An OCL Extension for Real-Time Constraints

Stephan Flake and Wolfgang Mueller

C-LAB, Paderborn University, Fürstenallee 11
33102 Paderborn, Germany
{flake, wolfgang}@c-lab.de

Abstract. The Object Constraint Language (OCL) was introduced to support the specification of constraints for UML diagrams and is mainly used to formulate invariants and operation pre- and postconditions. Though OCL is also applied in behavioral diagrams, e.g., as guards for state transitions, it is currently not possible to specify constraints concerning the dynamic behavior and timing properties of such diagrams.
This article discusses OCL's application for the dynamic behavior of UML Statechart diagrams and presents an OCL extension for specification of state-oriented time-bounded constraints. We introduce operations to extract state configurations from diagrams and define additional predicates over states and state configurations. The semantics of our OCL extension is given by employing time-bounded Computational Tree Logic (CTL) formulae. An example of a flexible manufacturing system with automated guided vehicles demonstrates the application of our extension.

1 Introduction

Currently, the Unified Modeling Language (UML) is well accepted in research and industry for a wide spectrum of applications. With the broad acceptance of UML, the Object Constraint Language (OCL) also has received a considerable visibility. OCL provides means for the specification of constraints in the context of UML diagrams, focusing on class diagrams and on guards in behavioral diagrams. However, OCL presently lacks sufficient specification means to cover constraints about the dynamic behavior of such diagrams, i.e., state configurations and evolution of states as well as state transitions over time cannot be expressed, so that OCL is currently not applicable for real-time specifications.

On the other hand, formal verification methods like equivalence and model checking have been well accepted through past years for some application domains. In particular, model checking has received a wide industrial acceptance for electronic system and protocol verification. Similar to the complementary means of UML and OCL, model checking needs a system description and a property specification as input, where properties are typically specified by formulae in temporal logics, mostly in Computational Tree Logic (CTL). Though already frequently applied, it often turns out that the specification of properties is regarded as a task too cumbersome for modelers and programmers who are not familiar with formal methods. With our OCL-based approach for model checking specification, it is possible to replace cryptic CTL specifications by more meaningful (extended) OCL specifications which are better tailored to the mental model of programmers.

In order to make OCL applicable for real-time specification in general and for model checking specification in particular, we introduce OCL extensions by concepts from temporal logics based upon a time-bounded variant of CTL. As we introduce our extensions on the basis of an OCL metamodel, they are seamlessly integrated into the existing OCL syntax and semantics (OCL Version 1.4). We enhance OCL by the notion of future-oriented state configurations as well as state transitions over time. All extensions come with a well defined semantics which is given by a translation of OCL statements to time-bounded CTL formulae. Though we present our approach as a constraint specification over the state space of UML Statechart diagrams by addressing future-oriented behavior, it is applicable without further modification for other state-oriented means like Activity Diagrams and also easily adaptable to past-oriented temporal logics.

The remainder of this article is structured as follows. In the next section, we give a brief overview of related works w.r.t. OCL extensions for formal verification. Section 3 gives an introduction to model checking and property specification by means of time-bounded CTL. In Section 4, we introduce our OCL extensions by collections of states and their operations and provide a semantics of the temporal operations by their relation to temporal logic formulae. Section 5 illustrates an application example before Section 6 briefly outlines our implementation. Finally, Section 7 summarizes and concludes this article.

2 Related Work

There currently exist very few approaches to apply OCL in the context of formal verification frameworks, though UML Statechart diagrams already apply well as a graphical front-end for model checking [3].

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 constraint specifications to dynamic logic (DL), an extension of Hoare logic. DL is used as input for formal verification. In that approach, OCL is applied to specify constraints on design patterns without modifying OCL.

Two other approaches consider temporal extensions for OCL. Distefano et al. define BOTL (Object-Based Temporal Logic) in order to facilitate the specification of static and dynamic properties [5]. BOTL is based on a combination of CTL and an OCL subset. Syntactically, BOTL is very similar to temporal formulae in CTL. Another temporal extension of OCL is defined by Ramakrishnan et al. [9, 10]. 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 logics formulae.

In contrast to previous approaches we introduce extensions to static OCL concepts towards dynamic concepts with only minor modifications to existing OCL syntax and semantics. In order to seamlessly integrate to existing OCL, our work is based on an OCL metamodel [2]. We have selected that model since it clearly separates metalevel and instance level for `OclType`. However, we think that an adaptation to another OCL metamodel such as [11] is possible without significant problems. Though our extensions are kept compliant with OCL syntax and existing types and operations, they also have 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 the underlying concepts covers Clocked CTL (CCTL), a time-bounded variant of CTL [12].

3 Model Checking

Symbolic model checking is mainly due to pioneering work of two groups: Clarke/ Emerson and Quielle/Sifakis [4]. Temporal logic based model checking is well established in hardware-oriented systems design for electronic circuits and protocol verification and receives growing interest in software design. Though the general problem is PSPACE-complete, symbolic representations like Binary Decision Diagrams allow verifications with up to 10^{120} states.

For model checking, given a parallel finite state machine (the model) and a temporal logic formula (the property specification), a model checker outputs either 'yes' if the model satisfies the formula or 'no' if the formula does not hold. In the latter case, usually a counter example can automatically be generated to show a particular model execution sequence which leads to a situation that contradicts the formula.

In the context of model checking, model representation is mostly based on Kripke structures (i.e., unit-delay temporal structures) which are derived from finite state machines. A Kripke structure $M = (P, S, s_0, T, L)$ is a tuple with a set of atomic propositions P, a set of states S, an initial state $s_0 \in S$, a transition relation between the states $T \subseteq S \times S$ such that every state has a successor state, and a state labeling function $L : S \rightarrow 2^P$.

For property specification, most model checkers are based on branching-time temporal tree logic specification. Temporal tree logic (TL) expresses information about states and future state transition paths. An execution path defines one possible future execution path starting from the current state as root. All possible execution paths establish an infinite tree with the current state as its root. One of the most frequently applied TLs is the Computational TL (CTL).

In CTL, temporal operators are always preceded by a path quantifier. Starting from the current state, the path quantifier either specifies to consider all possible execution paths (**A**) or it specifies that at least one execution path must exist (**E**) that satisfies the following formula part. Temporal operators specify the ordering of events along future-oriented execution paths. Table 1 gives an overview of the CTL operators.

In general, a CTL formula f can be built by applying the following recursive grammar:

$$f := \begin{array}{l} a \mid f \ \vee \ f \mid f \ \ /f \mid f \ \wedge \ f \mid f \ \oplus \ f \\ \mid \mathbf{EX}\, f \mid \mathbf{EF}\, f \mid \mathbf{EG}\, f \mid \mathbf{E}(f\ \mathbf{U}\ f) \\ \mid \mathbf{AX}\, f \mid \mathbf{AF}\, f \mid \mathbf{AG}\, f \mid \mathbf{A}(f\ \mathbf{U}\ f) \end{array}$$

where a is an atomic proposition.

Table 1. Temporal Operators

Name	Operator	Description
next-time	**X** f	next state on the path has to satisfy f
eventually	**F** f	some arbitrary state on the path has to satisfy f
always	**G** f	every state on the path has to satisfy f
until	f $\underline{\mathbf{U}}$ g	some state s on the path has to satisfy g and all states on the path up to s have to satisfy f

There are extensions to basic model checking for the verification of real-time systems. One variation is defined by Kropf and Ruf in [12] in the context of the RAVEN model checker. They extend Kripke structures to I/O-Interval structures and CTL to time-bounded Clocked CTL. The major difference with respect to Kripke structures is the introduction of a transition labeling function $I : T \rightarrow 2^{\mathbb{N}}$ with [min,max]-delay times. A state may be basically left at min-time and must be left after max-time.

Clocked CTL is a time-bounded variant of CTL with $\mathtt{X}_{[x]}$, $\mathtt{F}_{[x,y]}$, $\mathtt{G}_{[x,y]}$, $\underline{\mathtt{U}}_{[x,y]}$, where $x \in \mathbb{N}_0, y \in \mathbb{N}_0 \cup \{\infty\}$ are time bounds. The symbol ∞ is defined through: $\forall i \in \mathbb{N}_0 : i < \infty$. In the case of only one parameter the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the `X`-operator has no time bound, it is implicitly set to one.

In order to integrate more general concepts for specification of real-time systems, the extensions introduced in the next section refer to Clocked CTL concepts.

4 Real-Time OCL Extensions

In the domain of database systems, different types of *semantic integrity constraints* are distinguished [6]. *Static constraints* define required properties on nontransient system states, i.e., static properties within one system state. *Transition constraints* deal with system changes between two subsequent states. In real-time systems design, we additionally identify *temporal constraints* that consider sequences of state transitions in combination with time bounds. While static and transition constraints can already be expressed with OCL, it currently lacks means to express temporal constraints.

To overcome this, we introduce temporal OCL operations that enable modelers to specify state-oriented behavior. The OCL extensions presented in this article reason about possible future object states since we define the semantics based on a future oriented tree temporal logic without loss of generality. Accordingly, OCL can also be easily extended for specification of past-oriented constraints.

In the following, we first outline the concepts of our extensions based on an OCL metamodel. Thereafter, we describe the new types and their operations as well as necessary extensions to the predefined OCL type `OclAny`. The final paragraph of this section gives the semantics of the new operations by their translation to Clocked CTL expressions.

4.1 OCL Metamodel

At present, there are two metamodel proposals for OCL [11, 2]. We have selected the metamodel proposed by Baar and Hähnle [2] since it seems to be considerably stable and sufficiently generic for our purpose. That metamodel covers the complete OCL type system and aims to overcome difficulties in specifying metalevel constraints. Figure 1 gives an overview of that metamodel in form of a UML class diagram. In that figure, we have marked our extensions as bold, and it can be easily seen that our modifications are only marginal. We only require the additional metaclass `GenericParameter` and two new basic OCL types: `OclPath` and `OclConfiguration`.[1]

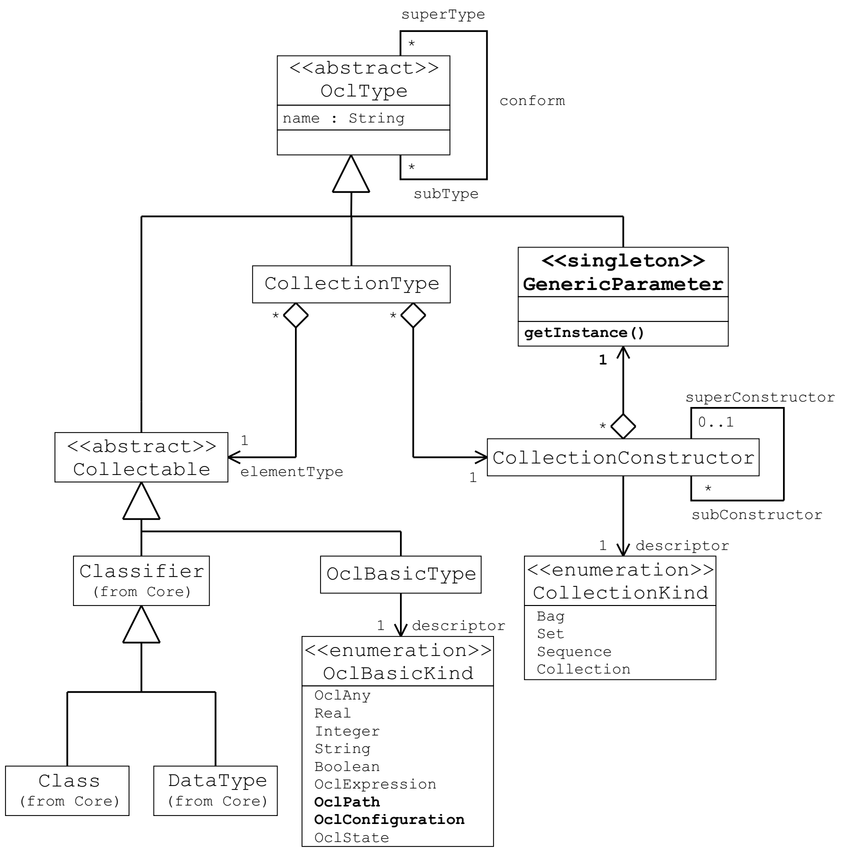


Fig. 1. Extented OCL Metamodel

[1] Note here that compared to [2] we have removed the literal `Enumeration` from `OclBasicKind`, due to issue 3143 in [14] (`Enumeration` instances are now retrieved from model classes by their path name).

The main idea of this metamodel is to consider the type `OclType` purely as a metatype, while all other predefined types are instances of (a subtype of) `OclType`, i.e., there is no subtype relationship between `OclType` and the predefined types like `Integer`, `String`, or `Boolean`. Instead, subtype relationships between OCL types are explicitly modeled by the association `conform`. The operations `attributes()`, `operations()`, and `associationEnds()` of `OclType` are in this metamodel only available for instances of the metaclass `Classifier` by inheritance from the UML core metamodel which does not affect OCL in its usage at the application level. Collection types are aggregations of a collection kind and an element type. An additional class `CollectionConstructor` is used to model the conformance between collection kinds, e.g., type `Set` conforms to `Collection`. The metamodel comes with OCL well-formedness rules to ensure that only predefined type names can be instantiated and that the conformance relationships are composed along the lines of standard specifications. Based on this metamodel, we introduce the following modifications.

– A unique generic parameter is defined in a singleton class `GenericParameter`. A generic parameter is required in declarations for instances of `CollectionConstructor` as a placeholder for a concrete basic OCL type. In OCL standard documents, this generic parameter is usually denoted as `T`, e.g., `Collection(T)`. As `OclType` has no operations in this metamodel, it is possible to regard `GenericParameter` as a subtype of `OclType`, and we add well-formedness rules to ensure the correct instance name:

  ```
  context GenericParameter
  inv: self.name = 'T'

  context OclType
  inv: self.allInstances->isUnique(name)
  ```

– Two new basic type names `OclConfiguration` and `OclPath` are added as literals to the enumeration `OclBasicKind`; detailed descriptions can be found in the next section.

4.2 States and Configurations

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 outlined 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] which is a part of the UML metamodel[3]. In that metamodel, state machines may have several kinds of states which are given as subtypes of the metaclass `StateVertex`. For our work, we are only interested in the states that represent a specific behavior, namely composite and simple states, and we do not consider pseudo, synch, stub, final, and submachine states here. The following code illustrates our extension of `OclState` that

[2] see [7], Section 3.75
[3] see [7], Section 2.12

introduces new attributes and operations compliant to the UML state machine metamodel.[4]

```
DataType <<enumeration>> StateType {
  composite,
  simple
}

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

We outline and informally describe the properties of `OclState` using the terms and identifiers of the UML state machine metamodel. Let `s` be an instance of type `OclState`. The enumeration attribute `stateType` points out whether `s` is a composite or a simple state. The boolean attribute `isConcurrent` indicates whether `s` contains concurrent substates (denoted as *regions*), and the boolean attribute `isRegion` checks whether `s` is a substate of a concurrent state. The state machine metamodel defines an association connecting states and their direct substates, and the corresponding association ends are `subvertex` and `container`. For `s`, we define the operations `subStates()` and `parentState()` which return the states accessible via this association. In the case of a top level state, `s.parentState()` returns `undefined`. Correspondingly, `s.subStates()` returns an empty set, if `s` is a simple state. `isActive()` evaluates to true if `s` is currently active. Its dual `notActive()` becomes true if `s` is not active. We also need the operation `anySubState()` which returns one non-deterministically chosen substate. When no substates are defined, the operation returns `undefined`.

Compliant with common OCL practice[5], we take some implicit presumptions for the remainder of this article. We assume that there is at most one Statechart diagram (resp. one state machine) for each classifier. 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`. Their properties are set according to their state machine specification.

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

[4] A grammar of the language we are using for declarations of basic and generic OCL types can be found in Appendix A.

[5] see [7], Section 7.5.10: States are already directly accessible in OCL expressions.

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.

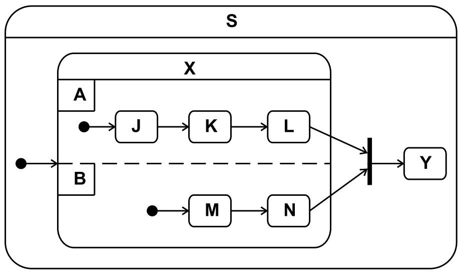


Fig. 2. Concurrent Statechart Diagram

Figure 2 gives an example of a concurrent Statechart diagram. That example has the top level state `S` which also denotes the classifier this Statechart diagram belongs to. Here, the initial configuration can be described by `Set{X::A::J,X::B::M}`. For this Statechart, `S.oclInState(X::A::J)` returns true in current OCL, although `X::A::J` is not a complete configuration of `S`. When investigating complete configurations on the level of simple states, we currently have to write

```
S.oclInState(X::A::J) and S.oclInState(X::B::M)
```

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 add the operation `config(): Set(OclConfiguration)` to `OclAny`. This operation returns the set of all valid configurations, i.e., all sets of *simple* states that are required to uniquely cover complete configurations. Table 2 gives some usage examples of `config()` when applied to the Statechart of Fig. 2. The notion of configuration also applies to substates, e.g., `X::A` and `X::A::J`. Note here that in the resulting sets all configurations are given by their complete path names and that the name of the topmost state is not included in the path of a configuration.

The operation `config()` returns a set of configurations, or more precisely, a set of sets of simple states, which is not flattened like other OCL operations on collections are. Note here that not necessarily all configurations have the same number of states, e.g., `{Y}` and `{X::A::J,X::B::M}` are both valid configurations for `S`. The operation `config()` checks configurations for their validity. For instance,

```
context S
inv: self.config->includes(c:OclConfiguration |
                                                c = Set {X::A::J,X::B::M} )
```

checks if `Set{X::A::J,X::B::M}` is a valid configuration for `S`. This also includes a check for the newly introduced OCL type `OclConfiguration`.

Table 2. Sample Usages of the Operation `config()`

Expression	**Result**
S.config	$\{$X::A::J, X::A::K, X::A::L$\} \times \{$X::B::M, X::B::N$\} \cup \{$Y$\}$
S.config→size	$3 * 2 + 1 = 7$
X::A.config	$\{$X::A::J, X::A::K, X::A::L$\}$
X::A.config→size	3
X::A::J.config	$\{$X::A::J$\}$
X::A::J.config→size	1

The next section presents basic operations for instances of that type, while Section 4.3 investigates on dynamic issues of configurations with respect to the runtime execution of state machines.

Operations of OclConfiguration. As we interpret `OclConfiguration` as a new built-in type for a representation of specific sets of OclStates, most operations of Ocl type `Set` can be reused. We only have to elaborate on operations that return collections, since the result might be an arbitrary set of OclStates rather than a valid configuration. Therefore, we cannot directly adopt operations like union(), intersection(), including(), excluding(), symmetricDifference(), select(), reject(), collect() for OclConfigurations without modification. Nevertheless, access to those set operations is still possible via type cast operations, e.g., `asSet()`.

Again, we use the grammar given in Appendix A to present the operations defined for `OclConfiguration`.

```
OclBasicType OclConfiguration <supertype> OclAny {
  --
  -- operations adopted from generic type Set
  --
  =           (c:OclConfiguration) : Boolean;
  <>          (c:OclConfiguration) : Boolean;
  size        ()                   : Integer;
  count       (s:OclState)         : Integer;
  isEmpty     ()                   : Boolean;
  notEmpty    ()                   : Boolean;
  exists      (e:OclExpression)    : Boolean;
  forAll      (e:OclExpression)    : Boolean;
  includes    (s:OclState)         : Boolean;
  includesAll (t:Set(OclState))    : Boolean;
  excludes    (s:OclState)         : Boolean;
  excludesAll (t:Set(OclState))    : Boolean;
  --
  -- new operations
  --
  isActive    ()                    : Boolean;
  notActive   ()                    : Boolean;
  --
  -- type cast operations
  --
  asSet       ()                    : Set(OclState);
  asBag       ()                    : Bag(OclState);
  asSequence  ()                    : Sequence(OclState);
}
```

Except the two newly introduced operations `isActive()` and `notActive()` all other operations of `OclConfiguration` can be immediately adopted from the generic types `Collection` and `Set`[6]. The semantics of `isActive()` and `notActive()` is described in the style of the official OCL specification as follows, where `cfg` denotes an instance of `OclConfiguration` in the context of a classifier `C`.

```
-------------------------------------------------------------------
cfg->isActive() : Boolean

  True if all states of cfg are active.

pre:  C.config->includes(cfg)
post: result = cfg->forAll(s : OclState | s.isActive)
-------------------------------------------------------------------
cfg->notActive() : Boolean

  True if at least one of the states of cfg is not active.

pre:  C.config->includes(cfg)
post: result = not cfg->isActive
-------------------------------------------------------------------
```

To access `OclConfiguration` properties, we make use of the arrow-operator. This solution is chosen to keep the compliance with existing OCL and its syntax for collection operations, although OclConfiguration is defined as a basic type.

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

Instances of a new basic type `OclPath` – representing sequences of OclConfigurations – are declared by the following grammar:

[6] see [7] Section 7.8.2

```
cfgSequence       := "Sequence" "{" cfgExprList "}"
cfgExprList       := cfgExpr (qualifiers)?
                       ( "," cfgExpr (qualifiers)? )*
cfgExpr           := ("not")?
                     ( configuration
                       | "(" configurationList ")"
                     )
configuration     := stateName
                     | "Set" "{" stateName ("," stateName)* "}"
configurationList :=  configuration ("or" configuration)*
stateName         := pathName
qualifiers        := "[" actualParameterList "]"
```

where the non-terminals `pathName`, `qualifiers`, and `actualParameterList` are defined according to the official OCL Grammar. Note here that OCL already covers sequence declarations through `literalCollection`, so that there is no need to add or modify rules with that respect.

In the above grammar, `qualifiers` refers to a [min,max]-time interval specification corresponding to the Clocked CTL time intervals introduced in Section 3. Thus, this expression has one or two comma seperated subexpressions. The latter is of type $Integer \times (Integer \cup \{'inf'\})$, specifying discrete [min,max]-time intervals with an optional infinite upper bound. In the case of only one subexpression, that expression has to evaluate to type `Integer` and specifies both lower and upper bound. Configuration expressions without qualifiers implicitly have the interval `[1,'inf']` as a default.

For an `OclPath` example, we take the Statechart diagram in Fig. 2 and define in the context of `S`:

```
let V = X::A
let W = X::B
```

The following `let`-expression specifies a sequence for state `S::X` which changes from the initial configuration {`V::J,W::M`} to {`V::K,W::M`} after some time, is staying in this configuration between 5 and 50 time units, then changes to {`V::L,W::M`} or {`V::K,W::N`}, remaining in this configuration for exactly 10 time units, and finally changes to configuration {`V::L,W::N`}.

```
let p = Sequence {  Set {V::J,W::M},
                    Set {V::K,W::M} [5,50],
                  ( Set {V::L,W::M} or Set {W::K,W::N} ) [10]
                    Set {V::L,W::N}
                 }
```

OclPath Operations. An instance of `OclPath` is interpreted as a possible execution sequence composed of OclConfigurations for a given Statechart diagram, resp. state machine. Similar to `OclConfiguration`, the existing OCL sequence operations can be immediately applied to `OclPath`. Nevertheless, we do not define all common sequence operations for `OclPath`, as many of them would result in arbitrary collections of OclConfigurations which are not valid OclPaths. Note here that access to all common sequence operations is still available through type casting.

The type`OclPath` is defined as follows where the semantics of all operations can be directly derived from the generic OCL types `Collection` and `Sequence`[7].

```
OclBasicType OclPath <supertype> OclAny {
  --
  -- basic operations derived from OclCollection
  --
  =            (p:OclPath)                : Boolean;
  <>           (p:OclPath)                : Boolean;
  size         ()                         : Integer;
  isEmpty      ()                         : Boolean;
  notEmpty     ()                         : Boolean;
  count        (t:OclConfiguration)       : Integer;
  exists       (e:OclExpression)          : Boolean;
  forAll       (e:OclExpression)          : Boolean;
  includes     (t:OclConfiguration)       : Boolean;
  includesAll  (s:Set(OclConfiguration)) : Boolean;
  excludes     (t:OclConfiguration)       : Boolean;
  excludesAll  (s:Set(OclConfiguration)) : Boolean;
  --
  -- basic operations derived from OclSequence
  --
  at           (i:Integer)                : OclConfiguration;
  first        ()                         : OclConfiguration;
  last         ()                         : OclConfiguration;
  append       (t:OclConfiguration)       : OclPath;
  prepend      (t:OclConfiguration)       : OclPath;
  subSequence  (l:Integer,u:Integer)      : OclPath;
  --
  -- type cast operations
  --
  asSet        ()               : Set(OclConfiguration);
  asBag        ()               : Bag(OclConfiguration);
  asSequence   ()               : Sequence(OclConfiguration);
}
```

4.3 Temporal Operations

Temporal operations have to be introduced 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, same concepts also apply to past operators and can be introduced correspondingly.

[7] see [7] Section 7.8.2

For our extension we first consider the `@pre` operator which is already available in OCL. This operator is only allowed in operation postconditions and used to recall the value of an object when the operation was called. Correspondingly, we define `@post` that regards future points in time. For a seamless integration of that operator, we interpret the symbol @ as an individual operator, such as the dot- and arrow-operators. This means to take `pre` and `post` as *operations* of `OclAny` and restrict the @-operator to be used only for those temporal operations. Under these assumptions we only need very few minor changes w.r.t. the OCL grammar, so that main syntax and semantics of OCL can be kept. For a complete summary of all OCL grammar changes, the reader is referred to Appendix B.

Extensions to OclAny. The OCL basic type `OclAny` is the abstract superclass of all OCL basic types and all other model classes (i.e., instances of the metatypes Class or DataType w.r.t. a given UML class diagram). In order to introduce temporal operations to OCL we extend `OclAny` with the operations `pre()`, `post()`, and `next()`. For the sake of completeness, the operation `config()` (cf. Section 4.2) is also listed in the following code fragment.

```
OclBasicType OclAny {
  -- keep standard OclAny operations
  ....
  -- new operations
  config () : Set(OclConfiguraion);
  pre    () : OclAny;
  post   () : Set(OclPath);
  next   () : Set(OclConfiguration);
}
```

As the return type of operation `pre()` can here only be declared as `OclAny`, we have to ensure type consistency by an additional postcondition. We define `post()` as an operation that returns a set of OclPaths, i.e., a *set* of possible future execution sequences. It is to be defined as a set since there can be various possible orders of executions in a Statechart diagram. Operation `next()` returns a set of all possible configurations after one time unit. Furthermore, we allow the declaration of a [min,max]-time interval in combination with `post()`, as already introduced for `OclPath`.

The informal semantics is given as follows, where `obj` denotes an instance of `OclAny`.

```
-------------------------------------------------------------------
obj@pre() : OclAny
  This operation may be used in operation postconditions only. It
  returns the value of obj at the time of entering the respective
  operation.
post: result.oclIsTypeOf(obj)
-------------------------------------------------------------------
```

```
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.
-----------------------------------------------------------------
obj@post()[b] : Set(OclPath)
  Same as obj@post[b,b]. b must be of type Integer.
-----------------------------------------------------------------
obj@post() : Set(OclPath)
  Same as obj@post[1,'inf'].
-----------------------------------------------------------------
obj@next() : Set(OclConfiguration)
  Similar to obj@post[1,1], but @next returns a set of
  OclConfigurations that are valid after the next time step.
-----------------------------------------------------------------
```

Until-Operator. For our temporal OCL extension, we have to introduce a logical `until` operator to be able to express causal dependencies between subsequent configurations. To define the semantics, we first have to define a validation relation $\models$ over OCL expressions that result in OclConfigurations:

Definition 1. *Given a time step t, a classifier* `S`*, its configuration* `c` *at time step t, and an OclExpression* `expr`*. Let* `res` *be the result from evaluating* `expr` *at time step t, then*

$$(\mathtt{c}, t) \models \mathtt{expr} \quad :\Leftrightarrow \quad \begin{cases} \mathtt{expr.evaluationType} = \mathtt{OclConfiguration} \\ \text{and } \mathtt{S.config}{\rightarrow}\mathtt{includes(expr)} \\ \text{and } \mathtt{c} = \mathtt{res} \end{cases}$$

We say that `expr` *is satisfied by* `c` *at time step t, if and only if* `expr` *evaluates to a valid instance* `res` *of* `OclConfiguration` *in the context of* `S` *and* `c` *equals* `res`*.*

The binary logical operator `until` is defined for pairs of OclExpressions that both evaluate to instances of type `OclConfiguration`. Optionally, `until` can be supplied with an interval declaration according to the grammar rules introduced for [min,max]-time intervals in Section 4.2. Implicitly, the interval is set to [1,'inf'] by default.

The logical operator `until` is defined as follows.

Definition 2. *Given a time step t, a classifier* `S`*, and its configuration* `c` *at time step t. Let* `p` *be an instance of* `OclPath` *whose configurations are all valid in* `S`*. Let* `expr1`*,* `expr2` *be two OCL expressions.*

$$(\mathtt{c}, t) \models (\mathtt{p} : \mathtt{OclPath} \mid \mathtt{expr1}\ \mathtt{until}[\mathtt{a}, \mathtt{b}]\ \mathtt{expr2})$$

$$:\Leftrightarrow$$

$$\textit{there exists an } \mathtt{i},\ \mathtt{a} \leq \mathtt{i} \leq \mathit{min}(\mathtt{p.size}, \mathtt{b}),\ \textit{such that } (\mathtt{p.at(i)}, t + \mathtt{i}) \models \mathtt{expr2}$$
$$\textit{and for all } \mathtt{j}, 1 \leq \mathtt{j} < \mathtt{i}, \textit{holds: } (\mathtt{p.at(j)}, t + \mathtt{j}) \models \mathtt{expr1}$$

Translating Temporal OCL Expressions to CCTL. We can now formally define our temporal OCL extensions by their translation to Computational CTL formulae as they were introduced in Section 3. We focus on OCL invariants, so that all corresponding CCTL formulae start with the `AG` operator, i.e., with 'always globally'. Table 3 lists temporal OCL operations that directly match to CCTL expressions. In that table, `expr` is of type `OclExpression` and `configuration` of type `OclConfiguration`. The table gives a translation by templates and it should be easy to see how it is applied to nested expressions. Due to space limitations we additionally use a compact form for the OCL expressions in that table, e.g., we use

```
obj@post[a,b]->exists(forAll(expr))
```

instead of explicitly declaring the iterators:

```
obj@post[a,b]->exists(p:OclPath |
                                   p->forAll(c:OclConfiguration | expr))
```

Table 3. Temporal OCL Expressions and Equivalent CCTL Formulae

Temporal OCL Expression	**Respective CCTL Formula**
inv: obj@post[a,b]→exists(forAll(expr))	AG EG$_{[a,b]}$(expr)
inv: obj@post[a,b]→exists(exists(expr))	AG EF$_{[a,b]}$(expr)
inv: obj@post[a,b]→exists(includes(configuration))	AG EF$_{[a,b]}$(configuration)
inv: obj@post[a,b]→exists(expr1 until[c,d] expr2]	AG EG$_{[a,b]}$ E(expr1 $\underline{\text{U}}_{[c,d]}$ expr2)
inv: obj@post[a,b]→forAll(forAll(expr))	AG AG$_{[a,b]}$(expr)
inv: obj@post[a,b]→forAll(exists(expr))	AG AF$_{[a,b]}$(expr)
inv: obj@post[a,b]→forAll(includes(configuration))	AG AF$_{[a,b]}$(configuration)
inv: obj@post[a,b]→forAll(expr1 until[c,d] expr2]	AG AG$_{[a,b]}$ A(expr1 $\underline{\text{U}}_{[c,d]}$ expr2)

Configuration sequences translate to CCTL formulae as follows. Let $e_1, e_2, ..., e_n$ be elements of a sequence declaration, where e_i can be either simple OclConfigurations or complex expressions, specified with time intervals $[a_i, b_i]$. The temporal OCL expression

`obj@post[a,b]`$\rightarrow$`includes(Sequence`$\{e_1[a_1, b_1], e_2[a_2, b_2], ..., e_n\}$`)`

translates to the CCTL formula

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

Note here that the path quantifier, which is applied to each sequence element, depends on the preceding operations.

5 Example

The following section outlines the previously described concepts by the example of a Holonic Manufacturing System (HMS) case study. The HMS case study was introduced by the IMS Initiative TC 5. It is composed of a set of different manufacturing stations and a transport system as shown by the virtual 3D model in Fig. 3. The different manufacturing stations transform workpieces, 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 AGVs (Automated Guided Vehicles), i.e., autonomous vehicles that carry workpieces between stations. We assume that stations have an input buffer for incoming workpieces and that each AGV can take only one workpiece at a time.

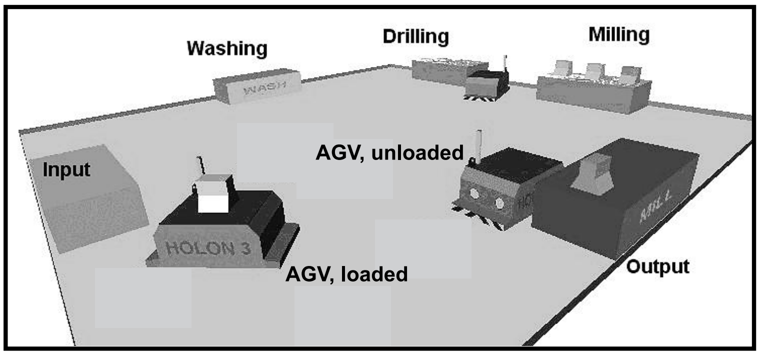


Fig. 3. 3D Model of the Manufacturing Scenario.

The whole system is basically characterized by the following application flow.

– An AGV v_i is idle until it receives a request for delivery from a station s_k. Then, it
  1. sends the distance d_i from its current position to s_k,
  2. moves to s_k on notification of acceptance from s_k,
  3. takes the workpiece from s_k and moves it to the next destination,
  4. moves to a parking position and returns to Step 1.

– Once having located a completed workpiece at its output, a station s_k
  1. sends a request for delivery to the next destination station s_{dest},
  2. is waiting for a notification from s_{dest} for a specific time period,
  3. returns to Step 1 if s_{dest} does not reply or answers with a reject to the request,
  4. broadcasts a request for delivery to all AGVs,
  5. is collecting messages with distances d_i from idle AGVs v_i for a specific time period,
  6. returns to Step 4 if no AGV replies,
  7. selects one AGV v_i from all received distances d_i, notifies AGV v_i for its acceptance and notifies the other AGVs for their rejection.

5.1 UML Statechart Diagrams

We assume that AGVs, stations, and the input and output storages are all modeled by class diagrams and that their behavior is given by Statechart diagrams. We focus here on the subaspects of the specification of a station input buffer that is in charge of delivery request management. The corresponding Statechart diagram (see Fig. 4) is separated into two parallel substates, one for handling messages from other stations which request an notification acceptance for delivery (Acceptor). The second one processes the loading after the acceptance of a delivery (Loader).

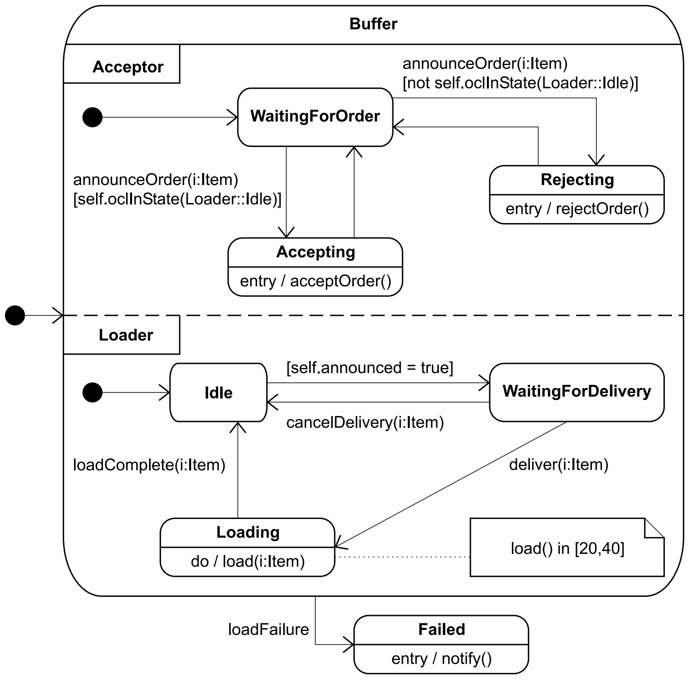


Fig. 4. Statechart Diagram of the Input Buffer

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]” means that `Loading` takes between 20 and 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.

5.2 OCL Constraints

We now specify some example constraints for the buffer of the previously described Statechart diagram, applying our OCL extension with temporal operations.

We first request that new workpieces have to periodically arrive at the input buffer within time intervals of at most 100 time units. In other words, state `Loading` can always be reached again within 100 time units. The corresponding OCL expression is

```
context Buffer
inv: Loader@post[1,100]->forAll(p:OclPath|p->includes(Loading))
```

Note that `Buffer::Loader` is a composite sequential state and that its configurations can be expressed by single states. The next invariant defines that a buffer must not accept a new order when still waiting for a delivery:

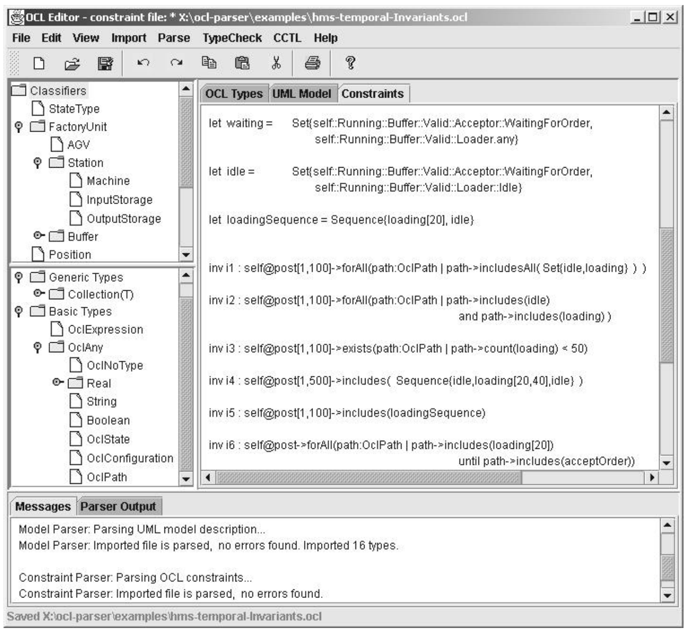
```
context Buffer
inv: self@post->forAll(p:OclPath | p->excludes(
           Set{Acceptor::Accepting, Loader::WaitingForDelivery}))
```

Finally, we present an example for an application of a configuration sequence and the usefulness of `let`-expressions in complex constraints. We request that immediately after an order is accepted, `Buffer::Acceptor` returns to state `WaitingForOrder` and `Buffer::Loader` changes to `WaitingForDelivery`. Thereafter, the order should arrive within 100 time units at the buffer, and state `Loading` is entered, while `Acceptor` must not be in state `Accepting`.

```
context Buffer
def: let waitForOrder   = Acceptor::WaitingForOrder
     let waitForDeliver = Loader::WaitingForDelivery
inv: let acceptPath =
          Sequence {
            Set{Acceptor::Accepting,Loader::Idle}[1],
            Set{waitForOrder,waitForDelivery}[1,100],
            Set{not Acceptor::Accepting,Loader::Loading}
          } in
     self@post->includes(acceptPath)
```

6 Implementation

The temporal extensions as presented here are integrated into our OCL parser and type checker (see Fig. 5). The checker is implemented in Java 1.3 using Swing components. The visual capture loads and edits OCL types, model descriptions, and OCL constraints in parallel. The parsers are implemented with JavaCC (www.webgain.com) based on an early implementation of OCL Version 1.1 [8]. Correctly parsed types are integrated into type tree structures. Class models and Statecharts are currently modeled by textual means. For this, we have implemented a system to parse textual descriptions of class models and Statecharts. Constraints with temporal operations are automatically translated to CCTL formulae for model checking.



```
let waiting =      Set{self::Running::Buffer::Valid::Acceptor::WaitingForOrder,
                       self::Running::Buffer::Valid::Loader.any}

let idle =         Set{self::Running::Buffer::Valid::Acceptor::WaitingForOrder,
                       self::Running::Buffer::Valid::Loader::Idle}

let loadingSequence = Sequence{loading[20], idle}


inv i1 : self@post[1,100]->forAll(path:OclPath | path->includesAll( Set{idle,loading} ) )

inv i2 : self@post[1,100]->forAll(path:OclPath | path->includes(idle)
                                                 and path->includes(loading) )

inv i3 : self@post[1,100]->exists(path:OclPath | path->count(loading) < 50)

inv i4 : self@post[1,500]->includes(  Sequence{idle,loading[20,40],idle} )

inv i5 : self@post[1,100]->includes(loadingSequence)

inv i6 : self@post->forAll(path:OclPath | path->includes(loading[20])
                                                 until path->includes(acceptOrder))
```

```
Model Parser: Parsing UML model description...
Model Parser: Imported file is parsed,  no errors found. Imported 16 types.

Constraint Parser: Parsing OCL constraints...
Constraint Parser: Imported file is parsed,  no errors found.
```

Fig. 5. OCL Parser and Type Checker

7 Summary and Conclusion

We have presented an OCL extension for the specification of real-time constraints in state-oriented UML diagrams. Our extensions were outlined on the basis of an OCL metamodel. The presented approach has demonstrated that an OCL extension for real-time specification is possible with only little changes to syntax and semantics of current OCL. It has been demonstrated that extensions into that direction are not in conflict with general OCL concepts. Due to the increasing importance of real-time systems we think that enhancements for real-time systems specification are worth to be considered in future official OCL versions.

The semantics of our extensions are given by their translation to Clocked CTL formulae which also provides a sound basis for a combined UML and OCL application for formal verification by model checking. The presented extensions are based on a future-oriented temporal logic. However, the general concepts can be easily extended to past-oriented constraints and to a generalization to capture various additional logics.

Acknowledgements

This work has been supported by the DFG project GRASP within the DFG Priority Programme ”Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen”. We appreciate the help of Prof. P.H.P. Bhatt for valuable discussions and proof reading.

References

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In M. Ojeda-Aciego, I. P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, volume 1919 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, Oct. 2000.

[2] T. Baar and R. Hähnle. An Integrated Metamodel for OCL Types. In R. France, B. Rumpe, J.-M. Bruel, A. Moreira, J. Whittle, and I. Ober, editors, *Proc. of OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core*, Minneapolis, Minnesota, USA, 2000.

[3] U. Brockmeyer and G. Wittich. Tamagotchis Need Not Die – Verification of STATEMATE Designs. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, 1998.

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

[5] D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-Based Systems. In S. F. Smith and C. L. Talcott, editors, *Proc. of FMOODS'2000 – Formal Methods for Open Object-Based Distributed Systems IV*, Stanford, CA, USA, September 2000.

[6] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley World Student Series, 3rd edition, 2000.

[7] Object Management Group (OMG). UML Unified Modeling Language Specification, Version 1.3, March 2000. URL: http://www.omg.org/technology/documents/formal/uml.htm (last visited on July 11th, 2001).

[8] OCL Parser, Version 0.3, 1997. URL: http://www-4.ibm.com/software/ad/library/standards/ocl-download.html (last visited on July 11th, 2001).

[9] 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.

[10] 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.

[11] 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. Second International Conference, Fort Collins, CO, USA*, volume 1723 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 1999.

[12] 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 (CHARME)*, pages 146–166, Montreal, Canada, October 1997. IFIP WG 10.5, Chapman and Hall.

[13] J. Warmer. The Draft 1.4 OCL Grammar, Version 0.1c. Technical report, Klasse Objecten, June 2000. URL: http://www.klasse.nl/ocl/ocl-grammar-01c.pdf (last visited on July 11th, 2001).

[14] J. Warmer. UML 1.4 RTF: OCL Issues – Changes from 1.3 to 1.4. Technical report, Klasse Objecten, March 2000. URL: http://www.klasse.nl/ocl/ocl-issues.pdf (last visited on July 11th, 2001).

Appendix A

Grammar

The here presented grammar in EBNF is for the definition of OCL predefined types based on the metamodel discussed in Section 4.1. A type definition consists of the type name, preceded by its mandatory metatype name, and succeeded by optional supertypes and the definition body. We have added a non-terminal `umlStereotype` to be able to declare types as ”abstract” or as ”enumeration”. We allow sequences of names in the rule `definitionBody` to define enumeration types. The grammar rule `returnType` is modified in comparison to its OCL definition, as we allow parameter names and operation names in a return type, e.g., `e.evaluationType`.

Moreover, we had to modify the rule for `operationName`; logical operation names like `implies` and `not` can be removed (cf. [14], issue 3138).

```
typeDefinitions     ::= "<startTypeDef>" ( typeDefinition )*
                        "<endTypeDef>"
typeDefinition      ::=  metatypeSpecifier ( umlStereotype )?
                           typeSpecifier ( supertypes )?
                           "{" definitionBody "}"
metatypeSpecifier   ::= "CollectionConstructor" | "CollectionType"
                      | "OclBasicType" | "Class" | "DataType"
umlStereotype       ::= "<<" name ">>"
typeSpecifier       ::= simpleTypeSpecifier
                      | collectionType
supertypes          ::= "<supertype>"
                          typeSpecifier ( "," typeSpecifier )*
definitionBody      ::= ( operation | attribute )*
                      | ( name ( ’,’ name )* )?
operation           ::= operationName
                          "(" ( formalParameterList )? ")"
                          ( ":" returnType )? ";"
operationName       ::= name | "=" | "+" | "-" | "<" | "<="
                      | ">=" | ">" | "/" | "*" | "<>"
formalParameterList ::= name ":" typeSpecifier
                          ( "," name ":" typeSpecifier )*
returnType          ::= ( collectionKind
                          "(" pathName ( "." operationName )? ")"
                        )
                      | ( pathName ( "." operationName )? )
attribute           ::= name ":" simpleTypeSpecifier ";"
collectionType      ::= collectionKind "(" simpleTypeSpecifier ")"
collectionKind      ::= "Set" | "Bag" | "Sequence" | "Collection"
simpleTypeSpecifier ::= pathName
pathName            ::= name ( "::" name )*
```

Example

For the previous grammar, the definition of the generic type `Set` is specified as follows:

```
CollectionConstructor Set(T) <supertype> Collection(T) {
    =             (s : Set(T))          : Boolean;
    <>            (s : Set(T))          : Boolean;
    select        (e : OclExpression)   : Set(T);
    reject        (e : OclExpression)   : Set(T);
    including     (t : T)               : Set(T);
    excluding     (t : T)               : Set(T);
    union         (s : Set(T))          : Set(T);
    union         (b : Bag(T))          : Bag(T);
    intersection  (s : Set(T))          : Set(T);
    intersection  (b : Bag(T))          : Set(T);
    -             (s : Set(T))          : Set(T);
    collect       (e : OclExpression)   : Bag(e.evaluationType);
    symmetricDifference       (t : T)   : Set(T);
}
```

`Set` inherits all operations from `Collection` because of the subtype relationship. Note here that operations = and <> are explicitly listed because of their different semantics in the generic types `Set`, `Sequence`, and `Bag`.

Appendix B

This appendix gives a listing of all OCL rules that had to be modified for our extension with respect to OCL version 1.4 RTF [13].

1. Add a new keyword `until` to the list of logical operators. Note that we do not restrict the qualifiers to conform to a well-formed interval declaration here, although this can be easily realized.

   ```
   logicalOperator ::= "and" | "or" | "xor" | "implies"
                     | ( "until" (qualifiers)? )
   ```

2. Add the @-operator as a third property access facility to the grammar rule `postfixExpression`. It may be used only for temporal operations.

   ```
   postfixExpression ::= primaryExpression
                          ( ( "." | "->" | "@" ) propertyCall )*
   ```

3. Remove the rule `timeExpression` from the OCL grammar, as `@pre` is now derived through `postfixExpression`. Note that we now regard `pre` as an operation defined in `OclAny`.
4. Remove the term `(timeExpression)?` from the rule `propertyCall` to obtain a consistent grammar again. The resulting rule is

   ```
   propertyCall ::= pathName (qualifiers)?
                      (propertyCallParameters)?
   ```