Towards OCL/RT

María Victoria Cengarle[1]* and Alexander Knapp[2]

[1] Technische Universität München
cengarle@in.tum.de
[2] Ludwig–Maximilians–Universität München
knapp@informatik.uni-muenchen.de

Abstract. An extension of the "Object Constraint Language" (OCL) for modeling real-time and reactive systems in the "Unified Modeling Language" (UML) is proposed, called OCL/RT. A general notion of events that may carry time stamps is introduced providing means to describe the detailed dynamic and timing behaviour of UML software models. OCL is enriched by satisfaction operators $@\eta$ for referring to the value in the history of an expression at the instant when event η occurred, as well as the modalities `always` and `sometime`. The approach is illustrated by several examples. Finally, an operational semantics of OCL/RT is given.

Keywords. Real-time systems, OCL, UML, events

1 Introduction

The "Object Constraint Language" (OCL [25]) provides means to constrain realisations of software models in the "Unified Modeling Language" (UML [3]) by textual specifications in a formal, navigational expression language. OCL specifications complement UML models where constraints for defining meaningful realisations can not or not conveniently be stated diagrammatically. The OCL focusses on the axiomatic specification of consistent system states by invariants and the transformations of system states by means of pre- and post-conditions for operations.

As it stands, OCL thus seems to be well-suited for describing constraints on UML models for conventional business applications [8,2], but shows distinct limitations for specifying reactive, embedded, or real-time systems as the language does not feature time or signal handling constructs, nor is capable of expressing general liveness properties of systems conveniently. Moreover, performance aspects, which play an important role in today's software systems, cannot be easily expressed in the OCL. On the other hand, employment of the UML for describing systems where time, performance, or reactive behaviour is in focus has gained considerable interest [6,7,10] building on the general impact of object-oriented technology in real-time software engineering [21]. In fact, the UML shows some

* This research has partially been carried out while at Fraunhofer Institut Experimentelles Software Engineering.

support for these kinds of systems by including a signal and an event concept, timed state machines, and collaborations with timing annotations. Moreover, specialised real-time language extensions and profiles have been devised [22,17]. However, most of these UML notions have only been provided with an intuitive semantics and have no formal counterpart. Methodologically, UML reactive and timing specifications, like state machines, tend to be rather concrete; the interspersing of modelling and constraint diagrams may make it hard to grasp the proof obligations.

What therefore may be called for is an enhancement of the OCL by constructs for time and signals in order to also complement UML real-time models by formal and abstract specifications. We propose such an extension to the OCL, called "OCL for real-time" (OCL/RT). In OCL/RT, time evolution as well as signal occurrences are captured by a generalised notion of UML events that carry a time stamp. In accordance with the design principles of the OCL, events are viewed on locally and are associated to instances. Based on this event concept, special satisfaction operators $\texttt{@}\eta$ enable referring to the system state at the occurrence of an event η and thus provide control over a history of system states. Furthermore, the modalities `always` and `sometime` provide means to specify safety as well as liveness properties. This proposal takes up some of the ideas present in Lano's "Real-time Action Logic" (RAL) for formal object-oriented software development [13] and the work by Trentelman and Huisman on extending the "Java Modeling Language" (JML) by temporal logic [24].

Related work. Several approaches to coping with time and events in OCL and related specification languages for the UML have already been reported in the literature: Conrad and Turowski [5] extend OCL by temporal modalities but do not consider real-time systems proper. Kleppe and Warmer [12], in the same vein as Álvarez et. al. [1] and the "Action Semantics for the UML" integrated with the UML 1.4 specification [19], define a dynamic semantics of UML and its actions using OCL. Though they capture history by local snapshots, they neither provide a notion of time nor a notion of event. These concepts are investigated in detail in the response to the request for proposals "Schedulability, Performance, and Time for the UML" [17], but an extension to the OCL is not discussed. Lavazza, Quaroni, and Venturelli [14] propose the use of "TRIO" real-time specifications to capture the semantics of UML state machines with time annotations; this approach, along the techniques introduced by Lano [13], indeed provides a powerful specification language, but lacks tight integration with conventional notations for UML.

Outline. In Sect. 2 we briefly review the OCL syntax, intended semantics, and expressiveness. The OCL/RT notion of event as well as its relationship to time is motivated in Sect. 3. In Sect. 4 the concepts and syntax of OCL/RT are introduced. We illustrate our proposal in Sect. 5 by means of several typical examples. Sect. 6 defines the formal semantics of OCL/RT. Finally, in Sect. 7, we conclude by drawing advantages and disadvantages of our proposal and hint at possible directions of future work.

2 OCL

We briefly summarise the syntax and semantics of the OCL by means of an example. An introduction to OCL is provided by Warmer and Kleppe [25], the syntax and semantics of OCL 2.0 is discussed in more detail in [16]. The overview of the OCL semantics given here is based on the operational semantics for OCL expressions by the authors [4].

The UML class diagram in Fig. 1 represents the *static structure* of a (over-simplified) model of several automatic teller machines (ATMs) connected to a single bank showing an *association* with according *multiplicities* between the *classes* ATM and Bank. An ATM has a depot *attribute*, holding the current amount of money it can spend; the identification number of the card currently put in, with cardId set to, say, zero if it holds no card; and a state indicating whether an error has occurred during processing. An ATM may spend an amount of money when *operation* spend is called on it. The bank offers two operations: credit withdraws an amount of money from the card holder's account if this amount is covered; requestRefill registers ATMs whose depots are running low.

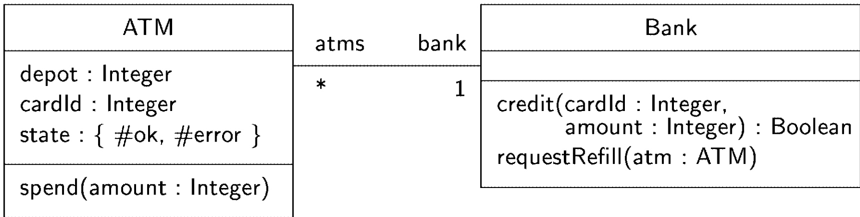


Fig. 1. UML class diagram for ATMs

2.1 Invariants, Pre-/Post-conditions, and Definitions

In OCL, a class *invariant* specifies a condition that has to be satisfied throughout the whole life-time of instances of the class. An OCL invariant for ATMs may require that, whenever the state of an ATM does not indicate an error, there is enough money to spend:

```
context ATM
inv: (self.state = #ok) implies (self.depot >= 100)
```

OCL uses the dot-notation for navigation to attributes and via associations (as well as for operation calls). The OCL expression `self` denotes the instance the constraint is evaluated on and may be omitted if the navigation reference remains unambiguous. Each OCL type, like `Enumeration` (for #ok), `Integer` or the types of the underlying UML static structure, shows a special undefined

value `undef`; an expression can be tested whether it results in `undef` using the predefined function `isUndef()`.

An *axiomatic specification* for an operation defines the behaviour of the operation by a pre-/post-condition pair. An OCL axiomatic specification for operation spend on an ATM may require that whenever spend is called, the ATM must not be in an error state, it must hold some card, the amount of money to be withdrawn is positive, and the depot covers the withdrawal. After spend has been executed, the right amount of money must have been spent or some error has occurred:

```
context ATM::spend(amount : Integer)
pre: (state = #ok) and (cardId <> 0) and
     (amount > 0) and (depot > amount+100)
post: (depot = depot@pre-amount) or (state = #error)
```

The post-condition expression makes use of the OCL operator `@pre` that yields an expression's value at pre-condition time.

OCL also provides a mechanism to introduce *auxiliary attributes* and (arbitrarily recursive) *operations* not specified in the underlying UML model. A bank may define an operation calculating the sum of the depots in its ATMs:

```
context Bank
def: depotsSum() : Integer =
       self.atms->iterate(i : ATM;
                          sum : Integer = 0 | sum+i.depot)
```

The expression `self.atms` evaluates to a set of instances of class ATM, reflecting the multiplicity of atms. The OCL predefined operation `iterate` iterates through a given collection and accumulates the result of evaluating an expression with an iterator variable bound to the current element and an accumulator variable bound to the previous result. Like for all collection operations, e.g., `select`, `reject`, or `collect`, a special arrow notation is used.

2.2 Actions

Kleppe and Warmer [11] have proposed an extension of the original OCL by action clauses for classes and operations; see also [16].

An *action clause for classes* requires that whenever a condition becomes satisfied, an operation has to be called. For example, if an ATM is about to run out of money, it has to request a refill from its bank:

```
context ATM
action: depot < 1000 ==> bank.requestRefill(self)
```

An *action clause for operations* specifies that, when some condition is satisfied at post-condition time, certain other operation calls must have happened while executing the operation. For example, during execution of spend operation credit must have been called on the bank with the current card identification and the amount of money to be withdrawn:

```
context ATM::spend(amount : Integer)
action: true ==> bank.credit(cardId, amount)
```

An action clause for operations implicitly assumes the pre-condition of the operation.

2.3 Semantics

Formally, the semantics of evaluating an OCL *expression* in a system state may be captured as follows: System states are formalised by *dynamic bases*. A dynamic basis comprises an implementation of the predefined OCL types and their operations as well as the set of current instances of classes together with their attribute valuations, connections to other instances, and implementations of operations. Moreover, a dynamic basis can be extended by implementations of auxiliary, user defined operations. Given an OCL expression e to be evaluated over a dynamic basis ω and a variable environment γ assigning values to variables (including `self`), we write $\omega; \gamma \vdash e \downarrow v$ for the judgement that e evaluates in this situation to the value v. Structural operational rules [4] define the procedure of evaluating an OCL expression.

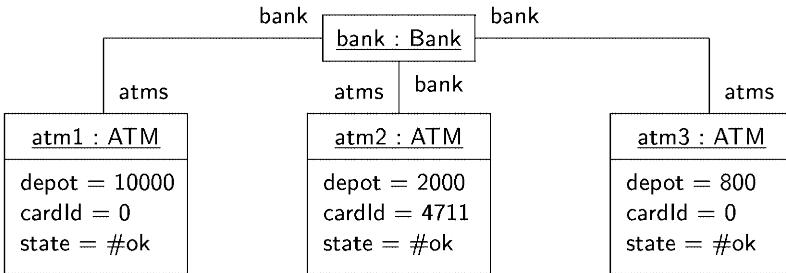


Fig. 2. UML object diagram for a sample ATM configuration

For example, the evaluation of a call to the auxiliary operation `depotsSum` on the bank bank over a dynamic basis ω corresponding to the system state described by the *object diagram* in Fig. 2 is given by the judgement:

$$\omega; \texttt{self} \mapsto \mathsf{bank} \vdash \texttt{self.depotsSum()} \downarrow 12800$$

Moreover, the ATM invariant above is satisfied for all instances of ATM, i.e., for $1 \leq i \leq 3$:

$$\omega; \texttt{self} \mapsto \mathsf{atm}i \vdash \texttt{self.state = \#ok implies self.depot >= 100} \downarrow \texttt{true}$$

OCL *constraints*, i.e., invariants, pre-/post-conditions, and action clauses, restrict the *runs* of systems modelled in UML. These constraints specify safety

properties of a system, such that if a constraint is satisfied by a system run, all finite initial segments of the system run satisfy the constraint. Roughly speaking, an OCL constraint has to hold for a (potentially infinite) sequence of dynamic bases $\vec{\omega} = \omega_0, \omega_1, \ldots$ where ω_0 represents the initial system state and ω_n is transformed into ω_{n+1} by a step of the system. However, the OCL semantics does not prescribe at which states of such a run an invariant has to hold indeed — an instance's invariant may be violated if an operation is currently executed on this instance (see, e.g., [15]). Taking some ω_m to be the system state where an operation is called and ω_n with $m \leq n$ the system state where this operation call terminates, a pre-/post-condition pair for this operation holds (cf. [20]) whenever a `true` pre-condition over ω_m implies a `true` post-condition over ω_n with expressions of the form e`@pre` evaluated over ω_m. But it is unclear how system states are to be identified where an operation is called or where an operation terminates. The interpretation of action clauses shows similar problems.

3 Time and Events

Though the OCL provides sufficient means to describe the functional behaviour of software systems modelled in the UML, its expressiveness is rather limited when it comes to either timing and performance issues or reaction to (external) signals. To some extent, this deficiency can be countered by employing UML diagrams as specifications; however, the lack of formal underpinning of the UML remains a main impediment.

3.1 Time and Signals in OCL and UML

In software systems, time plays a particular role when it is desirable or even absolutely necessary that a service be finished within certain time bounds. For example, the period to be waited for when asking for money at an automatic teller machine, i.e. calling spend on an ATM in the example of the previous section, must not exceed a certain predefined time limit. An OCL solution to such a requirement would be to define an explicit clock attribute requiring the clock to be reset to zero in the pre-condition of spend and putting a constraint on the clock's value in the post-condition:

```
context ATM::spend(amount : Integer)
pre: (clock = 0) and (state = #ok) and ...
post: (clock <= T) and ...
```

Such a solution may become unwieldy when several dependent timing constraints are required. Alternatively, a UML sequence diagram with timing annotations as suggested in the UML specification [18, Sect. 3.64] may be employed, see Fig. 3. The precise meaning, however, is undefined.

Time also plays a role when, the other way round, the occurrence of some (external) signal is waited for during a certain period, and if nothing happens the system has to react in a predefined way. For instance, when inserting the

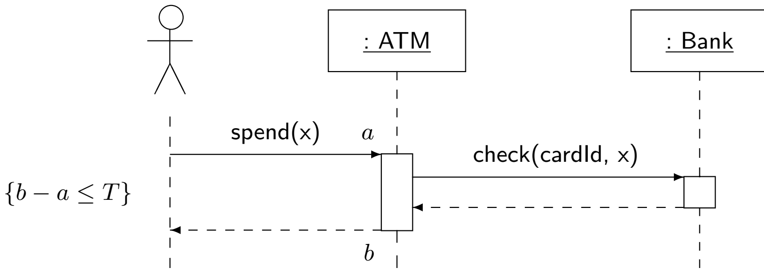
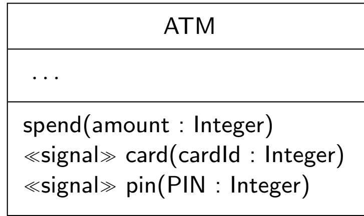
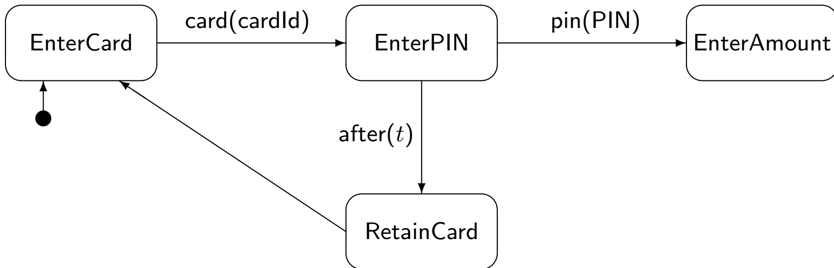


Fig. 3. Sequence diagram with timing constraints for ATM::spend



(a) ATM with signals



(b) Timed state machine for ATM with signals (fragment)

Fig. 4. UML specifications for ATMs with signals

bank card in an automatic teller machine, if the card holder does not enter the corresponding personal identification number (PIN) within a reasonable time, the card may be retained by the automatic teller machine, which is now ready to accept another card. In order to accommodate for this requirement, in Fig. 4(a) two signals card and pin for class ATM are introduced by defining appropriate UML receptions (which are meant to be asynchronous in contrast to synchronous operations). But OCL does not offer any convenient means to handle the occurrence of signals and thus to specify the intended behaviour. In UML, the deadline for entering the PIN may be expressed by using a state machine with time trig-

gers for class ATM as suggested in the UML specification [18, Sect. 3.74], see Fig. 4(b). However, the precise meaning is unclear and the specification may be too concrete in introducing state machines for classes.

3.2 Event Concept

The concepts of starting and finishing time of a service can be used to measure the duration of fulfilling the service by a system. In the same vein, when waiting for a signal for a limited amount of time, the occurrence time of a signal can be used to decide whether the signal arrived in time. It seems natural to reify these concepts in a system model as *events* that may be accompanied by a *time stamp*.

In fact, UML introduces several kinds of events as sub-classes of the meta-class Event that, however, cannot be marked with the time of their occurrence: meta-class CallEvent for operation calls, meta-class SignalEvent for signal occurrences, and meta-class ChangeEvent for changing of conditions. The UML meta-class TimeEvent for the running down of a timer that is started when a state of a state machine is entered cannot be linked to other events. Furthermore, also other kinds of events may be of interest in real-time or reactive systems: For example, an assignment to an attribute may be an event. In an object-oriented environment, an operation invocation may be subdivided into several events, e.g., an event originated by the caller (i.e., to send a message), an event originated by the run-time support (i.e., to place the message in the input queue of the callee), and an event originated by the callee (i.e., to choose the message for its process). In general, it is only up to the system specifier which are the events of discourse.

4 OCL/RT

In order to open up the event and thus the time perspective for OCL, language primitives for handling events, their occurrence time, and their sequence have to be introduced. The detailed structure of the available events may depend on the system's nature.

4.1 Events

OCL/RT is based on a modification and extension of the original UML abstract meta-class Event as depicted in Fig. 5. Each event (instance) shows the time at which it occurred by a link to the new primitive data type Time that represents the global system time. We assume that Time comes with a total ordering relation $\leq$ for comparing time values, an associative and commutative binary operation $+$ for adding time values, and a class attribute now that always yields the current system time. Events are associated to instances (of classifiers), such that an instance is linked to all its current events. Similar to UML, an event may carry a list of actual parameters.

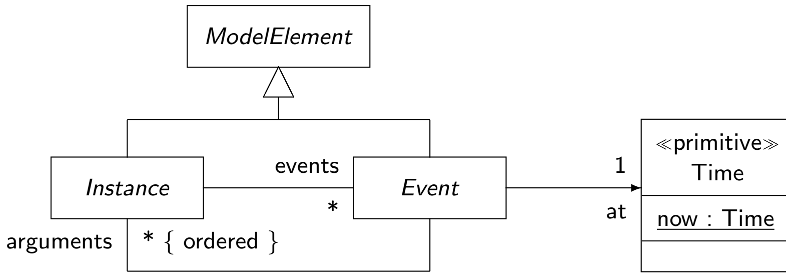


Fig. 5. Event meta-class

We extend OCL by the new types `Event` and `Time` that correspond to Event and Time, respectively and that reify their structure in OCL/RT. Each system state complying to an UML model based on the OCL/RT extension thus has to show a current time, which can be accessed by the OCL/RT expression `Time.now`. For each of the instances in such a system state a set of current events for this instance has to be present, which can be accessed by the OCL/RT expression e`.events` if e evaluates to an instance. The set of current events is accumulative, i.e., over a system run all events that have been raised for the instance value of e are present in e`.events`.

4.2 Constraints

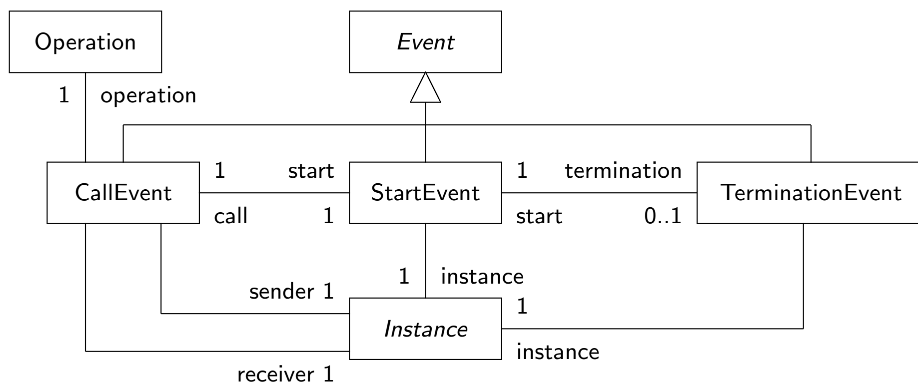
For the definition of OCL/RT constraints, which are evaluated over a sequence of systems states, we introduce a new clause

```
context C
constr: c
```

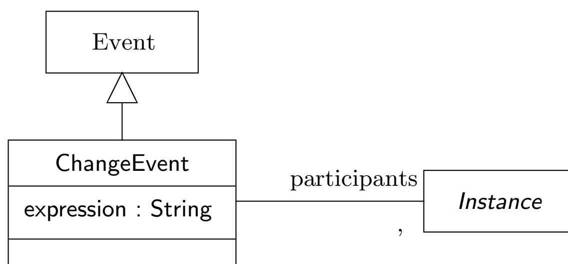
where C is a classifier and c an OCL/RT constraint expression. OCL/RT *constraint expressions* comprise all boolean OCL/RT expressions, but may also show the *modality* `always` such that `always` c for a OCL/RT constraint expression is satisfied over a system run when c evaluates to `true` in all states of the run. As is customary in modal logic, we define a modality `sometime` by abbreviating `not (always (not` c`))` to `sometime` c. Finally, OCL/RT expressions may include *satisfaction operators* `@`η that when applied to an expression correspond to evaluating the expression at the system state where event η occurred.

4.3 Invariants, Pre-/Post-conditions, and Action Clauses

The OCL/RT constraint language is expressive enough to subsume (interpretations of) the original OCL invariant, pre-/post-condition, and action clause constraints. Accepting those interpretations, `inv:`, `pre:`, `post:`, and `action:` can be used as convenient abbreviations in OCL/RT.



(a) Events for operations



(b) Events for changes

Fig. 6. Event model for OCL

We define a suitably expressive hierarchy of events, i.e. an *event model* for OCL in Fig. 6. For operations, see Fig. 6(a), we assume that a CallEvent (instance) occurs when a sender instance issues a call on operation operation to a receiver instance; it is visible to its sender and receiver. A StartEvent is raised, whenever instance is about to start executing the operation; it is visible to instance. Finally, a TerminationEvent occurs when the execution of an operation is finished; it is again visible to instance. CallEvent and StartEvent show as arguments the actual parameters of the operation call. All three event types are linked: A StartEvent can refer to its causing CallEvent, a TerminationEvent to its causing StartEvent. Moreover, if an operation call terminates, the StartEvent corresponding to starting execution of the operation has to show a link termination to a TerminationEvent. For changes, see Fig. 6(b), we assume that arbitrary expressions can be tested over a system state, and that whenever the expression value changes from `false` to `true`, a ChangeEvent occurs; a ChangeEvent is visible to all the instances of the system state.

Invariants. An OCL specification of an invariant

```
context C
inv: inv
```

may be interpreted as: *inv* must be satisfied for an instance of C, whenever an operation is called on the instance from outside; cf. [15]. In OCL/RT this interpretation, taking calls from the outside to be calls of public operations, reads as follows:

```
context C
def: publicCalls() : Event
       events->select(e |
         e.isTypeOf(StartEvent) and
         e.call.operation.visibility = #public)
constr: always (publicCalls()->forAll(s | inv@s))
```

In particular, an invariant may be violated during execution of an operation.

Pre-/post-conditions. OCL pre-/post-condition specifications

```
context C::o(x1 : τ1, ..., xn : τn)
pre: pre
post: post
```

can be expressed by

```
context C
def: startso() : Event
       events->select(e |
         e.isTypeOf(StartEvent) and
         e.call.operation.name = "o")
constr: always (startso()->forAll(s |
                  (pre~@s and (not isUndef(s.termination)))
                  implies
                  post~@(s.termination)))
```

The OCL/RT translations $\widetilde{pre}$ and $\widetilde{post}$ of the OCL expressions *pre* and *post* replace each reference to a parameter x_i by `s.arguments->at(`i`)`, i.e. the ith argument of event `s` representing the start of an operation execution. Moreover, in order to obtain $\widetilde{post}$, each occurrence of `@pre` is replaced by `@s`.

Action clauses for classes. An OCL action clause for a class

```
context C
action: cond ==> e.m(e1, ..., ek)
```

conveys that whenever condition *cond* becomes satisfied, an operation call on m has to be sent to the instance value of e. In OCL/RT this can be specified as follows, assuming that a short amount of time ε may pass between *cond* becoming satisfied and calling m:

```
context C
def: changes() : Event
       events->select(e | e.isTypeOf(ChangeEvent) and
                          e.expression = "cond")
def: callsm() : Event
       events->select(e |
         e.isTypeOf(CallEvent) and e.operation.name = "m")
constr: always (changes()->forAll(c | sometime
                  (callsm()->exists(m |
                     m.sender = self and m.receiver = e@c and
                     m.arguments->at(1) = e1@c and ... and
                     m.arguments->at(k) = ek@c and
                     c.at <= m.at and m.at <= c.at + ε))))
```

Action clauses for operations. An OCL action clause for an operation

```
context C::o(x1 : τ1, ..., xn : τn)
action: cond ==> e.m(e1, ..., ek)
```

means that if at termination time of an operation execution for o the condition *cond* holds, then an operation call for operation m on the instance value of e has been raised between the events of starting the operation o and its termination. This can be expressed by the following OCL/RT specification:

```
context C::o(x1 : τ1, ..., xn : τn)
def: terminateso() : Event
       events->select(e | e.isTypeOf(TerminationEvent) and
                          e.operation.name = "o")
def: callsm() : Event
       events->select(e |
         e.isTypeOf(CallEvent) and e.operation.name = "m")
constr: always (terminateso()->forAll(t | (~cond@t implies
                  callsm()->exists(m | sometime
                    (m.sender = self and m.receiver = ẽ@t and
                     m.arguments->at(1) = ẽ1@t and ... and
                     m.arguments->at(k) = ẽk@t and
                     t.start.at <= m.at and m.at <= t.at)))))
```

where $\tilde{e}, \tilde{e}_1, \ldots, \tilde{e}_k$ are defined as above.

5 Examples

We illustrate the use of OCL/RT by modelling common real-time paradigms, in particular deadlines and timeouts, for UML systems, as discussed in Sect. 3. A *deadline* requires that something must occur before a specified point in time is reached. Similarly, a *timeout* expects that something can occur before a specified point in time is reached and, if this is not the case, then something will occur.

5.1 Deadlines for Operations

We define an OCL/RT constraint for the desired deadline for the operation spend of class ATM (see Fig. 1) in the ATM example of Sect. 3.1: Executions of the operation spend have to be finished within a certain time T. The post-condition of spend (see Sect. 2.1) is rewritten as follows:

```
context ATM::spend(amount:Integer)
pre: ...
post: (depot = depot@pre - amount and
       Time.now <= Time.now@pre+T) or (state = #error)
```

This specification makes use of the abbreviation introduced in Sect. 4.3.

5.2 Deadlines for Reactions to Signals

Deadlines for signals pose a similar problem as the previous example. The essential difference resides in the fact that post-conditions not necessarily are available for signals. We define an OCL/RT event model for signals in Fig. 7. A SignalEvent is raised on instance if signal is received by this instance; it is visible to instance.

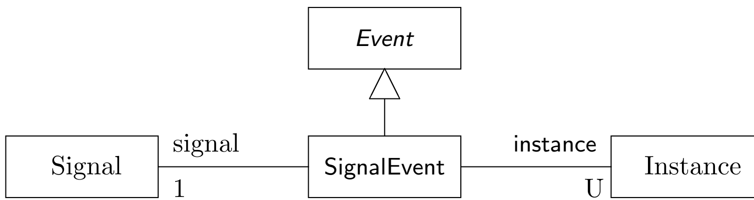


Fig. 7. Event model for signals

Assume a railway level crossing with an automatically controlled gate. Whenever a sensor signals the approach of a train, the gate has to be completely closed within a certain time limit. After the train has crossed, the gate starts opening. It may happen that the gate is opening when the next train is detected by the sensor; in this case, the gate has to stop opening and close again. This example is taken from [14]. A UML model for gate controllers is given in Fig. 8.

We specify constraints on the reactions to the open and close signals. In order to enhance readability of these constraints, we introduce several auxiliary attributes in the context of Gate that collect the signal events for close, the change events that are raised when angle becomes 0, and the change events that are raised when angle becomes Real.pi, respectively:

```
context Gate
def: closeSignals : Set(SignalEvent) =
       events->select(e |
         e.isTypeOf(SignalEvent) and e.signal.name = "close")
```

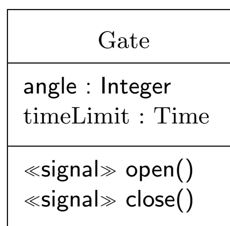


Fig. 8. UML class diagram for gate controllers

```
def: gateDownEvents : Set(ChangeEvent) =
       events->select(e |
         e.isTypeOf(ChangeEvent) and e.expression = "angle=0")
def: gateUpEvents : Set(ChangeEvent) =
       events->select(e | e.isTypeOf(ChangeEvent) and
                          e.expression = "angle=Real.pi")
```

The following constraint checks that, for any close signal, within timeLimit after the signal has arrived the gate is indeed closed. Furthermore, we require that between these two events, namely the arrival of a signal close and the gate reaching a horizontal position, the gate does not become open.

```
context Gate
constr: always (closeSignals->forAll(cs |
          sometime (gateDownEvents->exists(gd |
            cs.at < gd.at and gd.at <= cs.at+timeLimit
            and not gateUpEvents->exists(gu |
                  cs.at < gu.at and gu.at < gd.at)))))
```

For a constraint stating that the gate be opened within timeLimit after a open signal occurred, we could write a dual expression. However, we must ensure that, if the signal close arrives while opening the gate, the gate must start closing immediately. Similarly as for the case of close, we define an abbreviation for the set of open signals.

```
context Gate
def: openSignals : Set(SignalEvent) =
       events->select(e |
         e.isTypeOf(SignalEvent) and e.name = "open")
```

The desired constraint reads as follows:

```
context Gate
constr: always (openSignals->forAll(os |
          sometime (gateUpEvents->exists(gd |
            os.at < gu.at and gu.at <= os.at+timeLimit and
            not gateDownEvents->exists(gd |
              os.at < gd.at and gd.at < gu.at)))))
```

```
xor
sometime (closeSignals->exists(cs |
  cs.at < os.at+timeLimit)))
```

5.3 Timeouts

Timeouts are illustrated using an example consisting of an auctioneer and a set of bidders, such that goods are offered one after the other for a minimum bid. Bidders may place a bid greater than the current one for the item now being offered. Each item is sold to the person who offers the most money for it. The auction is closed by the auctioneer when no new bid has been placed for certain period, which represents a timeout. The auctioneer does this by hammering three times; this closing procedure can be interrupted by a bidder placing a new bid, and again that certain period where no bid is placed has to elapse in order for the auctioneer to be able to restart closing the auction. In other words, this setting presents two nested timeouts. This example is taken from [23]. Let the UML class Auctioneer be defined as shown in Fig. 9.

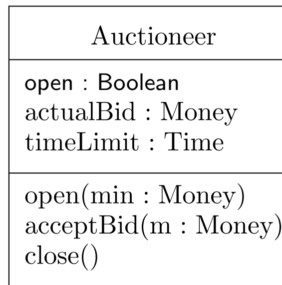


Fig. 9. UML class diagram for auctioneers

The following OCL/RT invariant constraint requires every auction to remain open as long as the constant timeLimit has not elapsed with no bid placed. Moreover, it triggers the operation close if timeLimit elapsed with no valid bid placed.

```
context Auctioneer
def: lastEvent(se : Set(Event)) : Event =
       se->iterate(e; r = undef |
         if r.isUndef() then e
         else if r.at < e.at then e else r endif endif)
def: openEvents : Set(Event) =
       events->select(e |
         e.isTypeOf(CallEvent) and e.operation.name = "open")
def: validBids : Set(Event) =
```

```
        self.events->select(e |
          e.isTypeOf(CallEvent) and
          e.operation.name = "acceptBid" and
          e.at > lastEvent(openEvents).at and
          e.arguments->at(1) > actualBid@e)
def: lastBidOrOpen =
        if lastEvent(validBids).isUndef()
        then lastEvent(openEvents)
        else lastEvent(validBids)
        endif
inv: (not open) iff (lastBidOrOpen.at+timeLimit < Time.now)
action: lastBidOrOpen.at+timeLimit < Time.now
          ==> self.close()
```

Note that the invariant forbids closing if at regular intervals a joker adds one cent to the last bid.

A further constraint is set on the operation close, which presupposes that the auction is open and that timeLimit has elapsed with no new bid having been placed. The post-condition simply states that the auction is indeed closed.

```
context Auctioneer::close()
pre:  open and lastBidOrOpen.at+timeLimit >= Time.now
post: not open
```

The situation for timeouts on ATMs (see Fig. 4) can be easily specified along the same lines.

6 Semantics

We define an operational semantics for OCL/RT constraints that are evaluated over system runs. This semantics conservatively extends the operational semantics for OCL expressions presented in [4], as sketched in Sect. 2.3.

6.1 Semantic Domains

Given a UML model, we denote by Σ the semantic domain of dynamic bases ω over the UML model. We use a map $\omega(\zeta)$ that for each class type in the UML model yields all instances of ζ that exist in ω.

The semantic domain E reflects a given event model such that, in particular, for each event the occurrence time and its arguments can be retrieved by suitable maps *at* and *arguments*. Moreover, we assume a map *relevant* that, given a set M of events in E and an instance υ of a dynamic basis in Σ, yields all those events in M that are relevant or visible for υ.

A system run or *trace* ρ is a finite or infinite sequence of pairs of dynamic bases and finite sets of events

$$(\omega_0, H_0), (\omega_1, H_1), (\omega_2, H_2), \ldots \in (\Sigma \times \wp_{\leq\omega} E)^* \cup (\Sigma \times \wp_{\leq\omega} E)^\infty$$

such that $at(\eta) < at(\eta')$ for all $\eta \in H_i$, $\eta' \in H_j$ with $i < j$. The dynamic basis ω_0 defines the initial system state; ω_n is transformed into ω_{n+1} by a single system step where exactly the events in H_n occur. We denote by $\omega(\rho)_n$ the nth dynamic basis in ρ, by $H(\rho)_n$ the nth event set in ρ, that is ω_n and H_n, respectively. Moreover, $\omega(\rho)_\eta$ denotes the dynamic basis where η occurred and $i(\rho)_\eta$ the index of this state, i.e., $\omega(\rho)_\eta$ is ω_k with $\eta \in H_{k-1}$ and $i(\rho)_\eta$ is k. Finally, we write $i(\rho)_v$ for the first state where instance v exists.

Further requirements on traces may be necessary for particular event models. For the OCL event model in Sect. 4.3, call, start, and termination events linked by call and start, respectively, must occur in this order.

6.2 Operational Rules

The operational semantics derives judgements of the form

$$(\rho, i); \gamma \vdash c \downarrow v$$

where ρ is a trace, i is an index in the trace, γ a variable environment, c an OCL/RT constraint, and v a value. Such a judgement conveys the fact that c evaluates to v at the ith system state in the trace ρ using the variable environment γ.

To begin with, the operational rules for deriving OCL/RT judgements comprise all rules of OCL as defined in [4], but generalising these rules to traces. For example, the rules for evaluating `self` and retrieving an attribute a of an instance originally read:

(Self$^{\downarrow}$)

$$\omega; \gamma \vdash \texttt{self} \downarrow \gamma(\texttt{self})$$

(Feat$^{\downarrow}$)

$$\frac{\omega; \gamma \vdash e \downarrow v}{\omega; \gamma \vdash e\,.\,a \downarrow \mathit{impl}_\omega(a, v)}$$

where $\mathit{impl}_\omega(a, v)$ yields the value of attribute a on instance v in the dynamic basis ω. For OCL/RT, these rules become

(Self$^{\downarrow *}$)

$$(\rho, i); \gamma \vdash \texttt{self} \downarrow \gamma(\texttt{self})$$

(Feat$^{\downarrow *}$)

$$\frac{(\rho, i); \gamma \vdash e \downarrow v}{(\rho, i); \gamma \vdash e\,.\,a \downarrow \mathit{impl}_{\omega(\rho)_i}(a, v)}$$

In particular, all rules are relativised to the event instance at which an expression currently is to be evaluated.

Furthermore, we define rules for evaluating OCL/RT constraints that contain the special OCL/RT instance attribute `events`, or one of the new operators $@\eta$ and `always`:

(Evt$^{\downarrow *}$)

$$\frac{(\rho, i); \gamma \vdash e \downarrow v}{(\rho, i); \gamma \vdash e.\texttt{events} \downarrow \mathit{relevant}(\bigcup_{0 \leq j \leq i} H(\rho)_j, v)}$$

(At$^{\downarrow *}$)

$$\frac{(\rho, i); \gamma \vdash e' \downarrow \eta' \qquad (\rho, i(\rho)_{\eta'}); \gamma \vdash e \downarrow v}{(\rho, i); \gamma \vdash e\texttt{@}e' \downarrow v}$$

(Alw$^{\downarrow *}$)

$$\frac{((\rho, i'); \gamma \vdash c \downarrow v)_{i \leq i'}}{(\rho, i); \gamma \vdash \texttt{always}\ c}$$

Thus $e.\texttt{events}$ comprises all events relevant for an instance e that have occurred up to the current state. An expression $e\texttt{@}e'$ evaluates e at the state where event e' occurred. A constraint `always` c must hold at all states after the current state.

Finally, a rule for general OCL/RT constraints is defined as follows:

(Constr$^{\downarrow *}$)

$$\frac{((\rho, i(\rho)_z); \gamma \vdash c \downarrow v_z)_{z \in \bigcup_{0 \leq i} \omega(\rho)_i(\zeta)}}{(\rho, i); \gamma \vdash \texttt{context}\ \zeta\ \texttt{constr:}\ c \downarrow \bigwedge_z v_z}$$

Hence, an OCL/RT constraint has to hold for all instances z of a class ζ at the state where z is created.

7 Conclusions and Outlook

OCL/RT extends OCL by a general notion of events accompanied by time stamps, satisfaction operators on expressions, and modal operators. This extension enables the specification of several common real-time paradigms for UML models. Moreover, OCL/RT provides means to clarify the semantics of OCL invariants, pre-/post-conditions, and action clauses.

Being based on a trace semantics for systems and on a notion of global time, OCL/RT neither takes into account the possibility that events be only partially ordered nor the concept of different observers with local time, as discussed, e.g., in [17]. Partial orderings may turn indispensable for modelling true concurrency, the notion of observer could prove useful for modelling distributed systems. Though, it might be difficult to reconcile these two notions, since different observers may equip event instances with incompatible partial orderings.

The event models presented for OCL and for signals serve as a basis for the development of the chosen examples, but we do not claim them to be complete. Further event notions like assignment events and return events may have to be

included. The specification of events could be refined by adding conditions that e.g. ensure that events get meaningful time stamps and are placed in time at the right place. In general, the relation between OCL/RT events and UML actions has to be clarified.

Moreover, OCL/RT does not offer means to specify which events force a state transition in the semantic trace. Right now, a successor state has more events, but apart from the obvious choice of change events that cause a state transition, it might also make sense to choose further events for provoking a state transition as well. For instance, close signals of the Gate example can be such kind of events; in this way, the constraint for the gate closed within a certain time after a close signal has arrived may be written just for the last close signal and not for all of them. Given that such a constraint has to hold `always`, i.e., in any state of the semantic trace, then this new constraint will be enough as well as more compact. However, for this proposal to make sense, we have to solve the question of two state changing events occurring simultaneously. Independently, OCL/RT may be further improved by including abbreviations that make it easier to write and read constraints. More ambitiously, a modal logic for reasoning on OCL/RT and UML may be devised.

Acknowledgements. We would like to thank the anonymous referees for their insightful comments.

References

1. José M. Álvarez, Tony Clark, Andy Evans, and Paul Sammut. An Action Semantics for MML. In Gogolla and Kobryn [9], pages 2–18.
2. Thomas Baar. Experiences with the UML/OCL-Approach to Precise Software Modeling. In *Proc. Net.ObjectDays*, Erfurt, 2000. http://i12www.ira.uka.de/ key/doc/2000/baar00.pdf.gz.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Mass., &c., 1998.
4. María Victoria Cengarle and Alexander Knapp. A Formal Semantics for OCL 1.4. In Gogolla and Kobryn [9], pages 118–133.
5. Stefan Conrad and Klaus Turowski. Temporal OCL: Meeting Specification Demands for Business Components. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 10, pages 151–166. Idea Publishing Group, 2001.
6. Bruce P. Douglass. *Real-Time UML*. Addison-Wesley, Reading, Mass., &c., 1998.
7. Bruce P. Douglass. *Doing Hard Time*. Addison-Wesley, Reading, Mass., &c., 1999.
8. Desmond F. D'Souza and Alan C. Wills. *Object, Components, Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., &c., 1998.
9. Martin Gogolla and Cris Kobryn, editors. *Proc. 4th Int. Conf. UML*, volume 2185 of *Lect. Notes Comp. Sci.* Springer, Berlin, 2001.
10. Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Systems with UML*. Addison-Wesley, Reading, Mass., &c., 2000.
11. Anneke Kleppe and Jos Warmer. Extending OCL to Include Actions. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conf. UML*, volume 1939 of *Lect. Notes Comp. Sci.*, pages 440–450. Springer, Berlin, 2000.

12. Anneke Kleppe and Jos Warmer. Unification of Static and Dynamic Semantics of UML. Technical report, Klasse Objecten, 2001. http://www.cs.york.ac.uk/puml/mmf/KleppeWarmer.pdf.
13. Kevin Lano. *Formal Object-Oriented Development*. Formal Approaches to Computing and Information Technology. Springer, London, 1995.
14. Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining UML and Formal Notations for Modelling Real-Time Systems. In *8th Europ. Conf. Software Engineering*, Wien, 2001.
15. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, &c., 1988.
16. Response to OMG RfP ad/00-09-03 "UML 2.0 OCL". Submission, OMG, 2001. http://cgi.omg.org/cgi-bin/doc?ad/01-08-01.
17. Response to OMG RfP ad/99-03-13 "Schedulability, Performance, and Time". Revised submission, OMG, 2001. http://cgi.omg.org/cgi-bin/doc?ad/01-06-14.
18. Object Management Group. Unified Modeling Language Specification, Version 1.4. Specification, OMG, 2001. http://cgi.omg.org/cgi-bin/doc?formal/01-09-67.
19. Object Management Group. Unified Modeling Language Specification (Action Semantics), Version 1.4. Specification, OMG, 2002. http://cgi.omg.org/cgi-bin/doc?ptc/02-01-09.
20. Mark Richters and Martin Gogolla. OCL — Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *Lect. Notes Comp. Sci.*, pages 38–63. Springer, Berlin, 2002.
21. Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
22. Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems. White paper, Rational Software Corp., 1998. http://www.rational.com/media/whitepapers/umlrt.pdf.
23. Shane Sendall and Alfred Strohmeier. Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML. In Martin Gogolla and Cris Kobryn, editors, *Proc. 4th Int. Conf. UML*, volume 2185 of *Lect. Notes Comp. Sci.*, pages 391–405. Springer, Berlin, 2001.
24. Kerry Trentelman and Marieke Huisman. Extending JML Specifications with Temporal Logic. In *Proc. 9th Int. Conf. Algebraic Methodology And Software Technology*, 2002. To appear.
25. Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison–Wesley, Reading, Mass., &c., 1999.