Formal semantics of static and temporal state-oriented OCL constraints

Stephan Flake, Wolfgang Mueller

C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany, E-mail: {flake,wolfgang}@c-lab.de

Abstract. The textual Object Constraint Language (OCL) is primarily intended to specify restrictions over UML class diagrams, in particular class invariants, operation pre-, and postconditions. Based on several improvements in the definition of the language concepts in last years, a proposal for a new version of OCL has recently been published [43]. That document provides an extensive OCL semantic description that constitutes a tight integration into UML. However, OCL still lacks a semantic integration of UML Statecharts, although it can already be used to refer to states in OCL expressions.

This article presents an approach that closes this gap and introduces a formal semantics for such integration through a mathematical model. It also presents the definition of a temporal OCL extension by means of a *UML Profile* based on the metamodel of the latest OCL proposal. Our OCL extension enables modelers to specify behavioral state-oriented real-time constraints. It provides an intuitive understanding and readability at application level since common OCL syntax and concepts are preserved. A well-defined formal semantics is given through the mapping of temporal OCL expressions to temporal logics formulae.

Keywords: Object Constraint Language – UML Statecharts – UML Profile – Real-time constraints – Temporal logics

1 Introduction

The Object Constraint Language (OCL) has been part of the UML since UML version 1.3 and enables modelers to specify constraints in the context of a given UML model. OCL is used to specify invariants attached to classes, pre- and postconditions of operations, and guards for state transitions.

In the official UML 1.5 specification, a concrete syntax of OCL is given, but due to a missing metamodel only an informal description of the semantics of OCL expressions is provided [26, Chapter 6]. In reply to the OCL 2.0 Request for Proposals by the Object Management Group (OMG), an extensive OCL metamodel proposal has been submitted to provide a tighter integration of OCL with UML [43]. In the remainder of this article, we refer to [43] as the *OCL 2.0 proposal*. That document comprises the work of several significant contributions concerning the development of OCL in recent years and has been recommended for adoption by the Analysis and Design Platform Task Force of the OMG in March 2003 [27]. The OCL 2.0 proposal includes two semantic descriptions, a normative one using UML concepts and an informative one that is based on the mathematical set-theoretical notion of the *object model* introduced by Richters [31].[1]

Most OCL language elements have an impact on (values of) elements of class diagrams, e.g., attributes, operations, and associations between classes. But it is also possible to refer to Statechart states in OCL expressions. However, the semantics of state-related operations is still only informally described in the OCL 2.0 proposal, i.e., an integration of UML Statecharts into the language concepts of OCL on the metalevel is still missing. No significant work is available that provides a formal definition of OCL and covers expressions over states in UML Statecharts.

Moreover, UML and OCL are missing adequate means to specify constraints over the *dynamic behavior* of a UML model. However, it is essential to support the definition of temporal constraints already in early phases of development in order to specify correct system behavior over time. While other approaches focus on UML

---

[1] Unfortunately, the two semantic definitions are not consistent yet, as, e.g., the `OclMessage` concept is not supported in object models.

Collaboration and Sequence Diagrams and consider temporal OCL constraints for event communication (e.g., [6, 22, 44]), we investigate consecutiveness of states and state transitions in UML Statecharts and time-bounded constraints over sequences of states.

This article is based on previous work concerning an OCL extension that enables modelers to specify state-oriented real-time constraints. Since the current UML specification does not come with an OCL metamodel, we first took the OCL type metamodel presented by Baar and Hähnle [1] and performed a rather heavyweight extension by directly extending that metamodel [18]. More recently, we developed a UML Profile for our temporal OCL extension based on the OCL 2.0 proposal [17]. This article is an enhanced version of the latter publication. We also provide a formal underlying model and clarify semantic issues with respect to the role of Statechart states in the evaluation of (temporal) OCL expressions. In particular, we address the following issues in order to overcome the previously mentioned deficiencies:

1. We perform a semantic integration of UML Statecharts into OCL language concepts by the formal definition of a *Statechart configuration*, i.e., a mathematically precise status description that captures the set of currently activated states of a UML Statechart. We integrate this notion into the formal object model approach, which was introduced by Richters and extend his description of an overall *system state* accordingly.

2. Based on Step 1, we define a formal semantics of operation `oclInState(s:OclState)`.

3. In order to be able to specify temporal OCL constraints, we introduce the notion of a *trace* over a given model. A trace is an (infinite) sequence of system states. Each element occurring in a trace indicates that a 'noteworthy' change to the model has happened, e.g., that an operation is called or terminated.

4. Last but not least, for giving a semantics to *time-bounded* temporal OCL expressions, traces have to be equipped with an inherent notion of time. We presume a given implicit global reference clock. However, we are aware that this assumption is not generally applicable, e.g., for distributed systems.

For readers not familiar with OCL, Sect. 2 provides a brief introduction to OCL. Section 3 discusses related work. In Sect. 4, we integrate syntax and static semantics of UML Statecharts as defined in the official UML 1.5 specification into the Richters' formal object models [31].

The UML Profile for state-based real-time constraints is defined in Sect. 5. It is based on temporal logics concepts derived from the domain of formal verification by Real-Time Model Checking. We present a mapping of (future-oriented) temporal OCL expressions to time-annotated formulae expressed in a discrete temporal logics called *Clocked CTL* [36]. With a mapping of (parts of) the UML model to a set of finite state machines

it is then possible to apply Real-Time Model Checking, i.e., a given model is checked if it satisfies required real-time properties specified by state-oriented temporal OCL expressions. Section 6 gives an application example before the final section closes with a summary and conclusion.

2 Introduction to OCL

OCL is a declarative expression-based language, i.e., evaluation of OCL expressions does not have side effects on the corresponding UML model. To integrate constraints into the visual UML modeling approach, invariants, pre- and postconditions are modeled as annotations and attached to the respective model elements in class diagrams.

Each OCL expression has a type. Besides user-defined model types (e.g., classes or interfaces) and some pre-defined basic types (e.g., `Integer`, `Real`, or `Boolean`), OCL also has a notion of object collection types (i.e., sets, ordered sets, sequences, and bags). Collection types are homogeneous in the sense that all elements of a collection have a common type. A new feature of the OCL 2.0 proposal is a built-in *tuple* type. Tuples are sequences of a fixed number of elements that can be of different types. A standard library is available that provides operations to access and manipulate values and objects of OCL types. In this context, predefined operations to access and select objects from object collections are of our special interest.

For an example, assume a UML model with classes `Machine` and `Buffer` and an association that connects those classes. We can navigate from class `Machine` to class `Buffer` via that association and make use of the role name `buffers` that is attached to the association-end at the `Buffer` side. The following invariant ensures that each object that is an instance of class `Machine` has at least one buffer:

```
context Machine
inv: self.buffers->notEmpty()
```

Let us briefly outline how to read this OCL invariant. The class identifier that follows the `context` keyword specifies the class for which the following expression should hold. The keyword `inv` specifies that this is an invariant, i.e., for each object of the context class the following expression must evaluate to true at any time. Note that an invariant may be violated during execution of an operation. In Sect. 4.3, we will therefore give a more accurate definition of what '*at any time*' means in this context. The (optional) keyword `self` refers to the object for which the constraint is evaluated. Attributes, operations, and associations can be accessed by dot notation, e.g., `self.buffers` results in a (possibly empty) set of instances of `Buffer`. The arrow operator indicates that a collection of objects is manipulated by one of the predefined OCL collection operations. For example, operation `notEmpty()` returns true if the accessed set is not empty.

3 Related work

Currently, commercial UML tools only provide limited support to specify and check OCL constraints. On the other hand, several OCL University tools are available. They implement syntax and type checks, dynamic constraint validation, test automation, and code generation of OCL constraints (see [31, 33] for an overview). OCL constraints are frequently used in the UML 1.5 specification on the UML metamodel level (M2) to define the static semantics of UML diagrams. Those so-called *well-formedness constraints* specify syntactical restrictions on diagrammatic model elements. Almost no industrial case study with applications of OCL is available. One recent study is documented in [45].

Due to the lack of an OCL metamodel in the UML standard, extensions of OCL have so far been defined purely on the concrete syntax level, in particular in the areas of business processes [8, 23], knowledge- and databases [5, 11], and real-time systems [41]. A semantics on the language definition level is not given so far. As soon as an OCL metamodel becomes part of the UML standard, we expect that extensions of OCL are developed by means of UML Profiles, just as it has already been done for other parts of UML, e.g., in the domain of modeling real-time systems [25, 40].

3.1 Temporal constraints in UML

There exist several approaches that either

(a) extend OCL for temporal constraints specification or
(b) investigate alternative means to express behavioral real-time constraints for UML diagrams.

Ramakrishnan et al. [30] extend the OCL syntax by additional grammar rules with unary and binary future-oriented temporal operators (e.g., *always* and *never*) to specify safety and liveness properties. Ziemann and Gogolla [44] introduce similar temporal operators based on a finite linear temporal logic. In that work, Richters' formal object model [31] is extended to provide a formal definition of system state sequences. However, it is left open *how* system state sequences are exactly derived. A similar approach has been published by Conrad and Turowski in the area of business modeling [8]. Their approach additionally considers past-temporal operators, but a formal semantics is not provided.

Distefano et al. [12] define *Object-Based Temporal Logic* (BOTL) in order to facilitate the specification of static and dynamic properties. BOTL is not directly an extension of OCL. It rather maps a subset of OCL into object-oriented Computational Tree Logic (CTL). Bradfield et al. [4] extend OCL by useful causality-based templates for dynamic constraints. A template consists of two clauses, i.e., the *cause* and the *consequence*. The cause clause starts with the keyword *after* followed by a Boolean expression, while the consequence

is an OCL expression prefaced by *eventually*, *immediately*, *infinitely*, etc. The templates are formally defined by a mapping to *observational mu-calculus*, a two-level temporal logic, using OCL on the lower level.

In the domain of real-time systems modeling, we know of three approaches for temporal constraint specification. Roubtsova et al. [34] define a UML Profile with stereotyped classes for dense time as well as parameterized specification templates for deadlines, counters, and state sequences. Each of these templates has a structural-equivalent dense-time temporal logics formula in Timed Computation Tree Logic (TCTL). Sendall and Strohmeier [41] introduce timing constraints on state transitions in the context of a restricted form of UML protocol state machines that define the temporal ordering between operations. Five time-based attributes on state transitions are proposed, e.g., (absolute) completion time, duration time, or frequency of state transitions. Using these attributes in an extended form of transition guards, they support exception handling, i.e., specification of actions to take when a timing constraint fails. Cengarle and Knapp [6] present OCL/RT, a temporal extension of OCL with modal operators `always` and `sometime` over event occurrences. These can be used to specify deadlines and timeouts of operations and reactions on received signals. Events are equipped with time stamps by introducing a metaclass `Time` with attribute `now` to refer to the time unit at which an event occurs. In turn, each object can access the set of currently queued events at each point in time.

3.2 Model checking

All current temporal OCL extensions with a formal semantics are due to application in formal verification by Model Checking. 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 (BDDs) allow verifications with up to 10^{120} states.

Given a parallel finite state machine (the model) and a temporal logic formula (the property specification), a Model Checking tool outputs either 'yes' if the model satisfies the formula or returns a counter example, i.e., one execution sequence of the model, which leads to a state contradicting the property specification.

Model representation is often based on Kripke structures, i.e., unit-delay temporal structures derived from finite state machines. A Kripke structure $M = (Pr, S, S_0, T, L)$ is a tuple with a set of atomic propositions Pr, a set of states S, a set of initial states $S_0 \subseteq S$, a transition relation $T \subseteq S \times S$ between states such that every state has a successor state, and a state labeling function $L : S \to \mathcal{P}(Pr)$, where $\mathcal{P}(Pr)$ denotes the power set of Pr.

Table 1. Semi-formal description of CCTL operators

Formula	Operator	Description
$g_0 \models p \quad (p \in Pr)$	Proposition	g_0 is valid in p, if $p \in L(s_0)$
$g_0 \models \phi \wedge \psi$	Conjunction	$g_0 \models \phi$ and $g_0 \models \psi$
$g_0 \models \phi \vee \psi$	Disjunction	$g_0 \models \phi$ or $g_0 \models \psi$
$g_0 \models \neg\phi$	Negation	g_0 is satisfied by $\neg\phi$ iff $g_0 \models \phi$ is false.
$g_0 \models \mathrm{EX}_{[a]} \phi$	Next	There exists a run $r = (g_0, \ldots, g_a, \ldots)$ such that $g_a \models \phi$
$g_0 \models \mathrm{EF}_{[a,b]} \phi$	Eventually	There exists a run $r = (g_0, \ldots, g_i, \ldots)$ and $i \in \mathbb{N}_0$, $a \le i \le b$, such that $g_i \models \phi$
$g_0 \models \mathrm{EG}_{[a,b]} \phi$	Globally	There exists a run $r = (g_0, \ldots, g_i, \ldots)$, such that for all $i \in \mathbb{N}_0$, $a \le i \le b$, holds $g_i \models \phi$
$g_0 \models \mathrm{E}(\phi \, \underline{\mathrm{U}}_{[a,b]} \, \psi)$	Strong Until	There exists a run $r = (g_0, \ldots, g_i, \ldots)$, and an $i \in \mathbb{N}_0$, $a \le i \le b$, such that $g_i \models \psi$ and for all $j \in \mathbb{N}_0$, $j < i$, holds $g_j \models \phi$
$g_0 \models \mathrm{E}(\phi \, \mathrm{U}_{[a,b]} \, \psi)$	Weak Until	There exists a run $r = (g_0, \ldots)$ and either (a) there exists an $i \in \mathbb{N}_0$, $a \le i \le b$, such that $g_i \models \psi$ and for all $j \in \mathbb{N}_0$, $j < i$, holds $g_j \models \phi$, or (b) for all $i \in \mathbb{N}_0$, $i \le b$ holds $g_i \models \phi$

There are extensions to basic model checking for the verification of real-time systems. One variation is defined by Kropf and Ruf in the context of the RAVEN model checker [36]. They extend Kripke structures to *Interval Structures* by adding a transition labeling function $I : T \to \mathcal{P}(\mathbb{N})$ with $[min,max]$-delay times. A state may be basically left at min-time and must be left after max-time. It is assumed 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*[2] $g = (s, v)$ of an Interval Structure $\mathfrak{I}$ is a state s associated with a clock value v. The set of all valid clocked states in $\mathfrak{I}$ is called G. To enable communication between a set of Interval Structures, an extension called *I/O-Interval Structures* has been proposed in [37].

Most BDD-based model checkers implement branching-time temporal logic specification for property specification. Temporal logic expresses information about states and future state transition paths. An execution path defines one possible future execution 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 frequently applied branching-time temporal logics is Computation Tree Logic (CTL). In CTL, temporal operators specify the ordering of states along future-oriented execution paths, e.g., operator F specifies that the associated sub-expression must *eventually* become true. 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.

Clocked CTL (CCTL) is an extension of CTL with time-bounded temporal operators over discrete time [35].

The syntax of CCTL's main operators is recursively defined by the following grammar:

$$\phi ::= p \mid \text{true} \mid \text{false} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$$
$$\mid \mathrm{EX}_{[a]} \phi \qquad \mid \mathrm{EF}_{[a,b]} \phi \qquad \mid \mathrm{EG}_{[a,b]} \phi$$
$$\mid \mathrm{E}(\phi \, \underline{\mathrm{U}}_{[a,b]} \, \phi) \quad \mid \mathrm{E}(\phi \, \mathrm{U}_{[a,b]} \, \phi)$$
$$\mid \mathrm{AX}_{[a]} \phi \qquad \mid \mathrm{AF}_{[a,b]} \phi \qquad \mid \mathrm{AG}_{[a,b]} \phi$$
$$\mid \mathrm{A}(\phi \, \underline{\mathrm{U}}_{[a,b]} \, \phi) \quad \mid \mathrm{A}(\phi \, \mathrm{U}_{[a,b]} \, \phi)$$

where $p \in Pr$ is a proposition, $a \in \mathbb{N}_0$, and $b \in \mathbb{N}_0 \cup \{\infty\}$. For the symbol ∞, we define $\forall i \in \mathbb{N}_0 : i < \infty$.

In contrast to classical CTL, temporal operators F (i.e., eventually), G (globally), U ('weak' until), and $\underline{\mathrm{U}}$ ('strong' until) are equipped with interval time-bounds $[a, b]$. Additional grammar rules, which are not listed here, allow that 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 (with $a \in \mathbb{N}$). 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 at execution time. Any arbitrary clocked state g_0 may be the starting point of a run. Table 1 shows semi-formal descriptions of the validation relation for a given Interval Structure $\mathfrak{I}$ and a given clocked state $g_0 = (s_0, v_0) \in G$. Note that ϕ and ψ stand for arbitrary CCTL (sub)formulae.

For reasons of brevity, Table 1 only gives the semantics for temporal operators with path quantifier E. Semantics for according formulae with path quantifier A (i.e., regarding *all* possible runs) can easily be derived, e.g., $\mathrm{AX}_{[a]}\phi$ is equivalent to $\neg\mathrm{EX}_{[a]}\neg\phi$. Another example is $\mathrm{AF}_{[a,b]}\phi$, which is equivalent to $\neg\mathrm{EG}_{[a,b]}\neg\phi$.

4 Extended object models

The OCL 2.0 proposal specifies a normative OCL semantics by means of UML concepts, structured into

---

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

three packages (cf. Fig. 1). The `Ocl-AbstractSyntax` package defines the general structure of OCL expressions based on user-defined classes and predefined OCL types. The `Ocl-Domain` package comprises the so-called *semantic domain*, i.e., the values OCL expressions may return as a result as well as rules to determine the result for a given expression. Note that these two packages reside on different levels in the 4-layer architecture of UML. The `Ocl-AbstractSyntax` package is on level M2, while the `Ocl-Domain` package is on level M1. The `Ocl-AS-Domain-Mapping` package that connects these two packages only specifies associations and no new classes. Note also that the `Ocl-AS-Domain-Mapping` cannot be assigned to any of the UML layers, neither M1 nor M2.

Additionally, an informative semantics of OCL is provided in the form of a mathematical set-theoretic approach, which is based on the work of Richters [31, 32]. It covers significant parts of the current OCL standard and new OCL 2.0 concepts like nested collections; but not yet concepts for OCL messages.

The drawback of both semantic descriptions – normative *as well as* informative – is that they do not integrate Statechart states as a basic concept, although it is permitted to apply operation `oclInState(s:OclState)` on the syntactical level in order to reason about the currently activated Statechart state(s) of an object. Consider an example in the context of a manufacturing scenario with a machine that has a limited input buffer to store items before processing them. The following invariant specifies that there must be at least one vacant input buffer position as long as the buffer object is in state `WaitingForDelivery`, i.e., a Statechart state representing the situation that delivery of an item is announced, but the item has not yet arrived.

```
context InputBuffer inv:
  self.oclInState(WaitingForDelivery)
  implies
  self.storedItems < self.maxItems
```

Basically, evaluation of an OCL expression is performed over a single *snapshot* or *system state*, i.e., a description of the current status of a model at run-time, including all objects with all their characteristic values.
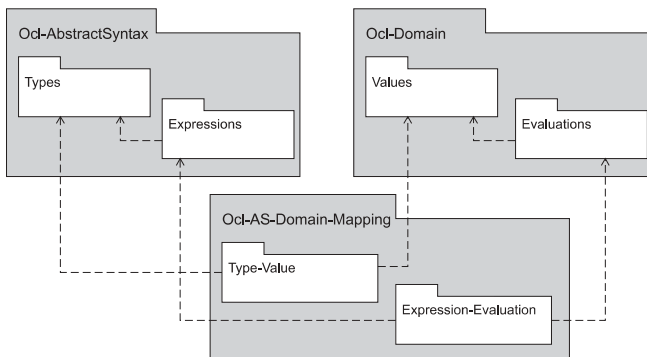
Note, however, that two system states have to be considered to evaluate operation postconditions when operator `@pre` is attached to objects or attributes, i.e., in addition to the system states after operation execution also values of the system state right before operation execution have to be regarded. Generally, evaluation of an OCL expression results in a value of the semantic domain.

Concerning Statechart states, the OCL 2.0 proposal simply assumes that according values are defined by a separate enumeration type called `OclState` in the semantic domain, such that they can be used as an argument of operation `oclInState (s:OclState)`. But in order to be able to evaluate an OCL expression that makes use of that operation, the system state must comprise the set of currently activated states of all objects. This aspect is still missing in both semantic descriptions of the OCL 2.0 proposal.

In the remainder of this section, we therefore formally define the syntax and semantics of *extended object models* that use Statecharts as a behavioral description of active classes. The formalization extends the definition of object models as presented by Richters in [31]. The following concepts have to be introduced:

– signals for classes with corresponding well-formedness rules,
– Statecharts and their association with classes,
– extension of the formal descriptor of a class,
– extension of the formal definition of a system state, and
– a definition of system state sequences (i.e., traces).

Section 4.1 formally defines the syntax of extended object models. The static semantics of extended object models is given in Sect. 4.2, where object identifiers, links, and state configurations are introduced that together build a description of the overall system state of a running system. A running system in this context is a particular instantiation of a given extended object model. In Sect. 4.3, we discuss how particular dynamic semantics of UML Statecharts affect evaluation of OCL invariants and state-related OCL operations.

4.1 Syntax

An *extended object model* is a tuple

$$\mathcal{M} \stackrel{def}{=} \langle CLASS, ATT, OP, SIG, SC, ASSOC,$$
$$\prec, \prec_{sig}, associates, roles, multiplicities \rangle$$

with

– a set $CLASS = ACTIVE \cup PASSIVE$ of active and passive classes,
– a set ATT of attributes, $ATT = \bigcup_{c \in CLASS} ATT_c$,
– a set OP of operations, $OP = \bigcup_{c \in CLASS} OP_c$,
– a set SIG of signals, $SIG \supseteq \bigcup_{c \in CLASS} SIG_c$,
– a set SC of Statecharts, $SC = \bigcup_{c \in ACTIVE} SC_c$,
– a set $ASSOC$ of associations between classes,

Fig. 1. OCL Packages as shown in [43, Chapter 5]

– generalization hierarchies $\prec$ for classes and $\prec_{sig}$ for signals, and
– functions *associates*, *roles*, and *multiplicities* that define a mapping for each element in $ASSOC$ to the participating classes, their according role names, and multiplicities, respectively.

In the following, the tuple elements of $\mathcal{M}$ are considered in more detail. For element names in $\mathcal{M}$, let $\mathcal{A}$ be an alphabet and $\mathcal{N} \subseteq \mathcal{A}^+$ a set of finite, non-empty names.

4.1.1 Types

We assume that there is a set $\Sigma \stackrel{def}{=} (T, \Omega)$, where T is a set of type identifiers and Ω a set of operations over types in T. T comprises a set of basic standard library types T_B, i.e., `Integer`, `Real`, `Boolean`, `String`, and `OclVoid`. The latter is a subtype of any other type that allows to operate with unknown values. The only value of `OclVoid` is called `OclUndefined` and is denoted in the following by $\perp$. We call the value set $I_{type}(t)$ represented by a type t the *type domain*.[3] In particular, we have

$$I_{type}(OclVoid) \stackrel{def}{=} \{\perp\}.$$

For convenience, we presume that $\perp$ is included in each type domain. Operations in Ω comprise common arithmetic operations, e.g., $+, -, *, /$ for Integer values. Moreover, so-called *collection types* are defined in Σ to manage collections of values, e.g., `Set(Integer)`.

4.1.2 Classes and their characteristics

A class is a description for a set of objects sharing the same characteristics, i.e., attributes, operations, signals, and associations. Note that associations are separately defined in the set $ASSOC$.

Definition 1. *(classes and types)*
$CLASS$ is a finite set of names, $CLASS \subseteq \mathcal{N}$. $CLASS$ is the union of two disjoint sets of active and passive classes, i.e., $CLASS \stackrel{def}{=} ACTIVE \cup PASSIVE$. Active classes specify entities capable of dynamic behavior, which is specified by an associated Statechart (see Definition 4).[4]

Each class $c \in CLASS$ induces a type $t_c \in T_C \subset T$ having the same name as the class. A value $val \in I_{type}(t_c)$ of a type $t_c \in T_C$ refers to an object of the corresponding class $c \in CLASS$.

In the remainder of this article, let $c \in CLASS$ be a class and $t_c \in T_C$ be the type of class c.

---

[3] Type domain refers to the values of the *semantic domain* in the normative OCL semantics.
[4] *UML allows multiple Statecharts to be applied to a single class. We here require that there is at most one Statechart SC_c for each $c \in ACTIVE$. Note that it is possible to combine multiple Statecharts to an equivalent single concurrent Statechart.*

Each class c is associated with a set ATT_c of attributes that describe characteristics of their objects. An attribute has a name $a \in \mathcal{N}$ and a type $t \in T$ that specifies the domain of attribute values. Though attribute names of a class must be pairwise distinct, attributes with the same name may appear in several classes, which are not related by generalization. A class may also be associated with a number of operations. Operations are used to describe behavioral characteristics of objects. The behavior might be specified by an associated Statechart diagram, but we here only consider *signatures* of operations that declare its interface.

Definition 2. *(operations)*
The operations of class c are defined by a set OP_c of operation signatures,

$$OP_c \stackrel{def}{=} \{(\omega : t_c \times t_1 \times \ldots \times t_n \to t) \mid \\ \omega \in \mathcal{N}, n \in \mathbb{N}_0, \ and \ t, t_1, \ldots, t_n \in T\}.$$

Symbol ω determines the operation name, and parameter t_c denotes the type of c to which operation ω is applied. Note that UML generally allows operation parameters to be of kind `in`, `out`, `inout`, or `result`. The OCL 2.0 proposal also considers these parameter kinds. It only assumes that at most one parameter of kind `result` is specified [43, App. A.3.2]. If neither a result type nor any `inout` or `out` parameters are specified for an operation, we set the result type t to the predefined type `OclVoid` $\in T$. If there are `inout` or `out` parameters specified, the operation result type is a tuple in which the relevant parameter values appear in their specified order, including the result value (if any) as the last element.

Richters did not consider asynchronous signals that are communicated between objects in his formal model so far. Reactions on signals received by an object *obj* are specified by a Statechart associated with the class to which *obj* belongs to. Consequently, when integrating Statecharts into the object model, signals now also have to be regarded as well. In UML, signals are classifiers, i.e., signals are generalizable model elements defined independently of the classes handling them. The set SIG in the model description defines all signals of a model. As we support generalization of signals, SIG is a superset of the individual signal sets SIG_c. The set SIG_c of signals that can be handled by objects of a class c is specified by so-called *receptions* [26, Sect. 3.26.6]. Note that signals can only be handled by instances of active classes, as passive classes do not have associated Statecharts.

Definition 3. *(signals)*
The signals that can be handled by instances of a class $c \in ACTIVE$ are defined by the set SIG_c of signal receptions,

$$SIG_c \stackrel{def}{=} \{(\omega : t_c \times t_1 \times \ldots \times t_n) \mid \\ \omega \in \mathcal{N}, n \in \mathbb{N}_0, \ and \ t_1, \ldots, t_n \in T\}.$$

Symbol ω denotes the signal name, and t_c refers to the type of c to which signal ω is applied. As signals are asyn-

chronous, no return value is expected, such that all signal parameters are all input parameters.

4.1.3 Abstract syntax of Statecharts

The UML 1.5 StateMachine package provides concepts for modeling discrete behavior by means of finite state-transition systems [26, Sect. 2.12]. The provided state machine formalism is an object-based variant of Harel's Statecharts [21]. Though state machines are applicable to various model elements within UML, the graphical form of *UML Statechart Diagrams* is most frequently used to model the reactive behavior of objects. The UML standard only informally defines the dynamic semantics of Statecharts by natural language and consequently has to leave open some semantics, e.g., the dispatching policy for selecting events from the implicit event queue to perform the next so-called *run-to-completion step* (RTC-step). Basically, an RTC-step consists of (a) selecting and taking an event from an (implicit) queue of perceived, waiting events, (b) determining the maximum set of transitions to take in the Statechart, and (c) subsequently executing the actions associated with exited states, determined transitions, and entered states. In recent years, numerous approaches have been published to formally define the dynamic semantics of UML Statecharts. A excellent overview is provided in [2].

The abstract syntax defined below comprises all relevant information to fully describe a *configuration* of a Statechart, i.e., a complete description of what is informally known in UML by an *active state configuration* reached after completion of an RTC-step [26, Sect. 2.12.4.3]. We here do not make assumptions of *how* an RTC-step is performed to get to the next configuration. This must be formally defined in the dynamic semantics a particular approach is taking.

Definition 4. *(abstract syntax of Statecharts)*
Let $c \in CLASS$ be a class. Each $c \in ACTIVE$ has an associated Statechart SC_c representing the reactive behavior of instances of c.

$$SC_c \stackrel{def}{=} \langle S_c, EVTS_c, GUARDS_c, ACTS_c, TR_c,$$
$$internalTrans_c, substates_c, entry_c,$$
$$exit_c, doActivity_c, deferrableEvents_c \rangle$$

For all $c \in PASSIVE$, we set $SC_c \stackrel{def}{=} \varnothing$.

To keep the definition concise, we omit the class annotator c for Statechart components in the remainder of this definition. The components of a Statechart SC are then defined as follows.

1. $S \subseteq \mathcal{N}$ is a set of states. S is the union of the following disjoint sets
   - pseudo states *Pseudo*, consisting of seven disjoint sets (a) initial states *Init*, (b) shallow history states *History*, (c) deep history states *DeepHistory*, (d) merging states *Join*, (e) splitting

states *Fork*, (f) static conditional branch states *Junction*, and (g) dynamic conditional branch states *Choice*,
   - synchronization states *Synch*,
   - simple states *Simple*,
   - composite states *Composite*, composed of the two disjoint sets of sequential composite states *Xor* and orthogonal composite states *And*,
   - and final states *Final*.

   We refer to [26, Sect. 2.12.2] for details about these states. For convenience, we define

   $$Proper \stackrel{def}{=} And \cup Xor \cup Simple.$$

2. $EVTS \subseteq EXPR_{Evts}$ is a set of events. We assume that there is an expression language $EXPR_{Evts}$ available to specify events such as operation calls, signals, timers, etc.
3. $GUARDS \subseteq EXPR_{Guards}$ is a set of conditions. We assume a language $EXPR_{Guards}$ for the definition of Boolean expressions.[5]
4. $ACTS \subseteq EXPR_{Acts}$ is a set of actions. We assume an expression language $EXPR_{Acts}$ to specify operational actions such as assignments, operation calls, signals, etc.
5. $TR \subseteq (S \setminus Final) \times EVTS \times GUARDS \times ACTS \times (S \setminus Init)$ is a set of transitions. A transition connects a source state $s \in S \setminus Final$ and a destination state $s' \in S \setminus Init$, may have a trigger event $e \in EVTS$, a guard condition $g \in GUARDS$, and an action expression $a \in ACTS$.
6. $internalTrans : Proper \rightarrow \mathcal{P}(EVTS \times GUARDS \times ACTS)$ gives the set of *internal transitions* for a given state $S \in Proper$. Internal transitions semantically differ from self-transitions. When triggering an internal transition in a state s, the exit- and entry-actions of s are not executed.
7. $substates : Composite \rightarrow \mathcal{P}(S)$ gives all substates of a state, such that
   (a) there is a unique state $top \in Composite$ such that $\forall s \in Composite : top \notin substates(s)$,
   (b) $\forall s \in And : substates(s) \subseteq Composite$,[6]
   (c) $\forall s \in Composite \setminus \{top\}$ there is exactly one path
   $$\langle s_1, \ldots, s_n \rangle \in Composite^n,$$
   with $s_1 = top \land s_n = s \land s_{i+1} \in substates(s_i)$ for $1 \leq i \leq n-1$.
8. Functions $entry, doActivity, exit : Proper \rightarrow ACTS$ give the executed actions when a state is entered, active, or left, respectively.

---

[5] In this context, Boolean OCL expressions are frequently applied.

[6] This is a well-formedness rule of the UML standard (see [26, Sect. 2.12.3.1]). In many alternative formal syntax definitions, even $s' \in Xor$ is required in this case, leading to a *normal form* of alternating Xor- and And-states in the state hierarchy.

9. $deferrableEvents : Proper \rightarrow \mathcal{P}(EVTS)$ gives the set of events to be retained for later consumption.

Definition 4 covers most of the abstract Statechart syntax as defined in the official UML 1.5 specification [26, Sect. 2.12.2]. Only a few details are left out, e.g., local variables and the bound of synch states, but can easily be added to the definition if required. We also ignore *submachine states* and *stub states* without loss of generality, as submachine states are a syntactical convenience to represent a 'call' to a another state machine as a 'subroutine' using stub states as entry and exit points. A submachine state is semantically equivalent to a composite state, and we can assume that all these states have already been copied into SC, such that all submachine states and stub states are eliminated. Additionally, several syntactical constraints have to be considered; more than 30 well-formedness rules are listed for the abstract syntax in the official UML 1.5 specification [26, Sect. 2.12.3]. Most of these constraints are already implicitly included in Definition 4 above, while the remaining ones can easily be translated into that context.

4.1.4 Associations

Associations are used to model structural relationships between classes. We here do not provide formal definitions for associations, role names, association multiplicities, and their syntactical restrictions, as they can be applied unchanged from Richters' original work [31]. We define function

$$navEnds : CLASS \rightarrow \mathcal{P}(\mathcal{N})$$

to denote the set of role names that can be directly accessed from a given class by navigating along the associations this class participates in.

4.1.5 Generalization

By generalization, we refer to a relationship between two classes, in which a general class is specialized into a more specific class.

Definition 5. *(generalization, child and parent classes)*
A generalization $\prec$ over classes is an irreflexive partial order on $CLASS$, i.e., $\prec$ defines an irreflexive, anti-symmetric, and transitive relation. Pairs in $\prec$ describe generalization relationships between two classes.

For $c_1, c_2 \in CLASS$ with $c_1 \prec c_2$, c_1 is called a child class of c_2, and c_2 is called a parent class of c_1.

Function parents gives the set of all transitive parents (or: ancestors) of a given class:

$$parents : \begin{cases} CLASS \rightarrow \mathcal{P}(CLASS) \\ c \mapsto \{c' \mid c' \in CLASS \land c \prec c'\} \end{cases}$$

A child class transitively inherits characteristics (attributes, operations, signals, and associations) of its parent classes.

Correspondingly, the generalization hierarchy $\prec_{sig}$ defines an irreflexive partial order on SIG. As a signal can be specified as the child of another signal, reception of that child signal may also trigger any transition in a Statechart that depends on any of its ancestor signals.

The set of characteristics defined in a class together with its inherited characteristics is called a *full descriptor of a class*. Thus, the complete set of attributes of a class c is defined by

$$ATT_c^* \stackrel{def}{=} ATT_c \cup \bigcup_{c' \in parents(c)} ATT_{c'}.$$

The complete sets OP_c^*, SIG_c^*, and $navEnds^*(c)$ of operations, signals, and navigable role names are defined correspondingly. Concerning Statecharts and their inheritance, we assume that there is exactly one Statechart for each active class that complies to some *inheritance policy*. More details about Statechart inheritance are given in Sect. 4.2.1.

Definition 6. *(full descriptor of a class)*
The full descriptor of a class $c \in CLASS$ is a tuple

$$FD_c \stackrel{def}{=} \langle ATT_c^*, OP_c^*, SIG_c^*, SC_c, navEnds^*(c) \rangle$$

containing all attributes, operations, signals, navigable role names, and – in the case of an active class – the associated Statechart.

The UML standard requires that characteristics of a full descriptor must be distinct, i.e., a class may not define an operation, attribute, or role name that is already defined in one of its ancestor classes. Such constraints are already formally captured by Richters in [31], and we here only list those well-formedness rules that have to be added for the consideration of signals:

1. A signal may only be defined once in a full class descriptor. The first parameter of a signal signature indicates the class in which the signal is defined. The following condition guarantees that each signal in a full class descriptor is defined in a single class.

$$\forall (\omega : t_c \times t_1 \times \ldots \times t_n) \in SIG_c^*,$$
$$\forall (\omega' : t_{c'} \times t_1' \times \ldots \times t_n') \in SIG_c^* :$$
$$(\omega = \omega' \land t_1 = t_1' \land \ldots \land t_n = t_n') \implies t_c = t_{c'}.$$

2. Operation and signal names (in combination with the corresponding parameters) must be pairwise distinct.

$$\forall (\omega : t_c \times t_1 \times \ldots t_n \rightarrow t) \in OP_c^*,$$
$$\forall (\omega' : t_{c'} \times t_1 \times \ldots t_n) \in SIG_c^* : \ \omega \neq \omega'.$$

Note that types $t_1, \ldots, t_n$ are fixed for ω' by the parameter types of ω.

3. For syntactical consistency among Statecharts and class definitions, for each operation call expression in Statechart SC_c (either as an event $evt \in EVT_c$ or as

an action $act \in ACT_c$), a corresponding operation signature must be defined in OP_c^*. Correspondingly, each signal in EVT_c or ACT_c must have a corresponding signal signature in SIG_c^*.

As we abstract from a particular expression syntax of $EVTS_c$ and ACT_c, a formalization is not provided here.

4.2 Configurations and system state

The domain of a class $c \in CLASS$ is the set of objects of this class and all of its child classes. Objects are referred to by object identifiers that are unique in the context of the whole system.

Definition 7. *(object identifiers and domain of a class) The set of object identifiers of a class $c \in CLASS$ is defined by an infinite set $oid(c) \stackrel{def}{=} \{\underline{oid}_1, \underline{oid}_2, \dots\}$. The domain of a class $c \in CLASS$ is defined as*

$$I_{class}(c) \stackrel{def}{=} \bigcup_{c' \in CLASS \text{ with } c' \prec c \vee c'=c} oid(c').$$

In the remainder of this article, we do not distinguish between objects and their identifiers, i.e., each object is uniquely determined by its identifier and vice versa.

4.2.1 Statechart inheritance

While the problem of consistency among generalization of classes and inheritance of characteristics w.r.t. attributes and operations has been studied extensively for object-oriented languages, *consistency among inheritance of behavior* in object-oriented design notations like UML has received less attention. Different notions for behavioral consistency have been identified in this context, e.g., [15, 39, 42]. Corresponding definitions make use of the dynamic execution of Statecharts by traces, which are execution runs through (the processes derived from) Statecharts.

Weak invocation consistency guarantees that each trace of the Statechart for the superclass is also contained in the set of traces of the Statechart for the subclass. In other words, a sequence of states (or: activities) performable on instances of a superclass can also be performed on instances of a subclass. Second, *strong invocation consistency* guarantees the latter property even if states (or: activities) specified for the subclass are arbitrarily inserted in that sequence. Finally, in *observation consistency*, the Statechart of the superclass specifies an *upper bound* to the behavior of the subclasses. It is guaranteed that every trace of an instance of a subclass is identified as a trace of the superclass, when states, events, and activities added to the subclass are neglected.

UML 1.5 provides an informal description of three different inheritance policies for state machines [26, Sect. 2.12.5]. *Subtyping* requires that a state in the sub-

class retains all its transitions. These transitions may lead to the same state or a new substate of that state (i.e., strengthening of the transition postcondition), and guard conditions may be weakened by adding disjunctions. (i.e., weakening of transition preconditions) This corresponds to *weak invocation consistency* and complies to the substitutability principle. The other two policies provided by UML 1.5 support neither observation nor invocation consistency and are instead oriented towards coding and inheritance issues; UML's *strict inheritance* is intended to reuse implementation rather than preserving behavior, and *general refinement* basically places no restrictions on Statechart inheritance.

If a class c has multiple superclasses, the default Statechart for c consists of all the Statecharts of its superclasses as orthogonal regions. This may be overridden through a specific Statechart inheritance if required.

4.2.2 State configurations

As a result from the previous paragraph, we assume in the following that an extended object model $\mathcal{M}$ complies to some predefined policy of Statechart inheritance. This means that for each active class $c \in ACTIVE$ there is a Statechart specification SC_c available that is 'consistent' with the Statechart specifications of the superclasses of c.

In a Statechart with composite and concurrent states, the term 'current state' cannot be applied without any disambiguities, as more than one state can be active at the same time. Consequently, UML 1.5 provides the notion of *active state configurations* [26, Sect. 2.12.4.3] as follows. If the Statechart is in a simple state that is contained in a composite state, then all the composite states that (transitively) contain the simple state are also active. Furthermore, as composite states in the state hierarchy may be concurrent, the currently active states are actually represented by a tree of states starting with the single state top_c at the root down to individual simple leaf states $s_i \in Simple_c$. Such a state tree is referred to as a state configuration in UML 1.5. In Definition 8, we give the corresponding formal definition of state configurations. But first, we define a convenience function $superstate_c$ that gives the direct superstate of a state $s \in S_c$:

$$superstate_c : \begin{cases} S_c \rightarrow Composite_c \\ s \mapsto \begin{cases} s', \text{ if } \exists s' \in Composite_c \\ \quad \text{with } s \in substates_c(s'), \\ \varnothing, \, else \end{cases} \end{cases}$$

UML 1.5 does not consider final states in state configurations. In contrast, we include final states in the following definition for state configurations, as they might be active after an RTC-step. However, a final state that is a direct child state of top_c is not part of any configuration, since entering that state is equivalent to termination (or:

destruction) of the corresponding object. Additionally, we explicitly exclude *immediate states*. Immediate states are proper states that are directly run through in an RTC-step, as they do not have outgoing transitions that have to wait for a triggering event. Consequently, they can never be part of an active state configuration after completion of an RTC-step. We here leave out a formal definition and simply refer to set $Immediate_c$ to denote the set of all immediate proper states of a Statechart SC_c.

Furthermore, we make use of the following help sets for classes $c \in ACTIVE$:

$$
\begin{aligned}
ProperStay_c &\overset{def}{=} Proper_c \setminus Immediate_c \\
Stay_c &\overset{def}{=} ProperStay_c \cup \\
&\quad \{f \in Final_c \mid f \notin substates_c(top_c)\}, \\
Basic_c &\overset{def}{=} (Simple_c \setminus Immediate_c) \cup \\
&\quad \{f \in Final_c \mid f \notin substates_c(top_c)\}.
\end{aligned}
$$

Definition 8. *(state configurations with respect to state s)*
Let $c \in ACTIVE$ and SC_c be the Statechart for c. A state configuration $\mathcal{C}$ with respect to a state s is a maximal set of states that the Statechart can be simultaneously in, taking state s as the root. Function cfg_c that maps a state $s \in ProperStay_c$ to the set of configurations $\mathcal{C}$ with respect to s is defined by

$$
cfg_c : \begin{cases}
ProperStay_c \to \mathcal{P}(\mathcal{P}(Stay_c)) \\
s \mapsto \{\, \mathcal{C} \in \mathcal{P}(Stay_c) \mid s \in \mathcal{C} \,\wedge \\
\quad \forall s' \in \mathcal{C} \cap And_c : substates_c(s') \subseteq \mathcal{C} \,\wedge \\
\quad \forall s' \in \mathcal{C} \cap Xor_c : |substates_c(s') \cap \mathcal{C}| = 1 \\
\quad \wedge \, \forall s' \in \mathcal{C} \setminus \{s\} : superstate_c(s') \in \mathcal{C} \,\}
\end{cases}
$$

Definition 9. *(state configuration)*
The set $I_{SC}(c)$ of overall state configurations for a class $c \in ACTIVE$, which are state configurations with respect to the top state top_c, is determined by $cfg_c(top_c)$.
For convenience, we define $I_{SC}(c)$ for all $c \in CLASS$ by

$$
I_{SC}(c) \overset{def}{=} \begin{cases} cfg_c(top_c), & if\,c \in ACTIVE, \\ \varnothing, & if\,c \in PASSIVE. \end{cases}
$$

By definition, each state configuration induces a state tree. But to uniquely determine a state configuration, it is sufficient to have information about terminal states, i.e., the simple and final states.
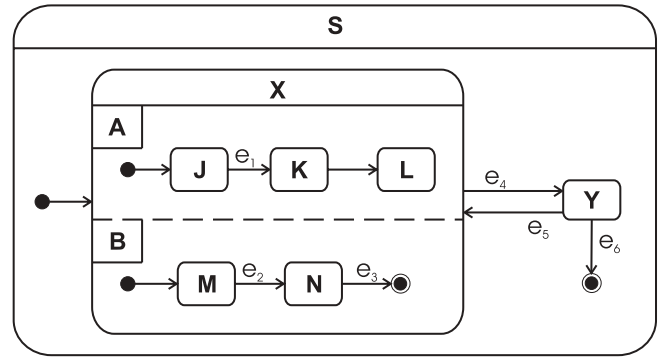
Definition 10. *(basic state configurations)*
Let $c \in ACTIVE$ and SC_c be the Statechart for c. Let $s \in ProperStay_c$ be a state and let $\mathcal{C} \in cfg_c(s)$ be a state configuration with respect to s. The set

$$
B_{\mathcal{C}} \overset{def}{=} \mathcal{C} \cap Basic_c
$$

is called a basic state configuration (with respect to $\mathcal{C}$). The set B_s of all basic configurations with respect to s is then defined by

$$
B_s \overset{def}{=} \{B_{\mathcal{C}} \mid \mathcal{C} \in cfg_c(s)\} \subseteq \mathcal{P}(\mathcal{P}(Basic_c)) .
$$

Proper States:	{ S,X,Y,A,B,J,K,L,M,N }
Final States:	{ S::FinalState,B::FinalState }
Immediate States:	{ K }
State Configurations:	{ {S,X,A,B,J,M},{S,X,A,B,J,N}, {S,X,A,B,J,B::FinalState}, {S,X,A,B,L,M},{S,X,A,B,L,N}, {S,X,A,B,L,B::FinalState}, {S,Y} }
Basic Configurations:	{ {J,M},{J,N},{J,B::FinalState}, {L,M},{L,N},{L,B::FinalState}, {Y} }

Fig. 2. Statechart example

Note that the following condition holds (cf. [28, Lemma 1]):

$$
\forall s \in ProperStay_c, \forall B_{\mathcal{C}} \in B_s : \\
superstate^*(B_{\mathcal{C}}) \cap substates^*(s) = \mathcal{C} .
$$

In other words, given a basic state configuration $B_{\mathcal{C}}$, we can uniquely determine the state configuration $\mathcal{C} = cfg_c(s)$ with respect to a state s. For reasons of brevity, we omit a formal definition of function

$$
superstate^* : \mathcal{P}(S_c) \to \mathcal{P}(ProperStay_c)
$$

here. Basically, that function gives the set of transitive superstates on a given set of states (including that given set of states). Function $substates^* : ProperStay_c \to \mathcal{P}(Stay_c)$ in turn gives the set of transitive substates on a given state (including this state).

Figure 2 gives a Statechart example with corresponding basic state configurations. All proper states except immediate state K have an outgoing transition with a specified event e_i, $1 \leq i \leq 6$. As UML does not provide a textual equivalent for final states, we use the parent state name, double colons, and the keyword `FinalState` to syntactically refer to final states. Note that `S::FinalState` is not part of the configuration set (cf. usage of set $Stay_c$ in Definition 8).

4.2.3 System state

In the following, we call a particular instantiation of an extended object model a *system*. A system is in different states as it changes over time, i.e., the (number of) objects, their attribute values, Statechart configurations, and other characteristics change when actually executing the system. But it still has to be defined what a single

system state exactly consists of. It is important to point out here that different notions of a system state are generally possible, depending on the scope of model analysis one wants to perform. In the original work on object models [31], a system state is a tuple consisting of three parts:

– the current set of objects,
– their attribute values, and
– the current links that connect the objects.

A semantics of a large part of standard OCL expressions is defined over such systems states in [31, Sect. 5.2]. However, as Statecharts are not considered in that work, state-related operations such as `oclInState (s:OclState)` cannot be handled.

In our approach, we additionally investigate *sequences* of system states, i.e., we are going to perform an analysis over possible future system states and thus reason about evolution of Statechart states. For this, we need a concise notion of *system state sequences* that also covers Statechart configurations. In order to be able to formally define such sequences, we need to define which operations are to be executed next (for operation preconditions) and which operations terminate later (for operation postconditions). In this context, we adopt ideas of [44] to formalize currently executed operations and define additional functions to capture that information.

Definition 11.   *(system state)*
A system state for an extended object model $\mathcal{M}$ is a tuple

$$\sigma(\mathcal{M}) \stackrel{def}{=} \langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \\ \Sigma_{CONF}, \Sigma_{OP}, \Sigma_{PARAM} \rangle$$

where

1. $\Sigma_{CLASS} \stackrel{def}{=} \bigcup_{c \in CLASS} \Sigma_{CLASS,c}$.
   The finite sets $\Sigma_{CLASS,c}$ contain all objects of a class $c \in CLASS$ existing in the system state, i.e.,

   $$\Sigma_{CLASS,c} \subseteq oid(c) \subseteq I_{class}(c).$$

   For further application, we define $\Sigma_{ACTIVE,c}$ for active and $\Sigma_{PASSIVE,c}$ for passive classes correspondingly.
2. The current attribute values are kept in set Σ_{ATT}. It is the union of functions $\sigma_{ATT,a} : \Sigma_{CLASS,c} \to I_{type}(t)$, where $a \in ATT_c^*$. Each function $\sigma_{ATT,a}$ assigns a value to a certain attribute of each object of a given class $c \in CLASS$.
3. $\Sigma_{ASSOC} \stackrel{def}{=} \bigcup_{as \in ASSOC} \Sigma_{ASSOC,as}$ comprises the finite sets $\Sigma_{ASSOC,as}$ that contain links that connect objects, where

   $$\forall as \in ASSOC : \Sigma_{ASSOC,as} \subseteq I_{ASSOC}(as).$$

   We refer to [31] for detailed information about links, i.e., elements of $I_{ASSOC}(as)$, and formalization of multiplicity specifications.

4. The current Statechart configurations are kept by

   $$\sigma_{CONF} \stackrel{def}{=} \\ \bigcup_{c \in ACTIVE} \{\sigma_{CONF,c} : \Sigma_{ACTIVE,c} \to I_{SC}(c)\}.$$

   Each function $\sigma_{CONF,c}$ assigns a state configuration with respect to the corresponding top state top_c to each object of a given class $c \in ACTIVE$.
5. Let $\mathcal{ID}$ be an infinite enumerable set, e.g., $\mathcal{ID} = \mathbb{N}$. The set of currently executed operations is denoted by

   $$\Sigma_{OP} \stackrel{def}{=} \\ \bigcup_{c \in CLASS} \{\sigma_{OP,c} : \Sigma_{CLASS,c} \times OP_c \to \mathcal{P}(\mathcal{ID})\}.$$

   Each function $\sigma_{OP,c}$ gives a set of unique identifiers $\in \mathcal{ID}$ that represents all currently executed operations for a given object $\underline{oid}$ and operation name op. At the starting point of an operation execution, a unique identifier $\in \mathcal{ID}$ is associated with that operation execution. We require that the associated identifier must not change until the execution of that operation terminates.
6. $\Sigma_{PARAM} \stackrel{def}{=}$

   $$\bigcup_{c \in CLASS} \{\sigma_{PARAM,c} : \Sigma_{CLASS,c} \times OP_c \times \mathcal{ID} \to \\ I_{type}(t_1) \times \ldots \times I_{type}(t_n) \times I_{type}(t) \}$$

   is a set of functions that gives the parameter values of each of the currently executed operations. For each $c \in CLASS$, we define $\sigma_{PARAM,c}$ as follows, where $op = (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP_c$:

   $$\sigma_{PARAM,c}(\underline{oid}, op, id) \mapsto \\ \begin{cases} \langle val(t_1), ..., val(t_n), val(t) \rangle, & \text{if } id \in \sigma_{OP,c}(\underline{oid}, op) \\ \varnothing, & \text{otherwise} \end{cases}$$

   In the definition above, $val(t_j) \in I_{type}(t_j)$ denotes an arbitrary value defined for type $t_j \in T$, $1 \le j \le n$. The same holds for $val(t) \in I_{type}(t)$. If an operation is not returning a result, the result type t of operation op is `OclVoid`. In that case, we set $val(t) = \perp$.

   Of course there are additional Statechart characteristics that could also be taken into account to be part of a system state, e.g., event queues and changes occurring to them, additional information required for re-entering composite states via history states, etc. While this can make sense in some specific approaches, the definition above is sufficient for reasoning about currently activated states and executed operations.

4.2.4 Semantics of operation oclInState

According to the OCL 2.0 proposal, the operation signature of `oclInState` is defined by

$$oclInState : OclAny \times OclState \to Boolean,$$

where the domain of *OclAny* is formally defined by

$$I_{type}(OclAny) = \left(\bigcup_{t \in T_B \cup T_E \cup T_C} I_{type}(t) \right) \cup \{\bot\}.$$

In this context, T_B, T_E, and T_C represent basic OCL types, enumeration types, and types induced by user-defined classes, respectively. We formally define the domain of type `OclState` by

$$I_{type}(OclState) = \left(\bigcup_{c \in ACTIVE} Stay_c \right) \cup \{\bot\}.$$

For an operation $op = (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP_c$, a semantics is generally defined by a total function with signature

$$I_{op} : I_{type}(t_c) \times I_{type}(t_1) \times \ldots \times I_{type}(t_n) \to I_{type}(t).$$

Accordingly, we define the semantics of operation `oclIn-State` on a given system state $\sigma(\mathcal{M})$, a given object $\underline{oid} \in \Sigma_{CLASS,c}$, and a state name $s \in I_{type}(OclState)$ by

$$I_{(oclInState:OclAny \times OclState \to Boolean)}(\underline{oid}, s) \stackrel{def}{=}$$

$$\begin{cases} true, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \wedge s \in Stay_c \\ & \quad \wedge s \in \sigma_{CONF,c}(\underline{oid}), \\ false, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \wedge s \in Stay_c \\ & \quad \wedge s \notin \sigma_{CONF,c}(\underline{oid}), \\ \bot, & \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c}, \\ \bot, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \wedge s \notin Stay_c \cup \{\bot\}, \\ \bot, & \text{if } s = \bot. \end{cases}$$

Note here that `oclInState` returns $\bot$ when $\underline{oid}$ is a passive object or when state s is not defined in Statechart $Stay_c$ of an active class $c \in ACTIVE_c$. Alternatively, we could have chosen to return $false$ instead in these cases. Neither the UML 1.5 standard nor the OCL 2.0 proposal give any information about this issue.

4.3 Dynamic semantics

We can now consider sequences of system states (or: *traces*). At this point, we have to decide and formally define a *valid* trace, i.e., when a new system state is appended to the trace at execution time. In the context of checking OCL constraints, we are, for instance, not interested in every single attribute value change that occurs during execution of an operation. Instead, we are interested in system states in which an operation has been completed or a signal has been consumed.

In the simplest case, e.g., when (an implementation of) the system is executed on a single CPU, there is a clear temporal order of operations. But when (the implementation of) the system is distributed, we have a partial order between configurations of different objects. This problem can be treated in an ideal case by introducing a *global clock* that allows for a global view on the system.

Definition 12. *(trace)*
A well-defined system state sequence called trace *for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states as defined in Definition 11,*

$$trace(\mathcal{M}) \stackrel{def}{=} \langle \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \ldots, \sigma(\mathcal{M})_{[i]}, \ldots \rangle$$

The first trace element $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state. Given a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, the next system state $\sigma(\mathcal{M})_{[i+1]}$ is added to the trace when for at least one object in $\Sigma_{CLASS[i]}$[7] one of the following happens at execution time:

- *an operation is called,*
- *an operation has terminated, or*
- *a new Statechart state configuration is reached.*

In the following, let

$$X_{[i]} = \bigcup_{op \in OP_c} \sigma_{OP,c}(\underline{oid}, op)_{[i]}$$

denote the operation identifier set of a given object $\underline{oid}$ in a system state $\sigma(\mathcal{M})_{[i]}, i \in \mathbb{N}_0$. We presume that the following restrictions apply to traces:

1. Two adjacent sequence elements may differ in at most one begin or end (i.e., termination) of operation execution *per object*.
   More formally, for each pair of adjacent system states $\sigma(\mathcal{M})_{[i]}$ and $\sigma(\mathcal{M})_{[i+1]}$ in $trace(\mathcal{M})$, it must hold that

$$\forall c \in CLASS, \forall \underline{oid} \in \Sigma_{CLASS,c[i]} :$$
$$\underline{oid} \in \Sigma_{CLASS,c[i+1]} \implies$$
$$abs(|X_{[i]} \cup X_{[i+1]}| - |X_{[i]} \cap X_{[i+1]}|) \leq 1$$

2. Each operation call occurring in the trace must eventually be terminated, i.e., for each system state $\sigma(\mathcal{M})_{[i]}, i \in \mathbb{N}_0$, it must hold that

$$\forall c \in CLASS, \forall \underline{oid} \in \Sigma_{CLASS,c[i]} :$$
$$\forall execOp \in X_{[i]} \; \exists j \in \mathbb{N}, j > i : execOp \notin X_{[j]}$$
$$\wedge \; \forall k \in \{i, \ldots, j\} : \underline{oid} \in \Sigma_{CLASS,c[k]}$$

The formula above additionally requires that an object must not be destroyed when one of its operations is still executed.

3. Values of operation parameters must not change until termination of the operation. The values of parameters of kind `inout` and `out` may change exactly when the operation call terminates. Values of parameters of kind `in` must not even change at termination of the operation. For operation return types $t \neq OclVoid$, the value of the (implicitly defined) result variable changes from $\bot$ to a well-defined value $val(t) \in I_{Type}(t)$ when the operation call terminates.

---

[7] *In the remainder, we are using the $[i]$-annotation for the components and functions defined in $\sigma(\mathcal{M})_{[i]}, i \in \mathbb{N}_0$, correspondingly.*

Traces as defined above should be seen as a rather general approach to capture those parts of the system runtime information that is necessary to reason about (sequences of) system states.

When to Check Invariants. In addition to checking invariants for all objects that exist in the initial state $\sigma(\mathcal{M})_{[0]}$, we require to check invariants on an object each time when the object status changes, i.e., we evaluate invariants for object $\underline{oid} \in \Sigma_{CLASS,c[i]}$ at location i of $trace(\mathcal{M})$, $i \geq 1$, when it holds that

$$\underline{oid} \notin \Sigma_{CLASS,c[i-1]} \lor \quad // \text{ new object}$$
$$X_{[i]} \neq X_{[i-1]} \lor$$
$$\sigma_{CONF,c}(\underline{oid})_{[i]} \neq \sigma_{CONF,c}(\underline{oid})_{[i-1]}.$$

The first condition states that the object has been newly created. Note that this coincides with an operation call of some other object. This is considered in the second condition, i.e., the callee object's set $X_{[i]}$ differs from $X_{[i-1]}$ because of an operation call. Operation termination is captured analogously by that condition. The third condition ensures that invariants are also checked when the state configuration of an active object changes. This condition is necessary, as not every configuration change is due to operation calls. In UML Statecharts, so-called *time events* and *change events* can be specified attached to transitions, e.g., `after(1 sec)` or `when(value > 100)`. Those observers are permanently checking for the condition to become true and then raise an internal event to trigger the corresponding transition in the next RTC-step. Thus, a new Statechart configuration can also be entered without any operation call. Similarly, signals consumed in an RTC-step also cause a new Statechart configuration to be entered.

4.4 Time in UML

The current UML standard does not have an inherent notion of time. This has been investigated by different groups for the design of time-dependent systems, e.g., RT-UML [13], UML-RT based on ROOM [40], and *UML Profile for Scheduling, Performance and Time* [25].

The UML 1.5 standard leaves several issues open that inhibit a unique formal definition for the dynamic semantics of UML Statecharts. E.g., there is no particular dispatching policy defined, and it is not clear which event is selected next from the event queue to trigger the next RTC-step within the Statechart. Studying the numerous publications on formal semantics of UML Statecharts, it can be observed that none of these covers all concepts of the extensive syntax of UML Statecharts. An overview is, for instance, given in [2]. Nevertheless, it is often not necessary to regard the whole syntax of UML Statecharts in a specific modeling approach; the sublanguage and the dynamic semantics for the specific context must be clearly identified. Thus, a precise modeling approach that

makes use of UML Statecharts must still additionally define a formal dynamic semantics, either by referring to an existing one or defining a new one.

In our work, we use a *timed variant* of syntactically restricted UML Statecharts. Basically, we neglect some pseudo-state concepts (e.g., synch states) and do not allow transitions to cross borders of And-states. Furthermore, we annotate operations in Class Diagrams by operation times. This is possible with standard UML means by a stereotyped note attached to an operation (cf. the *UML Language User Guide* [3, p. 324]). Another, though non-standard, way is to simply attach a time or timing interval directly to the operation as shown in Fig. 3.

A time expression attached to an operation in such ways specifies the operation's time complexity, typically the minimal/maximal time of *expected* completion of an operation execution. Such specifications can be used in different ways, e.g., the resulting running system can be compared with the asserted times specified in the model. Alternatively, by adding up (asserted or actual) operation times, compound times of entire transactions can be computed. Dynamic semantics of our Statechart variant (that takes such operation times into account) is given by a mapping to I/O-Interval Structures as briefly introduced in Sect. 3.2. As the approach is similar to other works, e.g., [10, 24], we do not go into further details here.

When UML Statecharts are equipped with time, system state traces as given by Definition 12 must be extended to capture timing information as well. In this context, the *UML Profile for Scheduling, Performance and Time* provides a variety of timing concepts [25, Chapter 5]. In particular, timing mechanisms by means of a stereotype $\ll$ `RTclock` $\gg$ can be introduced together with appropriate tagged values, e.g., `RTresolution`. Progress of time is usually measured by counting the number of expired cycles of a strictly periodic *reference clock*. This results in a discretization of time, i.e., distinct physical instants might be associated with the same clock instant when they are temporally 'too close' to each other. Therefore, a sufficient resolution of the refer-
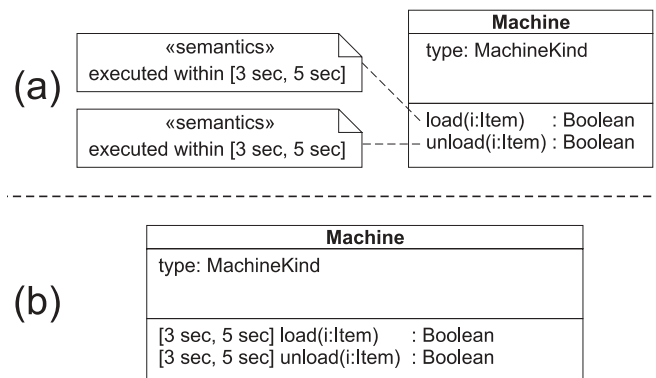

Fig. 3. Operations specified with execution times, (**a**) in standard UML notation using structured text, and (**b**) our shorthand notation

ence clock must be chosen for the particular model under investigation.

We assume in the following that a system-wide reference clock is defined together with a known resolution. The duration between two time instants is referred to as one *time unit*. This leads to an Integer-based notion of unit time delay, i.e., each time instant can be represented by an Integer value (in contrast to dense time, where time instants are represented by Real values). A trace in such a timed model is then defined as follows.

Definition 13. *(time-based trace)*
A time-based trace *for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states,*

$$trace(\mathcal{M}) \stackrel{def}{=} \langle \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \dots, \sigma(\mathcal{M})_{[i]}, \dots \rangle,$$

where each $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, represents the system state i time units after start of execution. In particular, $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state.

Note that we still require the same properties as in common traces, in particular, only one operation call per object is permitted in consecutive elements of the trace. This can be guaranteed by assuming that execution of an operation takes at least one time unit. System states of time-based traces can be compared to *clocked states* of *runs* for Interval Structures as described in Sect. 3.2 and Table 1.

5 UML Profile for real-time constraints with OCL

The integration of Statechart states into the formal model for OCL expressions allows to extend OCL towards specification of constraints that regard the *state-related dynamic behavior* of a model.

For example, consider again the manufacturing scenario with classes `Machine` and `InputBuffer`. Assume that class `InputBuffer` has an associated Statechart in which state configuration `Set{Loading}` represents that an item is currently being loaded into the buffer. In such simple cases, we allow to omit the set-notation and may simply specify `Loading` to denote a configuration.

To ensure production progress, we require that items have to periodically arrive at the input buffer within 400 time units. With other words, state `Loading` is always reached again within 400 time units. In our temporal OCL extension, an according OCL constraint is

```
context InputBuffer inv:
  self@post(1,400)->forAll( trace |
                     trace->includes(Loading))
```

Operation `post(a,b)` basically returns the set of all possible traces of state configurations starting in the current system state. Parameters `a` and `b` are timing delimiters that specify the timing interval to consider. In the example, this is the next 400 time units, i.e.,

`post(1,400)` returns a set of traces, where each trace is a sequence of 400 elements. The elements of a trace in turn are state configurations (formally, we restrict on the component Σ_{CONF} of trace $\sigma(M)$ as defined in Definition 12). In the example, state `Loading` already specifies a state configuration, such that we can apply `p->includes(Loading)` to require that trace `p` must include state configuration `Loading`.

Further examples can be found in Sect. 6. In the remainder of this section, we define an according language extension based on the OCL 2.0 proposal by the following approach.

Syntactically, we first extend the abstract OCL syntax by stereotypes for temporal expressions in Sect. 5.1. But to support modeling at the user level, a concrete syntax and operations have additionally to be defined for this extension on layer M1 of the UML 4-layer architecture. Therefore, we add some new production rules to the concrete syntax grammar of the OCL 2.0 proposal in Sect. 5.2. Note that we cannot avoid the overlap with the M1 layer in an OCL Profile, since OCL predefines types and operations on that level. As the concrete OCL syntax only partly provides the operations that are defined in OCL expressions, a standard library of predefined OCL operations is specified in [43, Chapter 6]. Correspondingly, we define operations in the context of temporal expressions in Sect. 5.3.

Semantically, our proposed state-based temporal OCL extension makes use of the notion of time-based traces as given in Definition 13). We describe the according semantic mapping in Sect. 5.4.

5.1 OCL metamodel extensions

The OCL 2.0 proposal distinguishes two subpackages for its metamodel package `Ocl-AbstractSyntax` (see Fig. 1); the *OCL type metamodel* describes the predefined OCL types and affiliated UML types, while the *OCL expression metamodel* describes the structure of OCL expressions.

States in OCL. In the OCL type metamodel, the metaclass for Statechart states is `OclModelElementType`. Generally, the metaclass `OclModelElementType` represents the types of elements that are `ModelElement`s in the UML metamodel. In that particular case, the model elements are states (or more precisely, instances of a concrete subclass of the abstract metaclass `State`), and the corresponding instance of `OclModelElementType` on layer M1 is the predefined OCL type `OclState`.

For each state, there implicitly exists a corresponding enumeration literal in `OclState`, i.e., `OclState` is seen as an enumeration type on the M1 layer, accumulating the state names of all Statechart diagrams. As there is no particular information provided how these enumeration literals are syntactically defined, we require here that the complete path – excluding the top state – is used (cf. Definition 4, item 7(c)). The state names along the

path are syntactically separated by double colons, e.g., state `N` in Fig. 2 becomes the enumeration literal `X::B::N`. In anticipation of the concrete syntax changes to be introduced, we identify final Statechart states by the new OCL keyword `FinalState`.

Configurations. The building blocks of Statecharts are hierarchically ordered states. Note that we do not regard pseudo states (like synch, stub, or history states) in this context and recall that a *composite state* is known as a state that has a set of substates and can be *concurrent*, i.e., consisting of orthogonal regions which in turn are (composite) states. *Simple states* are non-pseudo, non-composite states. To uniquely identify an active state configuration, it is sufficient to list the comprising simple states, which we denote as a *basic configuration* in accordance with Definition 10.

However, several other notions are imaginable in this context and can be easily adapted, e.g., the approach of UML 1.5 that takes the whole state tree as a configuration. For explicit specification purposes, we might also allow for *underspecified configurations* to represent sets of valid configurations. For instance, in Fig. 2, `Set{X::A::J,X::B}` could be a valid configuration specification in the sense that it denotes the set of configurations

```
{ Set{X::A::J,X::B::M},
  Set{X::A::J,X::B::N},
  Set{X::A::J,X::B::FinalState} }
```

Temporal Expressions. In the OCL expression metamodel, we introduce a new kind of operation call, i.e., stereotype `TemporalExp` represents a temporal expression that refers to traces of state configurations (cf. Fig. 4)[8]. It is the abstract superclass of stereotypes

---

[8] For our stereotype definitions, we make use of the graphical notation suggested in the official UML 1.5 specification [26,
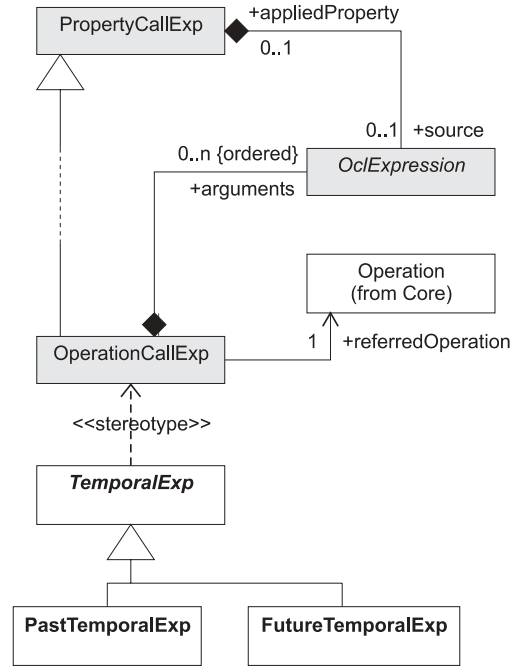

Fig. 4. Stereotypes for temporal expressions

`PastTemporalExp` for past-oriented and `FutureTemporalExp` for future-oriented temporal expressions, respectively. We need these two stereotypes in order to define a semantics for corresponding temporal operations (see Sect. 5.4).

Trace Literals. As we want to reason about traces by means of states and configurations, we also need a mechanism to explicitly specify traces with annotated timing intervals by literals. We therefore define stereotypes

---

Sects. 3.17, 3.18, 3.35, and 4.3]. In Figs. 4 and 5, metaclasses taken from the OCL 2.0 metamodel proposal are marked by gray boxes.
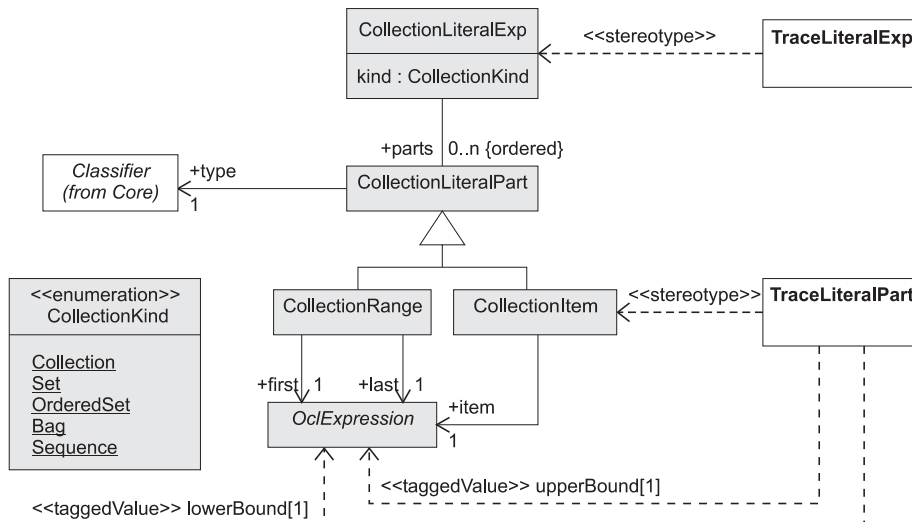

Fig. 5. Parts of the OCL expression metamodel with stereotypes for traces

`TraceLiteralExp` and `TraceLiteralPart` as illustrated in Fig. 5. The following restrictions apply here, leaving out the corresponding well-formedness rules by means of OCL for reasons of brevity.

1. The collection kind of stereotype `TraceLiteralExp` is `CollectionKind::Sequence`.
2. The type associated with a `TraceLiteralPart` must be `Set(OclState)`. Note that we do not require explicit specification of a set when a state configuration can already be specified by one state only. In this case, type `OclState` is implicitly casted to `Set(OclState)`.
3. Each `TraceLiteralPart` has a lower bound and an upper bound.
4. Lower bounds must evaluate to non-negative Integer values.
5. Upper bounds must evaluate to non-negative Integer values or to the String ′`inf`′ (for *infinity*). In the first case, the upper bound value must be greater or equal to the corresponding lower bound value.

5.2 Concrete syntax changes

Having defined new classes for temporal expressions on the abstract syntax level, modelers are not yet able to use these extensions, as they specify OCL expressions by means of a concrete syntax. In Chapter 4 of the OCL 2.0 metamodel proposal, a concrete syntax is given that is compliant with the current OCL standard. The new concrete syntax is defined by an attributed grammar with production rules in EBNF that are annotated with synthesized and inherited attributes as well as disambiguating rules. *Inherited attributes* are defined for elements on the right hand side of production rules. Their values are derived from attributes defined for the left hand side of the corresponding production rule. For instance, each production rule has an inherited attribute `env` (environment) that represents the rule's namespace. *Synthesized attributes* are used to keep results from evaluating the right hand sides of production rules. For instance, each production rule has a synthesized attribute `ast` (abstract syntax tree) that constitutes the formal mapping from concrete to abstract syntax. *Disambiguating rules* allow to uniquely determine a production rule if there are syntactically ambiguous production rules to choose from.

In the following, we present some additional production rules for the concrete syntax of the OCL 2.0 metamodel proposal. A mapping to the extended abstract OCL syntax is provided for each new production rule.

OperationCallExpCS[9]
Eight different forms of operation calls are already defined in the OCL 2.0 concrete syntax. In particular, it is

distinguished between infix and unary operations, operation calls on collections, and operation calls on objects (with or without '@pre' annotation) or whole classes. We additionally introduce rule [J] for temporal operation calls and list the synthesized and inherited attributes for syntax [J] below. Disambiguating rules for syntax [J] are defined in the specific rules for temporal expressions.

```
[A] OperationCallExpCS ::= OclExpressionCS[1]
                  simpleNameCS OclExpressionCS[2]
[B] OperationCallExpCS ::= OclExpressionCS '->'
                  simpleNameCS '(' argumentsCS? ')'
[C] OperationCallExpCS ::= OclExpressionCS '.'
                  simpleNameCS '(' argumentsCS? ')'
...
[J] OperationCallExpCS ::= TemporalExpCS

Abstract Syntax Mapping:
 -- (Re)type the abstract syntax tree variable 'ast'
 OperationCallExpCS.ast : OperationCallExp
Synthesized Attributes:
 -- Build the abstract syntax tree
 [J] OperationCallExpCS.ast = TemporalExpCS.ast
Inherited Attributes:
 -- Derive the namespace stored in variable 'env'
 [J] TemporalExpCS.env = OperationCallExpCS.env
```

TemporalExpCS
A temporal expression is either a past- or future-oriented temporal expression.

```
[A] TemporalExpCS ::= PastTemporalExpCS
[B] TemporalExpCS ::= FutureTemporalExpCS
```

We leave out the rather simple attribute definitions here. Basically, the abstract syntax mapping defines `TemporalExpCS.ast` to be of type `TemporalExp`, the synthesized attribute `ast` is built from the right hand sides, and the inherited attribute `env` is derived from `TemporalExpCS`.

FutureTemporalExpCS
A future-oriented temporal expression is a kind of operation call. We additionally have to introduce the operator '@' to indicate a subsequent temporal operation. Note that an operation call in the abstract syntax has a source, a referred operation, and operation arguments, so the abstract syntax tree `ast` must be built with corresponding synthesized attributes.

```
FutureTemporalExpCS ::= OclExpressionCS '@'
                  simpleNameCS '(' argumentsCS? ')'

Abstract Syntax Mapping:
 FutureTemporalExpCS.ast : FutureTemporalExp
Synthesized Attributes:
 FutureTemporalExpCS.ast.source= OclExpressionCS.ast
 FutureTemporalExpCS.ast.arguments= argumentsCS.ast
 FutureTemporalExpCS.ast.referredOperation=
    OclExpressionCS.ast.type.lookupOperation(
            simpleNameCS.ast,
            if argumentsCS->notEmpty()
```

---

[9] All non-terminals are postfixed by 'CS' (short for *Concrete Syntax*) to better distinguish between concrete syntax elements and their abstract syntax counterparts.

```
                then argumentsCS.ast->collect(type)
                else Sequence{}
            endif )
Inherited Attributes:
   OclExpressionCS.env = FutureTemporalExpCS.env
   argumentsCS.env     = FutureTemporalExpCS.env
Disambiguating Rules:
   -- Operation name must be a (future-oriented)
   -- temporal operator.
   [1] Set{'post'}->includes(simpleNameCS.ast)
   -- The operation signature must be valid.
   [2] not FutureTemporalExpCS.ast.
                   referredOperation.oclIsUndefined()
```

If other temporal operations than @ `post(a,b)` need to be introduced at a later point of time, only disambiguating rule [1] has to be modified correspondingly. For instance, `next()` might be introduced as a shortcut for `post(1,1)`, or `post()` without any parameters could be the shortcut for `post(1,'inf')`.

A corresponding extension to past temporal operations can be easily introduced, e.g., by means of the operation name `pre()`. In the remainder of this article, we only focus on `FutureTemporalExpCS`. Note that `pre` and `post` as operation names cannot be mixed up with pre- and postcondition labels or the @ `pre` time marker, because operations require subsequent brackets.

TraceLiteralExpCS

Trace literal expressions are a special form of collection literal expressions, as they represent sequences of explicitly specified configurations. In order to allow interval definitions for trace specifications, we have to specify some new production rules. We first introduce a new chain production rule to provide an alternative to common collection literal expressions.

```
[A] CollectionLiteralExpCS ::=
                     CollectionTypeIdentifierCS
             '{' CollectionLiteralPartsCS? '}'
[B] CollectionLiteralExpCS ::= TraceLiteralExpCS

Abstract Syntax Mapping:
   CollectionLiteralExpCS : CollectionLiteralExp
Synthesized Attributes:
   ...
   [B] CollectionLiteralExpCS.ast.parts =
                        TraceLiteralExpCS.ast.parts
   [B] CollectionLiteralExpCS.ast.kind  =
                        TraceLiteralExpCS.ast.kind
Inherited Attributes:
   ...
   [B] TraceLiteralExpCS.env =
                        CollectionLiteralExpCS.env
```

In syntax [A], `CollectionTypeIdentifierCS` distinguishes between literals for collections (`Set`, `OrderedSet`, `Sequence`, and `Bag`), and production rule `Collection-LiteralPartsCS` collects a number of expressions. Option [B] is added to provide a notation for traces. The

collection kind of traces is `CollectionKind::Sequence` by default, as specified below.

```
TraceLiteralExpCS ::= 'Trace'
                      '{' TraceLiteralPartsCS '}'

Abstract Syntax Mapping:
   TraceLiteralExpCS.ast : TraceLiteralExp
Synthesized Attributes:
   TraceLiteralExpCS.ast.parts =
                        TraceLiteralPartsCS.ast
   TraceLiteralExpCS.ast.kind =
                        CollectionKind::Sequence
Inherited Attributes:
   TraceLiteralPartsCS.env = TraceLiteralExpCS.env
```

We here introduce the new keyword `Trace` to denote trace specifications, but note that no new kind of collection type is necessary on the metalevel, as we treat traces simply as sequences.

TraceLiteralPartCS

The production rule `TraceLiteralPartsCS` assembles the individual elements of a trace specification. It is defined correspondingly to the already existing production rule for collection literal parts, such that definitions of `ast` and `env` are left out for reasons of brevity.

```
TraceLiteralPartsCS[1] ::= TraceLiteralPartCS
                ('',' TraceLiteralPartsCS[2] )?
```

For each trace literal part, a timing interval may be associated, which specifies how long a configuration is active. Intervals are of the syntactical form `[a,b]`, with `a` evaluating to a non-negative Integer, and `b` either a non-negative Integer with $b \geq a$ or the String literal `'inf'` (cf. well-formedness rules of `TraceLiteralExp` in Sect. 5.1). If only one delimiter is specified, this is taken as the upper bound, and the lower time bound is implicitly set to zero. If no interval is specified at all, the bounds are implicitly set to `[0,'inf']`. The according grammar rule is as follows.

```
TraceLiteralPartCS ::= OclExpressionCS[1]
         ( '[' (OclExpressionCS[2] ',')?
                (OclExpressionCS[3] | 'inf') ']'
         )?

Abstract Syntax Mapping:
   TraceLiteralPartCS.ast : TraceLiteralPart
Synthesized Attributes:
   TraceLiteralPartCS.ast.item =
                          OclExpressionCS[1].ast
   TraceLiteralPartCS.ast.lowerBound =
              if OclExpressionCS[2]->notEmpty()
                then OclExpressionCS[2].ast
                else '0'
              endif
   TraceLiteralPartCS.ast.upperBound =
              if OclExpressionCS[3]->notEmpty()
                then OclExpressionCS[3].ast
                else 'inf'
              endif
Inherited Attributes:
```

```
OclExpressionCS[1].env = TraceLiteralPartCS.env
OclExpressionCS[2].env = TraceLiteralPartCS.env
OclExpressionCS[3].env = TraceLiteralPartCS.env
```

CollectionTypeCS

To allow trace specifications as part of variable definitions and provide a means for explicit typing on the concrete syntax level, we need to add a rule for *explicit* referencing to a type called `Trace`. We therefore add an alternative production rule in the context of `collectionTypeCS`.

```
[A] collectionTypeCS ::= collectionTypeIdentifierCS
                                      '(' typeCS ')'
[B] collectionTypeCS ::= 'Trace'

Abstract Syntax Mapping:
   typeCS.ast : CollectionType
Synthesized Attributes:
 ...
[B] collectionTypeCS.ast.oclIsKindOf(SequenceType)
[B] collectionTypeCS.ast.elementType.
                            oclIsKindOf(SetType)
[B] collectionTypeCS.ast.elementType.elementType.
                            oclIsKindOf{OclState}
Inherited Attributes:
  -- none for [B]
```

5.3 Standard library operations

In our previous work [18], we introduced two new built-in types called `OclConfiguration` and `OclPath` on the M1 layer to handle temporal expressions. We present an alternative approach that avoids to introduce new types and instead operates on the already existing OCL collection types.

Configuration Operations. For configurations as a special form of sets of states, we have to elaborate on operations applicable to sets that return collections since the resulting collection can be an invalid configuration with an arbitrary set of states. Nevertheless, most of the existing general collection operations [43, Sect. 6.5.1] can be directly applied to configurations. These are: =, <>, size(), count(), isEmpty(), notEmpty(), includes(), includesAll(), excludes(), and excludesAll(). In addition, iterator operations exists(), forAll(), any(), one() are applicable as well [43, Sect. 6.6.1]. Other OCL set operations applied to configurations, e.g., union() and intersection(), might result in arbitrary sets of states rather than in valid configurations. We allow such operations, but explicitly mention that they have to be used with care.

Trace Operations. Similar to configurations, many of the existing OCL sequence operations can immediately be applied to traces of configurations. These operations are: =, <>, size(), isEmpty(), notEmpty(), includes(), includesAll(), excludes(), excludesAll(), subSequence(), prepend(), first(), at(), exists(), forAll(), any(), one(). Operations last() and append() can be applied to traces

of finite length only. Note that some sequence operations may result in invalid traces, e.g., select() and collect().

Additional Operations for OclAny. We introduce an operation `oclInConf()` that checks for an active configuration. Given a system state $\sigma(\mathcal{M})$, an object $\underline{oid} \in \Sigma_{CLASS,c}$, and a set of states $cfg \in I_{type}(Set(OclState))$, the semantics of operation `oclInConf()` is then defined by function

$$I_{(oclInConf:OclAny \times Set(OclState) \to Boolean)}(\underline{oid}, cfg) \stackrel{def}{=}$$

$$\begin{cases} true, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \land cfg \in B_c \\ & \land cfg = \sigma_{CONF,c}(\underline{oid}), \\ false, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \land cfg \in B_c \\ & \land cfg \neq \sigma_{CONF,c}(\underline{oid}), \\ \bot, & \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c}, \\ \bot, & \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \land cfg \notin B_c \cup \{\bot\}, \\ \bot, & \text{if } cfg = \bot \end{cases}$$

In the definition above, B_c denotes the set of basic configurations of Statechart SC_c (based on Definition 10). The definition does not consider underspecified configurations as discussed in Sect. 5.1. We here only describe the idea how to achieve the complete formal semantics. First, we additionally define the set $UnderSpecified_c$ of valid underspecified configurations for a given Statechart SC_c. Then, we provide a mapping $basicConfs_c : UnderSpecified_c \to B_c$ that gives for each underspecified configuration the according set of basic configurations. Finally, the conditions of the formal semantics are adjusted, e.g., $UnderSpecified_c$ replaces B_c and condition $cfg = \sigma_{CONF,c}(\underline{oid})$ is replaced by the condition $\forall b \in basicConfs_c(cfg) : b \in \sigma_{CONF,c}(\underline{oid})$.

We also introduce operation `post(a,b)` as a new temporal operation of `OclAny` and allow the @-operator to be used only for such temporal operations. @ `post(a,b)` returns *a set of possible future traces* in the interval [a,b]. First, all possible traces that start with the current configuration are regarded, and then the timing interval [a,b] determines the subtraces that have to be returned by the operation. The result has to be a *set* of traces, as there are typically different orders of executions possible in the future steps of a Statechart. Note that in an actual execution of a Statechart there is of course only exactly one of the possible traces selected. An informal semantics of `post(a,b)` is given as follows.

```
OclAny.post(a:Integer,b:OclAny):
                    Set(Sequence(Set(OclState)))
pre: a >= 0 and
    ( (b.oclIsTypeOf(Integer) and b >= a) or
     (b.oclIsTypeOf(String) and b = 'inf'))
```

The operation returns a set of possible future state configuration traces in the interval [a,b] including the configurations of time points a and b.

Additional operations, such as @ `post(a:Integer)` or @ `next()`, can be easily added [18]. These are operations basically derived from @ `post(a,b)`.

5.4 Semantics of temporal expressions

In this subsection, we define a formal semantics of operation `post(a,b)`. We make use of the nested collection type $TRACE \overset{def}{=} Sequence(Set(OclState))$.

Given a system state $\sigma(\mathcal{M})_{[i]}$ at time instant i, an object $\underline{oid} \in \Sigma_{CLASS,c}$, an integer value $a \in I_{type}(Integer)$, and a value $b \in I_{type}(Integer) \cup \{\infty\}$. For parameter b, we assume here that the string `'inf'` defined in the concrete syntax is directly mapped to ∞. For the symbol ∞, it holds that

$$\forall i \in \mathbb{N}_0 : i < \infty \wedge i + \infty = \infty \wedge i - \infty = \infty.$$

A trace $trace_{\underline{oid},a,b[i]} \in I_{type}(TRACE))$ for object $\underline{oid}$ that starts at time $i + a$ and ends at time $i + b - a$ is then defined by

$$trace_{\underline{oid},a,b\ [i]} \overset{def}{=} \langle cfg_0, \dots, cfg_{b-a} \rangle,$$
$$\text{where } \forall j \in \{0, \dots, b-a\} :$$
$$cfg_j \in \sigma_{CONF}(\underline{oid})_{[i+j]}$$

Each $trace_{\underline{oid},a,b\ [i]}$ is interpreted as a *possible* future execution path. It is just a possible trace, as is is not determined at time i whether $cfg_j, j \in \{1, \dots, b-a\}$, will be reached at the later point of time $i + j$.

In the case that $b = \infty$, a trace $trace_{\underline{oid},a,b\ [i]}$ is of infinite length. An explicit instantiation of such traces as part of the model is therefore not intended. However, it is possible to give according formal specifications by means of temporal logics, as illustrated below. Temporal logics specifications can then be directly used by model checkers.

Denoting the set of all possible future execution paths by $\{trace_{\underline{oid},a,b\ [i]}\}$, the semantics of operation `post(a,b)` is then defined by

$$I_{(post:OclAny \times Integer \times OclAny \rightarrow Set(TRACE))}(\underline{oid}, a, b) \overset{def}{=}$$

$$\begin{cases} \{\ trace_{\underline{oid},a,b\ [i]}\ \}, \text{if } \underline{oid} \in \Sigma_{ACTIVE,c} \\ \qquad\qquad \wedge\ a \geq 0 \wedge b \geq a, \\ \bot, \qquad \text{if } \underline{oid} \notin \Sigma_{ACTIVE,c}, \\ \bot, \qquad \text{if } a < 0 \vee a = \bot \\ \qquad\qquad \vee\ b < a \vee b = \bot\ . \end{cases}$$

In the remainder of this section, we provide a mapping from instances of `FutureTemporalExpCS` to CCTL formulae as described in Sect. 3.2.

By definition, OCL invariants for a given class must be true for all its instances at any time [43, Sect. 2.3.3]. In the context of (time-based) traces, this means that the invariant expression must be true on all possible traces at each position. Consequently, corresponding CCTL formulae have to start with the `AG` operator ('On **A**ll paths **G**lobally'), i.e., the expression following `AG` must be true on all possible future execution paths at all times. Table 2 lists OCL operations that directly match to CCTL expressions. In that table, `expr` denotes a Boolean OCL expression. `cctlExpr` is the equivalent Boolean expression in CCTL syntax. `cfg` denotes a valid configuration and `cctlCfg` is the corresponding set of states in CCTL syntax. `p` and `c` are iterator variables for traces and configurations, respectively.

Consider, for example, the last row of Table 2. When taking the particular interval `[1,100]` and a configuration from Fig. 2 for `cfg`, the resulting OCL expression is:

```
inv: obj@post(1,100)->forAll( trace |
    trace->includes(Set{X::A::L,X::B::N}))
```

We read that formula as: At any time, given the current configuration of the Statechart associated to object `obj`, all future traces `p` starting from the current configuration reach – at a certain point of time within the next 100 time units – the configuration represented by `Set{X::A::L,X::B::N}`.

Note that with the CCTL formulae of Table 2 we can only investigate models with 'persistent' active objects, i.e., corresponding objects must exist from the initial system state onwards during the complete execution time. Otherwise, we have to determine the maximal number of created objects for a model *in advance*. Only then we are able to build a corresponding set of communicating finite state machines by means of I/O-Interval Structures for each object. Dynamic object creation and deletion has to be explicitly handled by additional variables within the Interval Structures, e.g., by a Boolean variable `obj1.isAlive` for an object `obj1`. The value of that variable is then additionally checked in the CCTL formulae of the mapping. E.g., in the example above, the resulting CCTL formula is

Table 2. Mapping temporal OCL expressions to CCTL formulae

Temporal OCL Expression	CCTL Formula
inv: obj@post(a,b)→exists(p | p→forAll(c | expr))	AG EG$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→exists(p | p→exists(c | expr))	AG EF$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→exists(p | p→includes(cfg))	AG EF$_{[a,b]}$(cctlCfg)
inv: obj@post(a,b)→forAll(p | p→forAll(c | expr))	AG AG$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→forAll(p | p→exists(c | expr))	AG AF$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→forAll(p | p→includes(cfg))	AG AF$_{[a,b]}$(cctlCfg)

```
AG( obj1.isAlive →
    A(obj1.isAlive U_[1,100]
      ( !obj1.isAlive | ( obj1.isAlive &
                          obj1.S_X_A = L &
                          obj1.S_X_B = N)
    )))
```

For mapping trace literal expressions, let $e_1, e_2, \ldots, e_n$ be the trace literal parts of `TraceLiteralExpCS` with timing intervals $[a_i, b_i]$, $1 \leq i \leq n - 1$. The temporal OCL expression

```
inv: obj@post(a,b)-> includes (
        Sequence {e_1[a_1, b_1], e_2[a_2, b_2], ... , e_n})
```

maps to CCTL applying the *strong until* temporal operator (i.e., $expr_1 \ \underline{U}_{[a,b]} \ expr_2$ requires that $expr_1$ must be true between a and b time units until $expr_2$ becomes true) as follows:

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

Though we have given only some examples here, more complex formulae can be easily derived from the above.

6 Example

As a case study, we have applied our temporal OCL extension in the context of MFERT. MFERT stands for 'Modell der FERTigung' (German for: Model of Manufacturing) and provides means for specification and implementation of planning and control assignments in manufacturing processes [38]. In MFERT, nodes represent either storages for production elements or production processes. *Production Element Nodes* (PENs) are used to model logical storages of material and resources and are drawn as annotated shaded triangles. *Production Process Nodes* (PPNs) represent logical locations where material is transformed and are drawn as annotated rectangles. PENs and PPNs are composed to form a bipartite graph connected by *directed edges*, which define the flow of production elements.

MFERT graphs establish both a static and a dynamic view of a manufacturing system. On the one hand, the nodes are statically representing the participating production processes and element storages. On the other hand, edges represent the dynamic flow of production elements (i.e., material and resources) within the manufacturing system.

A sample MFERT graph is shown in Fig. 6, where transportation of items by means of automated guided vehicles (AGVs) between processing steps is illustrated. This is a small outtake of a model that is composed of different manufacturing stations and transport vehicles that transport items between stations.

In MFERT, processing is specified by means of finite state machines that are associated with PPNs. Although several variations to formally define the behavior of PPNs
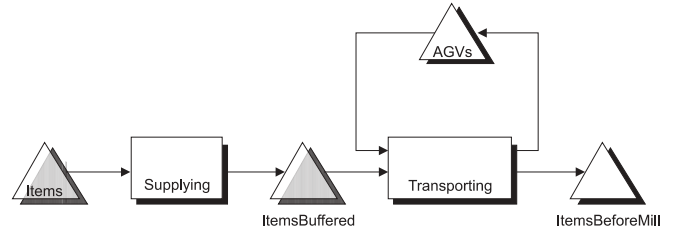

Fig. 6. Transporting Items between Machine Buffers with Automated Guided Vehicles (AGVs)

are presented in literature[10], we focus in our work on UML Statecharts. A corresponding profile for MFERT can be found in [16, 17].

The UML Statechart in Fig. 7 shows parts of the behavior specification of PPN Transporting – details of the negotiation part for accepting transportation orders are left out here, as we want to concentrate on temporal requirements for performing transportations. The transport part basically consists of a chain of activities to perform – an instance of PPN Transporting is thus controlling the activities of an AGV object. The activities are initiated by operation calls on AGVs, such as `move()`, `load()`, and `unload()`.

Recall that we allow to associate (estimated) execution times to these operations in class diagrams, as shown

---

[10] E.g., Quintanilla uses a graphical representation called interaction diagrams and formally defines them in [29].
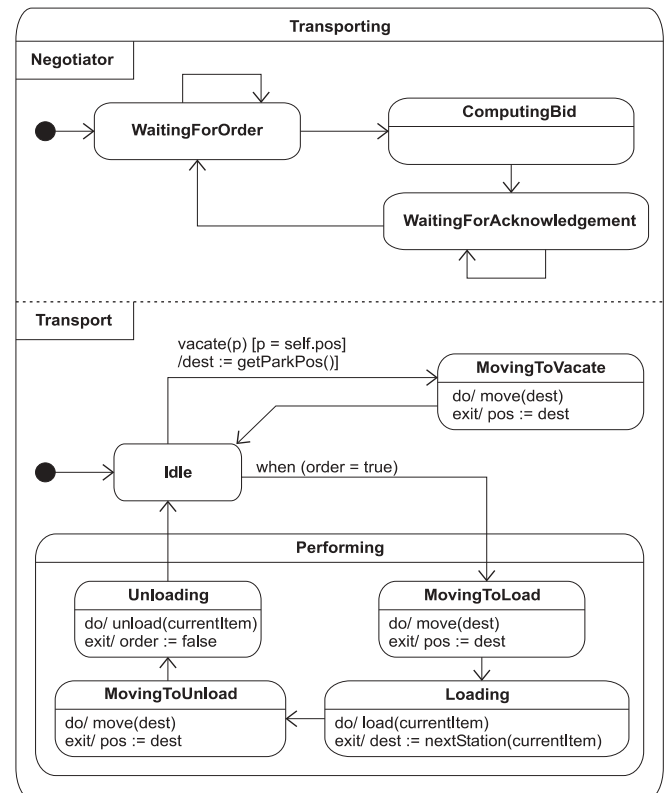

Fig. 7. Transporting Statechart

in Sect. 4.4. E.g., operations `load()` and `unload()` might take between 3 and 5 time units, while operation `move()` might take between 20 and 50 time units (depending on distances and the need of detours to avoid collisions).

We now specify some requirements, provide corresponding OCL expressions, and start with a non-temporal constraint that makes use of operation `oclInConf()`.

We want to specify that the Statechart must never be in an accepting state in the negotiation part while it is performing a transport. This can be expressed by excluding that states `WaitingForAcknowledgement` and `Performing` are both active at the same time. An according OCL constraint is

```
context Transporting inv:
 not self.oclInConf(
      Set{Negotiator::WaitingForAcknowledgement,
          Transport::Performing}
      )
```

Note that the configuration employed above is an underspecified configuration, as state `Performing` is a composite state (cf. Sect. 5.1). Such a notation is a useful shortcut, and in this case it represents a set of four basic configurations. The corresponding CCTL expression is expressed by

```
AG !(transporting_negotiator =
                       waitingForAcknowledgement
   & (transporting_transport = movingToLoad   |
      transporting_transport = loading         |
      transporting_transport = movingToUnload |
      transporting_transport = unloading       )
   )
```

Next, we require that each item that is loaded must be unloaded within 120 time units. In the context of PPN `Transporting`, the corresponding temporal OCL invariant may be

```
context Transporting inv:
 self.oclInState(Transport::Performing::Loading)
                                        implies
  self@post(1,120)->forAll(trace |
   trace->exists(conf:Set(OclState) |
    conf->includes(Transport::Performing::
                                     Unloading)
))
```

In the case that we do not consider dynamic creation and deletion of objects, a corresponding CCTL formula is

$$AG(\ transporting\_transport = loading \rightarrow$$
$$AF_{[1,120]}(transporting\_transport = unloading))$$

To ensure production progress, we require that a transporting object is not idle for too long, e.g., after at most 400 time units it has to again load an item. Note here that it is not sufficient to specify that state `Idle` will eventually be left within 400 time units, as leaving state `Idle` may also be due to a movement to vacate a position. Thus, a corresponding OCL constraint is, e.g.,

```
context Transporting inv:
 self@post(1,400)->forAll(trace |
  trace->exists(conf:Set(OclState) |
```

```
     conf->includes(Transport::Performing::
                                        Loading)))
```

The resulting CCTL formula can directly be derived from Table 2. For further readings, more examples can be found in [16, 19].

To investigate the potential for domain-independent application of our temporal OCL approach, we have mapped the general *property patterns* identified by Dwyer et al. [14] to corresponding temporal OCL expressions [20]. It turned out that only some minor extensions are necessary to cover all property patterns. First, a new operation needs to be introduced that is particularly applicable to traces, i.e., operation `startsWith(Sequence(Set (OclState)):Boolean` that checks for a matching subsequence of configurations. And second, specification means for trace literal parts have to be extended. A trace literal part becomes a logical expression with configurations as operands and unary and binary operators (such as `not`, `and`, `or`) as connectives.

7 Summary and conclusion

Though Statechart states can be already referred to in OCL syntax, their semantics in the context of OCL expressions has not been sufficiently regarded so far. To overcome this, we formalized UML Statechart configurations in the first part of this article and added them to Richters' formal object model. This builds the foundation for a semantics of OCL expressions that make use of Statechart states and predefined operation `oclInState()`. We see this work as one important issue to complete the formal semantics description of the OCL 2.0 proposal. Nevertheless, a clear definition of the OCL message concept is still missing.

We have presented a UML Profile for the specification of state-oriented real-time constraints on the basis of the latest OCL 2.0 metamodel proposal. Our approach is the first one that extends OCL using UML extension mechanisms by profiles, i.e., stereotypes, tagged values, and constraints. The approach demonstrates that an OCL extension by means of a UML Profile towards temporal real-time constraints can be seamlessly applied on M2 layer. Nevertheless, extensions have to be made on the M1 layer as well in order to enable modelers to use OCL extensions like the temporal one we have proposed here. The presented extensions are based on a future-oriented temporal logic. We currently also work on the extension to past-oriented logics.

As an example, we applied our temporal OCL extensions to MFERT [17]. A semantics is given to both, MFERT Profile and temporal OCL expressions, by a mapping to synchronous time-annotated finite state machines (I/O-Interval Structures) and temporal logics formulae (CCTL), respectively. This provides a sound basis for formal verification by Real-Time Model Checking with the RAVEN model checker [35].

We additionally have implemented an editor for MFERT [9]. Code generation for I/O-Interval Structures is currently under implementation. The temporal OCL extensions as presented here have been integrated into our OCL parser and type checker [18], which translates constraints with temporal operations to CCTL formulae.

References

1. Baar, T., Hähnle, R.: An Integrated Metamodel for OCL Types. In: France, R., Rumpe, B., Bruel, J.-M., Moreira, A., Whittle, J., Ober, I. (eds.) Refactoring the UML – In Search of the Core. Workshop at OOPSLA'2000, Minneapolis, MN, USA, October 2000

2. v. d. Beeck, M.: A Structured Operational Semantics for UML-Statecharts. Software and Systems Modeling (SoSyM) 1(2): 130–141, Springer, December 2002

3. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999

4. Bradfield, J., Kuester Filipe, J., Stevens, P.: Enriching OCL Using Observational mu-Calculus. In: Kutsche, R.-D., Weber, H. (eds.) Fundamental Approaches to Software Engineering (FASE 2002), Grenoble, France, LNCS, vol. 2306. Springer, April 2002, pp. 203–217

5. Casanova, M., Wallet, T., D'Hondt, M.: Ensuring Quality of Geographic Data with UML and OCL. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000 – The Unified Modeling Language. Advancing the Standard. 3rd International Conference, York, UK, October 2000, LNCS, vol. 1939. Springer, 2000, pp. 225–239

6. Cengarle, M., Knapp, A.: Towards OCL/RT. In: Eriksson, L.-H., Lindsay, P. (eds.) Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, LNCS, vol. 2391. Springer, July 2002, pp. 389–408

7. Clark, T., Warmer, J., editors. Object Modeling with the OCL, LNCS, vol. 2263. Springer, Heidelberg, Germany, February 2002

8. Conrad, S., Turowski, K.: Temporal OCL: Meeting Specifications Demands for Business Components. In: Siau, K., Halpin, T. (eds.) Unified Modeling Language: Systems Analysis, Design, and Development Issues. IDEA Group Publishing, 2001, pp. 151–165

9. Dangelmaier, W., Darnedde, C., Flake, S., Mueller, W., Pape, U., Zabel, H.: Graphische Spezifikation und Echtzeitverifikation von Produktionsautomatisierungssystemen. In: 4. Paderborner Frühlingstagung 2002, ALB-HNI-Verlagsschriftenreihe, Paderborn, Germany, April 2002. (in German)

10. David, A., Möller, M., Yi, W.: Formal Verification of UML Statecharts with Real-Time Extensions. In: Kutsche, R.-D., Weber, H. (eds.) 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France, LNCS, vol. 2306. Springer, 2002, pp. 218–232

11. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: Gogolla, M., Kobryn, C. (eds.) UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, LNCS, vol. 2185. Springer, 2001, pp. 104–117

12. Distefano, D., Katoen, J.-P., Rensink, A.: On a Temporal Logic for Object-Based Systems. In: Smith, S.F., Talcott, C.L. (eds.) Formal Methods for Open Object-Based Distributed Systems IV (FMOODS'2000), Stanford, CA, USA. Kluwer Academic Publishers, September 2000, pp. 285–304

13. Douglass, B.P.: Doing Hard Time: Developing Real Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley, 2000

14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering (ICSE99), May 1999, Los Angeles, CA, USA. ACM Press, May 1999, pp. 411–420

15. Ebert, J., Engels, G.: Observable or Invocable Behaviour: You have to Choose. Technical report, Universität Koblenz, Koblenz, Germany, 1994

16. Flake, S., Mueller, W.: A UML Profile for MFERT. Technical report, C-LAB, Paderborn, Germany, March 2002. http://www.c-lab.de/vis/flake/publications/index.html

17. Flake, S., Mueller, W.: A UML Profile for Real-Time Constraints with the OCL. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002 – The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, LNCS, vol. 2460. Springer, 2002, pp. 179–195

18. Flake, S., Mueller, W.: An OCL Extension for Real-Time Constraints. In: Clark, T., Warmer, J. [7], pp. 150–171

19. Flake, S., Mueller, W.: Specification of Real-Time Properties for UML Models. In: Sprague, R.H., Jr. (ed.) Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS-35), Hawaii, USA, IEEE Computer Society, January 2002

20. Flake, S., Mueller, W.: Expressing Property Specification Patterns with OCL. In: The 2003 International Conference on Software Engineering Research and Practice (SERP'03), Las Vegas, Nevada, USA, June 2003. CSREA Press, 2003

21. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8(3): 231–274, June 1987

22. Kleppe, A., Warmer, J.: Extending OCL to include Actions. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000 – The Unified Modeling Language. Advancing the Standard. 3rd International Conference, York, UK, October 2000, LNCS, vol. 1939. Springer, 2000, pp. 440–450

23. Knapman, J.: Statistical Constraints and Verification. In: Clark, T., Warmer, J. [7], pp. 172–188

24. Knapp, A., Merz, S., Rauh, C.: Model Checking Timed UML State Machines and Collaborations. In: Damm, W., Olderog, E.-R. (eds.) 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002, LNCS, vol. 2469. Springer, 2002, pp. 395–416

25. Object Management Group (OMG). UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02, September 2002. http://cgi.omg.org/-docs/ptc/02-03-02.pdf

26. Object Management Group (OMG). Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003. http://www.omg.org/technology/documents/-formal/uml.htm

27. Object Management Group Technology Committee, Analysis and Design Platform Task Force. UML 2.0 OCL RFP, May 2003. http://www.omg.org/techprocess/meetings/-schedule/UML_2.0_OCL_RFP.html (visited June 7th, 2003)

28. Petersohn, C., Urbina, L.: A Timed Semantics for the STATE-MATE Implementation of Statecharts. In: Fitzgerald, J., Jones, C., Lucas, P. (eds.) Proceedings of 4th Int. Symposium of Formal Methods Europe (FME'97): Industrial Applications and Strengthened Foundations of Formal Methods, September 1997, Graz, Austria, LNCS, vol. 1313. Springer, 1997, pp. 553–572

29. Quintanilla de Simsek, J.: Ein Verifikationsansatz für eine netzbasierte Modellierungsmethode für Fertigungssysteme. PhD thesis, Heinz Nixdorf Institute, HNI-Verlagsschriftenreihe, Band 87, Paderborn, Germany, 2001. (in German)

30. Ramakrishnan, S., McGregor, J.: Extending OCL to Support Temporal Operators. In: 21st International Conference on Software Engineering (ICSE99), Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, USA, May 1999

31. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Bremen, Germany, 2001

32. Richters, M., Gogolla, M.: A Metamodel for OCL. In: France, R., Rumpe, B. (eds.) UML 1999 – The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, LNCS, vol. 1723. Springer, 1999, pp. 156–171

33. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In: Clark, T., Warmer, J. [7], pp. 42–68

34. Roubtsova, E.E., van Katwijk, J., Toetenel, W.J., de Rooij, R.C.M.: Real-Time Systems: Specification of Properties in UML. In: Proceedings of the 7th Annual Conference of the Advanced School for Computing and Imaging (ASCI 2001). Het Heijderbos, Heijen, The Netherlands, May 2001, pp. 188–195

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

36. Ruf, J., Kropf, T.: Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In: Cerny, E., Probst, D. (eds.) Conference on Correct Hardware Design and Verification Methods (CHARME'97), Montreal, Canada. IFIP WG 10.5, Chapman and Hall, October 1997, pp. 146–166

37. Ruf, J., Kropf, T.: Modeling and Checking Networks of Communicating Real-Time Systems. In: Conference on Correct Hardware Design and Verification Methods (CHARME'99), Bad Herrenalb, Germany. IFIP WG 10.5, Springer, September 1999, pp. 265–279

38. Schneider, U.: Ein formales Modell und eine Klassifikation für die Fertigungssteuerung – Ein Beitrag zur Systematisierung der Fertigungssteuerung. PhD thesis, Heinz Nixdorf Institute, HNI-Verlagsschriftenreihe, Band 16, Paderborn, Germany, 1996. (in German)

39. Schrefl, M., Stumptner, M.: Behavior Consistent Specialization of Object Life Cycles. ACM Transactions of Software Engineering and Methodology (ACM TOSEM), ACM Press 11(1): 92–148, January 2002

40. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. White Paper, 1998.
http://www.rational.com/media/whitepapers/umlrt.pdf

41. Sendall, S., Strohmeier, A.: Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML. In: Gogolla, M., Kobryn, C. (eds.) UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, LNCS, vol. 2185. Springer, 2001, pp. 391–405

42. Stumptner, M., Schrefl, M.: Behavior Consistent Inheritance in UML. In: Laender, A., Liddle, S., Storey, V. (eds.) Proceedings of the 19th International Conference on Conceptual Modeling (ER 2000), Salt Lake City, UT, USA, October 2000, LNCS, vol. 1920. Springer, 2000, pp. 527–542

43. Warmer, J., Ivner, A., Johnston, S., Knox, D., Rivett, P.: Response to the UML2.0 OCL RfP, Version 1.6 (Submitters: Boldsoft, Rational, IONA, Adaptive Ltd., et al.). OMG Document ad/03-01-07, January 2003

44. Ziemann, P., Gogolla, M.: An Extension of OCL with Temporal Logic. In: Jürjens, J., Cengarle, M.V., Fernandez, E.B., Rumpe, B., Sandner, R. (eds.) Critical Systems Development with UML. Technische Universität München, Institut für Informatik, 2002, pp. 53–62

45. Zschaler, S.: Evaluation der Praxistauglichkeit von OCL-Spezifikationen. Master's thesis, Technical University of Dresden, Faculty of Computer Science, Dresden, Germany, August 2002. (in German)

Stephan Flake is currently a research assistant at the Co-operative Computing & Communication Laboratory (C-LAB), a joint R&D institute operated by Paderborn University and Siemens Business Services in Paderborn, Germany.

Stephan received the degree Diplom-Informatiker (M.Sc. in Computer Science) from Paderborn University, Germany, in 1999.

His research interests include UML, especially the semantics of Statecharts and OCL, formal verification by model checking, abstraction means for temporal logics, and multi-agent systems.

Wolfgang Mueller received his Diploma in Computer Science from Paderborn University, Germany, in 1989 and his doctoral degree in 1996. He is employed at C-LAB since 1989. There, he is heading the group Advanced Design Technologies (ADT).

Dr. Mueller was and is member of several programme and executive committees of various conferences like DATE and FDL. Since 1989 he authored more than 100 national and international publications in the areas of user interfaces, system design methodologies, system description languages, and system integration technologies.