type Set can be reused. We only have to elaborate on operations returning collections since the result might not be a valid configuration, but an arbitrary set of OclStates. Therefore, we can only adopt the operations $=$, $<>$, size(), count(), isEmpty(), notEmpty(), exists(), forAll(), includes(), includesAll(), excludes(), excludesAll()[4]. The other OCL operations for collections, e.g., union() and intersection(), cannot be applied for OclConfigurations without modification of their semantics, as they usually result in arbitrary sets of states rather than valid configurations. Nevertheless, type cast operations, e.g., asSet(), still give access to the omitted operations.

[4]see [5], Section 6.8.2

Additionally, we introduce two new operations `isActive()` and `notActive()`. For an instance `cfg` of type `OclConfiguration`, `cfg->isActive()` returns true if all states of `cfg` are active, while `cfg->notActive()` is the boolean opposite to this. To access `OclConfiguration` properties, we make use of the arrow-operator. This solution is chosen to keep compliant with existing OCL and its syntax for collection operations, although we regard `OclConfiguration` as a new basic type.

6.1.3. OclPath

In order to reason about execution sequences of state machines, we require means to represent sequences of configurations. Our notion of "sequence" assumes strong successorship, i.e., no other configuration may occur in between two subsequent elements of a specified sequence. Additionally, a configuration in a sequence may hold for a certain time or a timing interval. In that case, the interval specification is appended to the expression as an additional qualifier.

Current OCL already covers sequence declarations through `literalCollection`, so that there is no need to add or modify OCL grammar rules with that respect. We illustrate the declaration of OclPaths by an example in the context of Figure 4. A sequence for state `InputBuffer` which directly changes from its initial configuration to `Set{Acceptor::Accepting, Loader::Idle}` after an arbitrary time and then immediately changes to the respective waiting states of both substates is specified by the following configuration sequence:

```
Sequence{ Set{Acceptor::WaitingForOrder,
              Loader::Idle              } [1,'inf'],
          Set{Acceptor::Accepting,
              Loader::Idle              } [1],
          Set{Acceptor::WaitingForOrder,
              Loader::WaitingForDelivery}
        }
```

6.1.4. OclPath operations

An instance of `OclPath` is interpreted as a possible execution sequence composed of OclConfigurations in the context of a given Statechart resp. state machine. Similar to `OclConfiguration`, many of the existing OCL sequence operations can be immediately applied to `OclPath`. These operations are $=$, $<>$, size(), isEmpty(), notEmpty(), exists(), forAll(), includes(), includesAll(), excludes(), excludesAll(), at(), first(), last(), append(), prepend(), subSequence(), asSet(), asBag(), and asSequence(). The semantics of almost all these operations can be directly derived from the generic OCL types `Collection` and `Sequence`[5].

We cannot make all common sequence operations directly available to `OclPath`, as they would result in arbitrary collections of OclConfigurations, which are not valid OclPaths, e.g., operations `select()` and `collect()` that extract certain elements of a sequence. Nevertheless, type casting operations still permit access to all common sequence operations.

[5] see [5], Section 6.8.2

6.2. Temporal operations

We introduce temporal operations to obtain object values with respect to certain points in time. Since our application domain is future-oriented branching time logic, we focus our definition only on future-oriented operations. However, we see no limitation to extend our work further to the specification of past-oriented constraints.

We first consider the `@pre` operator which is already available in OCL. This operator is only allowed in postconditions and used to recall the value of an object when an operation was started. Correspondingly, we define `@post` that regards to future points in time. For a seamless integration of that operator, we interpret the symbol @ as a separate operator such as the dot- and arrow-operator. This means to take `pre()` and `post()` as *operations* of `OclAny` and limit the @-operator to be used only for those temporal operations. With this interpretation, we only need very few minor changes w.r.t. the OCL grammar, so that syntax and semantics of existing OCL can be kept.

6.2.1. Temporal extensions to OclAny

For the introduction of temporal operations we extend OCL type `OclAny` as follows:

```
OclBasicType OclAny {
  -- standard OclAny operations are kept
  ...
  -- the following new operations are introduced:
  config () : Set(OclConfiguraion);
  pre    () : OclAny;
  post   () : Set(OclPath);
  next   () : Set(OclConfiguration);
}
```

We already introduced operation `config()` in Section 6.1.1; it returns a set of all possible configurations. In order to stick with the existing OCL return type of `pre()`, it has to be of type `OclAny`. We define `post()` as an operation that returns a set of OclPaths, i.e., a *set* of possible future execution sequences. It has to be a set, as there are typically different possible orders of executions in a Statechart. Operation `next()` returns a set of all possible next configurations. Furthermore, we allow the declaration of a [min,max]-interval in combination with `post()`, as already introduced for `OclPath`.

An informal semantics is given as follows. Let `obj` be an object in the context of OCL.

```
-------------------------------------------------------
obj@pre : OclAny
  This operation may be applied in operation postcondi-
  tions only. It returns the value of obj at the time
  of entering the respective operation.
-------------------------------------------------------
obj@post[a,b] : Set(OclPath)
  Returns a set of possible future execution sequences
  in the interval [a,b]. The configurations of time
  points a and b are included. Qualifier a must be of
  type Integer, and b must either be of type Integer or
  of type String (in the latter case, b must be equal
  to 'inf').
-------------------------------------------------------
obj@post[b] : Set(OclPath)
  Equal to obj@post[b,b]. b must be of type Integer
-------------------------------------------------------
obj@post : Set(OclPath)
  Equivalent to obj@post[1,'inf'].
-------------------------------------------------------
obj@next : Set(OclConfiguration)
  Similar to obj@post[1,1], but this operation returns
  a set of OclConfigurations which can be reached after
  the next time step.
-------------------------------------------------------
```

6.2.2. Mapping temporal OCL expressions

We formally define our temporal extensions by the means of CCTL formulae as they were introduced in Section 4. For OCL invariants, all corresponding CCTL formulae start with an AG operator, i.e., with 'always globally'. Table 2 lists OCL operations that directly match to CCTL expressions. In that table, expr is supposed to be of type OclExpression with expr.evaluationType() = Boolean, and cctlExpr is the equivalent boolean expression in CCTL syntax. It should be easy to see how more complex nested expressions correspond.

We finally demonstrate how configuration sequences correspond to CCTL formulae. Let $e_1, e_2, ..., e_n$ be elements of a sequence declaration where each e_i can be either a simple OclConfiguration or a complex expression, specified with timing intervals $[a_i, b_i]$. The temporal OCL expression

```
inv:  obj@post[a,b]→includes(
                    Sequence{e1[a1,b1], e2[a2,b2], ..., en})
```

maps to the CCTL formula

$$\mathtt{AG}_{[a,b]}\ \mathtt{EF(\ E(}e_1\ \underline{\mathtt{U}}_{[a_1,b_1]}\ \mathtt{E(}e_2\ \underline{\mathtt{U}}_{[a_2,b_2]}\ \mathtt{E(...\ E(}e_{n-1}\ \underline{\mathtt{U}}_{[a_{n-1},b_{n-1}]}\ e_n\mathtt{)...))))}$$

Note that the path quantifier, which is applied to each sequence element, depends on the preceded operations. Though we only give that single mapping as an example here, it should be clear how general expressions relate.

6.3. Constraints

In this section, we specify typical real-time constraints with respect to the manufacturing case study described in Section 3. The sample constraints presented here mainly concern the Statechart for InputBuffer. Each constraint is introduced in the following way: First, an informal requirement specification in natural language is given, then a respective CCTL formula is presented to illustrate a compact formal representation of the constraint, and finally an according OCL formula is given, applying our OCL extension with temporal operations.

To keep the constraint formulae short, we define some abbreviations for those subexpressions that are frequently applied. In RIL, the following definitions are declared with global visibility:

```
DEFINE               // RIL code for global definitions
waitForOrder   := (acceptor.state =
                            acceptor.waitingForOrder)
accepting      := (acceptor.state = acceptor.accepting)
rejecting      := (acceptor.state = acceptor.rejecting)
loaderIdle     := (loader.state   = loader.idle)
loaderWaiting  := (loader.state   =
                           loader.waitingForDelivery)
loaderLoading  := (loader.state   = loader.loading    )
```

In OCL, definitions for a given classifier are declared by let-expressions:

```
context InputBuffer  -- OCL code for global definitions
def: let waitForOrder   = Acceptor::WaitingForOrder
     let accepting      = Acceptor::Accepting
     let rejecting      = Acceptor::Rejecting
     let loaderIdle     = Loader::Idle
     let loaderWaiting  = Loader::WaitingForDelivery
     let loaderLoading  = Loader::Loading
```

The variables declared above are used in the following constraints:

1. We demand that items have to periodically arrive at the input buffer within timing intervals of at most 400 time units. In other words, state Loader::Loading can always be reached again within 400 time units. An according CCTL formula is

   ```
   AG EF[1,400] (loaderLoading)
   ```

   The corresponding OCL formula is

   ```
   context InputBuffer inv:
     InputBuffer::Loader@post[1,400]->exists(
               p:OclPath | p->includes(loaderLoading))
   ```

   In this constraint, we directly access the (sub)state InputBuffer::Loader, whose configurations can be expressed by single states, e.g., by loaderLoading.

Table 2. Temporal OCL Expressions and Equivalent CCTL Formulae

Temporal OCL Expression	Respective CCTL Formula
inv: obj@post[a,b]→exists(p:OclPath | p→forAll(c:OclConfiguration | expr))	AG $EG_{[a,b]}$(cctlExpr)
inv: obj@post[a,b]→exists(p:OclPath | p→exists(c:OclConfiguration | expr))	AG $EF_{[a,b]}$(cctlExpr)
inv: obj@post[a,b]→exists(p:OclPath | p→includes(cfg))	AG $EF_{[a,b]}$(cctlCfg)
inv: obj@post[a,b]→forAll(p:OclPath | p→forAll(c:OclConfiguration | expr))	AG $AG_{[a,b]}$(cctlExpr)
inv: obj@post[a,b]→forAll(p:OclPath | p→exists(c:OclConfiguration | expr))	AG $AF_{[a,b]}$(cctlExpr)
inv: obj@post[a,b]→forAll(p:OclPath | p→includes(cfg))	AG $AF_{[a,b]}$(cctlCfg)

2. The previous constraint can only become valid if orders have already been announced and accepted beforehand. To guarantee this, we require that configuration `Set{Acceptor::Accepting, Loader::Idle}` is always reached again within 400 time units. An according CCTL formula is

```
AG AF[1,400] (accepting & loaderIdle)
```

   The corresponding OCL expression is

```
context InputBuffer inv:
  self@post[1,400]->includes(
                        Set{accepting,loaderIdle})
```

3. To ensure correct processing of orders, the buffer must not accept a new order when it is still waiting for a delivery. In CCTL, this can be expressed by

```
AG( !(accepting & loaderWaiting) )
```

   In OCL, a corresponding constraint is

```
context InputBuffer inv:
  let errorCfg = Set{accepting,loaderWaiting}
  in
  self@post->forAll(p:OclPath |
                       p->excludes(errorCfg))
```

4. After accepting an order, the according item has to be delivered at most 80 time units later. In CCTL, we write

```
AG( (accepting & loaderIdle)
    -> AF[1,80](waitForOrder & loaderLoading) )
```

   In OCL, this constraint is expressed by a concatenation of collection operations, such that results from a preceding collection operation deal as input for the next operation.

```
context InputBuffer inv:
  let acceptCfg = Set{accepting,loaderIdle}
  let loadCfg   = Set{waitForOrder,loaderLoading}
  in
  self@post->forAll(p:OclPath |
                    p->first = acceptCfg
                    implies
                    p->subSequence(1,80)->
                               includes(loadCfg))
```

5. We demand that immediately after an order is accepted, `Acceptor` returns to state `WaitingForOrder` and `Loader` changes to `WaitingForDelivery`. Thereafter, the order should arrive within 100 time units at the buffer, and state `Loader::Loading` is entered, while `Acceptor` must not enter state `Accepting`. A corresponding CCTL formula is

```
AG EF E( accepting & loaderIdle
      U[1] E( waitForOrder & loaderIdle
           U[1,100] (!accepting & loaderLoading)
            )
       )
```

   For an OCL formula, we define a configuration sequence that is reused in a temporal expression:

```
context InputBuffer inv:
  let acceptPath =
        Sequence {
          Set{accepting,    loaderIdle   }[1],
          Set{waitForOrder, loaderWaiting}[1,100],
          Set{not accepting,loaderLoading}
        }
  in
  self@post->includes(acceptPath)
```

These constraints appear in a similar way for Statecharts of the other classes, e.g., AGVs have to periodically deliver items to stations, must wait until they are loaded, and should meet required delivery time limits. Moreover, AGVs should not collide, which is expressed by the following CCTL and OCL formulae[6]. In CCTL, we write:

```
AG(   (agv1.pos != agv2.pos) & (agv1.pos != agv3.pos)
    & (agv1.pos != agv4.pos) & (agv1.pos != agv5.pos)
    & (agv2.pos != agv3.pos) & (agv2.pos != agv4.pos)
    & (agv2.pos != agv5.pos) & (agv3.pos != agv4.pos)
    & (agv3.pos != agv5.pos) & (agv4.pos != agv5.pos) )
```

Note here that all instance combinations have to be explicitly named in the CCTL formula. This leads to quite long formulae, e.g., in 10 conjunction elements when there are 5 AGVs. In OCL, this constraint can be more easily specified:

[6] We are assuming that AGVs are moving along paths defined by fixed positions.

```
context AGV inv:
 self.allInstances(x,y:AGV | (x <> y) implies
                                    (x.pos <> y.pos))
```

7. Implementation

The OCL extensions presented here are integrated into our OCL editor which is implemented in Java 1.3 using Swing components. Users can load and edit OCL types, model descriptions, and OCL constraints in parallel. For parsing OCL type declarations, we have defined a separate grammar that considers the unique declaration needs for collection operations [4]. Class models and Statecharts are currently read as textual descriptions, using again a separate grammar similar to the one for OCL types. OCL expressions are parsed according to the official OCL grammar version 1.4 (see [5], Section 6.9). The three parsers are implemented with JavaCC (http://www.webgain.com) based on an early implementation for OCL version 1.1 [12].

An integrated type checker investigates whether OCL constraints are correctly declared w.r.t. a given system model. Additionally, constraints with temporal operations can automatically be translated to CCTL formulae and exported for further use in a model checking tool.

8. Summary and conclusion

We have presented OCL extensions for the specification of real-time constraints in state-oriented UML diagrams. Our extensions are based on an existing OCL metamodel and extend it by additional operations on existing types and by completely new types. The presented approach has demonstrated that an OCL extension for real-time specification is possible with only minor changes of current OCL syntax and semantics. Regarding the manufacturing case study, we were able to express all relevant real-time constraints with our OCL extension. We think that enhancements into this direction are necessary for future OCL versions to ensure correct behavior of real-time systems that are developed with UML. Our formal semantics by means of equivalences to CCTL formulae provides a sound basis for formal treatment such as an interface to a formal verification tool. Though our extensions are based on a future-oriented temporal logic, we see no limitation to extend it further to the specification of past-oriented constraints.

Acknowledgements

This work has been supported by a grant from the Deutsche Forschungsgemeinschaft (DFG) within the Priority Programme 1064 ”Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen”.

References

[1] W. Ahrendt et al. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In M. Ojeda-Aciego et al., editors, *8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, volume 1919 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 2000.

[2] T. Baar and R. Hähnle. An Integrated Metamodel for OCL Types. In *Proc. of OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core.*, Minneapolis, MN, USA, 2000.

[3] D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-Based Systems. In *Proc. of FMOODS'2000 - Formal Methods for Open Object-Based Distributed Systems IV*, Stanford, CA, USA, September 2000.

[4] S. Flake and W. Mueller. An OCL Extension for Real-Time Constraints. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2001.

[5] OMG. Unified Modeling Language Specification, Version 1.4. Technical report, Object Management Group, September 2001. URL: http://www.omg.org/technology/documents/formal/uml.htm (last visited September 18, 2001).

[6] 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, CA, USA, May 1999.

[7] S. Ramakrishnan and J. McGregor. Modelling and Testing OO Distributed Systems with Temporal Logic Formalisms. In *18th International IASTED Conference Applied Informatics'2000*, Innsbruck, Austria, 2000.

[8] M. Richters and M. Gogolla. A Metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Fort Collins, CO, USA*, volume 1723 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 1999.

[9] J. Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *Journal on Universal Computer Science (J.UCS), Springer-Verlag, Heidelberg*, 7(1):89–104, February 2001.

[10] J. Ruf and T. Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In *Conference on Correct Hardware Design and Verification Methods (CHARME97)*, Montreal, Canada, October 1997.

[11] J. Ruf and T. Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Conf. on Correct Hardware Design and Verification Methods (CHARME99)*, Bad Herrenalb, Germany, September 1999.

[12] J. Warmer. OCL Parser, Version 0.3. URL: http://www-4-.ibm.com/software/ad/library/standards/ocl.html (last visited on September 18, 2001).

[13] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[14] E. Westkaemper, M. Hoepf, and C. Schaeffer. Holonic Manufacturing Systems (HMS) - Test Case 5. In *Proceedings of Holonic Manufacturing Systems*, Lake Tahoe, CA, 1994.