

UML 2.0 Interactions with OCL/RT Constraints

Daniel Calegari García
 Instituto de Computación
 Universidad de la República
 Montevideo 11300, Uruguay
 Email: dcalegar@fing.edu.uy

María Victoria Cengarle
 Institut für Informatik
 Technische Universität München
 Garching bei München 85748, Germany
 Email: cengarle@in.tum.de

Nora Szasz
 Facultad de Ingeniera
 Universidad ORT
 Montevideo 11100, Uruguay
 Email: szasz@ort.edu.uy

Abstract—The Unified Modeling Language 2.0 Interactions language describes inter-component behavior. However, it cannot define meaningful time constraints. OCL for Real Time is a language for real-time constraints specification well-suited for describing constraints on interactions. This work defines a formal semantics for the merger of those languages. The semantics allows the recognition of valid and invalid behaviors of a system with time constraints. An analysis of the properties derived from the semantics is also done. In particular, the notions of refinement of interactions and refinement of constraints, intended for formal verification, are explored.

I. INTRODUCTION

Behavioral specifications describe how system elements interact over time. Inter-component behavior can partially be specified using the Unified Modeling Language 2.0 Interactions language [1]. This language is not expressive enough, it lacks constructs for the specification of constraints. The Object Constraint Language (OCL [2]) is part of the UML and it is a natural candidate to complement the language for interactions. The OCL, however, has some limitations: it offers neither time nor signal handling constructs, it is incapable of expressing general liveness properties or performance aspects of systems conveniently. OCL for Real Time (OCL/RT) is an extension of OCL which covers these aspects [3].

Our concern is to set the ground for lifting UML 2.0 interactions to a formal design technique. The UML 2.0 specification introduces some guidelines for the formal understanding of interactions. However, the specification is rather vague and obscure in some aspects since it is given in natural language. In [4] a denotational semantics is defined intended to verify *traces* (sequences of messages) against interactions specifications; this semantics allows to discern when a trace is positive (a valid execution), negative (an invalid execution), or inconclusive for a given interaction.

We go a step beyond and relate the language of interactions with OCL/RT. We propose a formal semantics for the merger of UML 2.0 Interactions language and OCL/RT, based on the works mentioned above. The semantics recognizes valid and invalid behavior of a system execution considering both interaction structure and constraints satisfaction. We also study

This work has been partially sponsored by the Uruguayan Technological Development Program PDT Project 54-106 "Extensions of UML models for behavioral design of real-time systems and product lines", and the DFG project rUML.

some properties defined in terms of the semantics, like refinement of interactions and constraints. These definitions are intended for formal verification of system properties.

Some languages like UML/RT [5] and ROOM [6] extend UML for real-time specifications. These works do not present formal semantics and only take into account basic timing annotations. Some other works define the semantics of UML 2.0 Interaction as those in [4, 7–11]. However, only a few of them consider interactions with basic time constraints. Finally, there are several approaches for coping with time and events in OCL and related specification languages for the UML. As far as we know, only in [12] there is an OCL extension to be used with interactions; this work focuses on liveness properties and does not consider any formal semantics. A detailed comparison between related works can be found in [13].

The remainder of this paper is structured as follows. Section II briefly introduces the UML 2.0 Interactions and OCL/RT languages. A formal semantics for an enhanced UML 2.0 Interactions language with OCL/RT constraints support is presented in Section III. The semantics is used in Section IV to review the notions of implementation and refinement of an interaction, and to present the notion of refinement of constraints. Section V presents a short summary of the practical application of the concepts developed in previous sections. Finally, Section VI presents a short summary with concluding remarks and an outline of future work.

II. BACKGROUND

We briefly review the background of our work. Although we assume the reader is familiar with UML interactions, we present UML 2.0 Interaction and OCL/RT languages.

A. UML 2.0 Interactions

UML 2.0 Interactions [1] describe possible message exchanges between system instances. A *message* is a communication between two instances and is defined by two events, representing the sending and the receiving of that message.

Interactions are graphically represented by Sequence Diagrams. They focus on message interchange among lifelines. The instances involved in the diagram are on the horizontal axis, each one is represented by an object rectangle with the object or class name. Below the rectangle, a vertical dashed line called lifeline represents the instance execution over time. Communications between instances are represented

as horizontal message arrows between the instance lifelines. Sequence Diagrams are read from top to bottom to view the exchange of messages taking place as time passes.

Each interaction may contain sub-interactions called *interaction fragments*, which can be structured and combined using *interaction operators*. The resulting *combined fragments* enclose messages or fragments within a rectangular frame with the name of the operator shown in a pentagram at the upper-left corner. These define special behavior like sequential, parallel, and iterative composition of interactions, alternative, optional and invalid interactions, and interactions considering and ignoring a set of messages, among others. The UML also introduces visual elements to express time and duration observations, and time and duration constraints.

As an example, consider the interaction in Figure 1 where an instance **a** of **A** sends the message **n** to an instance **b** of **B** and **b** responds sending up to **N** times the message **m**. There is a visual constraint restricting the duration of message **n** between 0 and 3 time units.

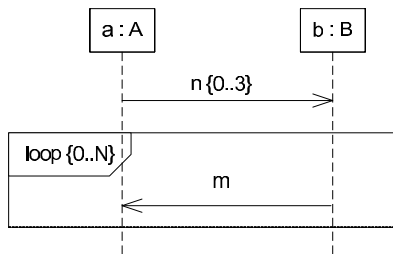


Figure 1. Interaction with a combined fragment

[4] defines the abstract syntax of the language by a context-free grammar which does not consider all the features introduced by the UML 2.0 specification. They also define the semantics of interactions, which decides when a *trace* (sequence of messages) is positive (a valid execution) or negative (an invalid execution) for some interaction.

B. OCL/RT

OCL/RT (OCL for Real Time) is defined in [3] as an extension of OCL 1.4 for real-time and reactive systems constraints specification. The language is based on the notion of traces of events with timestamps and allows the specification of the temporal behavior of a system. The constraints are evaluated over sequences of system states (system execution) instead of just on a given state as OCL does.

For the definition of OCL/RT constraints, that are evaluated over a sequence of system states, a new clause is introduced

```
context C
constr: c
```

where **C** is a classifier and **c** an OCL/RT constraint.

OCL is furthermore extended with particular OCL/RT features: the modality `always c` and the satisfaction operator `@ η` . The modality `always c` satisfies the constraint **c** over a system run when the constraint evaluates to true in every system state. The satisfaction operator `@ η` yields the value of a given expression at the system state where event η occurred.

System states are formalized 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. OCL/RT constraints restrict system runs. A *system run* ρ is a finite or infinite sequence of pairs of dynamic bases and finite sets of events. OCL/RT is equipped with an operational semantics that permits the derivation of judgments of the form

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

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

III. CONSTRAINED INTERACTIONS

The abstract syntax of the fragment of the language of UML 2.0 defined in [4] is extended with OCL/RT time constraints as shown in Figure 2. The production rule `constraint(Term, Interaction)` considers an OCL/RT term and the interaction over which the term is evaluated; the *Term* must be of boolean type, expressions of other type are not permitted.

```
Interaction ::= Basic
              | CombinedFragment
CombinedFragment ::= strict (Interaction, Interaction)
                  | seq (Interaction, Interaction)
                  | par (Interaction, Interaction)
                  | loop (Nat, (Nat| $\infty$ ), Interaction)
                  | ignore (Messages, Interaction)
                  | alt (Interaction, Interaction)
                  | neg (Interaction)
                  | assert (Interaction)
                  | constraint (Term, Interaction)
```

Figure 2. Abstract syntax of interactions with OCL/RT constraints (fragment)

In order to consider timed events the meta-class `Event` (UML 1.5) was extended in [3]. Each event shows the time at which it occurred by a link to the primitive data type `Time` that represents the global system time, as in the original definition. It is assumed 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.

We extend the event meta-model as shown in Figure 3 in order to model a system based on message passing through instances. Every (partial) message is composed of at least one and at most two events representing the sending of that message by an instance (of a classifier) and the reception of it by another instance (possibly the same). Any classifier instance is linked to all its current events, message events among them. There is also a well-formedness rule on system runs: send and receive events of the same message must occur in this order.

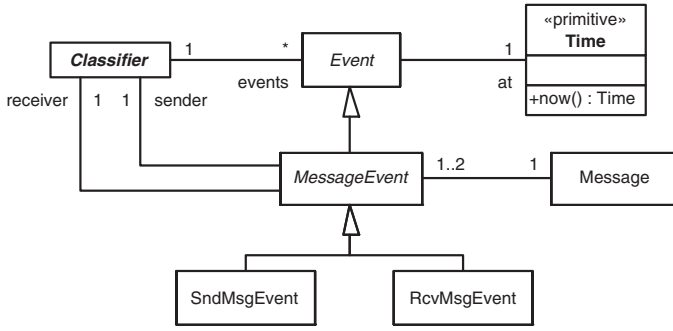


Figure 3. Event model for messages

As an example, consider the interaction in Figure 1. The constraint on the first message can be expressed as follows:

```

context B
def: sndN : Event =
  events->select(e | e.ocIsTypeOf(SndMsgEvent)
    and e.message.name = "n")->any()
def: rcvN : Event =
  events->select(e | e.ocIsTypeOf(RcvMsgEvent)
    and e.message.name = "n")->any()
constr: always((rcvN.at <= sndN.at + 3)@rcvN)
    
```

The abstract syntax term representing the diagram of Figure 1 is thus

$$\text{seq}(\text{constraint}(\varphi_1, B_1), \text{loop}(0, N, B_2))$$

where B_1 is a basic interaction representing the dispatch and arrival of message n , B_2 is a basic interaction representing the dispatch and arrival of message m , and φ_1 is the above constraint. Notice that there exists an implicit weak sequence between B_1 and the loop.

We adapt the formal semantics in [4] to recognize valid and invalid behavior for some given interaction with constraints. We use a more expressive notion of trace: a system run. This notion allows the use of OCL/RT constraints in interactions and the fusion of both semantics. As in [4] the semantics is divided into two fragments: one that does not contain occurrences of the `neg` nor `assert` operators (positive fragment), and the other that contains them (negative fragment). In what follows we first introduce the unified semantic domains and then present both fragments of the semantics.

A. Semantic Domains

We assume four primitive domains for *events* \mathbb{E} , *messages* \mathbb{M} , *abstract time points* \mathbb{T} and natural numbers \mathbb{N} . An event $\eta \in \mathbb{E}$ is either of the form $\text{snd}(s, r, m)$ or of the form $\text{rcv}(s, r, m)$, representing the dispatch and the arrival of message m from *sender* instance s to *receiver* instance r , respectively. Events are univocally identified, i.e., two arbitrary events, even if they are both a send (or receive) event of the same message from the same sender to the same receiver, can be distinguished.

We let certain events be anonymous or unobservable by replacing them by a predefined *silent event* denoted by τ . We define the domain \mathbb{E}_τ as $\mathbb{E} \cup \{\tau\}$. A silent event represents an

event that is not of interest for some given interaction, e.g. an attribute value change, such that its occurrence is nevertheless considered for it may cause a state change. System runs contain every event occurred during the system execution, while interactions can specify partial behavior of the system.

We say that the instance s is active for $\text{snd}(s, r, m)$ and, similarly, that the instance r is active for $\text{rcv}(s, r, m)$. We define a binary, symmetric conflict relation $\approx \subseteq \mathbb{E} \times \mathbb{E}$ on events: If an instance is active for both events η_1 and η_2 then $\eta_1 \approx \eta_2$. Trivially, $\tau \not\approx \eta$ for any $\eta \in \mathbb{E}_\tau$.

Given an event $\eta \in \mathbb{E}$, the occurrence time of the event can be retrieved by the map $at : \mathbb{E} \rightarrow \mathbb{T}$. The domain \mathbb{T} is equipped with a total ordering relation \leq for comparing time values and a binary operation $+$ for adding time values.

We write $\text{lin}_{at}(E)$ for all possible linearizations of the set of events E given by the function at , i.e., $l \in \text{lin}_{at}(E)$ if, and only if l is the isomorphism class $[(X, \leq_X, \lambda_X)]$ of the totally ordered, labeled sets (X, \leq_X, λ_X) with $\lambda_X : X \rightarrow E$ bijective and such that $\forall x_1, x_2 \in X. at(\lambda_X(x_1)) < at(\lambda_X(x_2)) \implies x_1 \leq_X x_2$ and $\forall x_1, x_2 \in X. x_1 \leq_X x_2 \vee x_2 \leq_X x_1$.

A basic interaction is given by an event-labeled pomset [14] $[(X, \leq_X, \lambda_X)]$ such that conflicting events do not occur concurrently, i.e., if $x_1, x_2 \in X$ with $\lambda_X(x_1) \approx \lambda_X(x_2)$, then $x_1 \leq_X x_2$ or $x_2 \leq_X x_1$. An event-labeled pomset is a pomset with an injective labeling function $\lambda_X : X \rightarrow \mathbb{E}$ (or simply λ). The order $x_1 \leq_X x_2$ is interpreted as “the occurrence of event $\lambda(x_1)$ precedes the occurrence of event $\lambda(x_2)$ ”. The empty interaction $[(\emptyset, \emptyset, \emptyset)]$ is denoted by ε .

We write $\text{lin}(p)$ for all possible linearizations of a pomset p , i.e., all traces that extend the ordering of p : $[(X', \leq_{X'}, \lambda_{X'})] \in \text{lin}([(X, \leq_X, \lambda_X)])$ if, and only if, $X' = X$, $\lambda_{X'} = \lambda_X$, and $\leq_X \subseteq \leq_{X'}$ where $x_1 \leq_{X'} x_2$ or $x_2 \leq_{X'} x_1$ for all $x_1, x_2 \in X'$.

System states are formalized by *dynamic bases*. A *system-run* ρ is a finite or infinite sequence of pairs of dynamic bases and finite sets of events possibly containing the silent event τ

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

such that if $\eta_1 \in H_i \setminus \{\tau\}$ and $\eta_2 \in H_j \setminus \{\tau\}$ with $i < j$, then $at(\eta_1) < at(\eta_2)$. The dynamic basis ω_0 defines the initial system state; ω_n is transformed into ω_{n+1} by a single system step where the events in H_n occur.

We denote by $\omega(\rho)_n$ the n th dynamic basis in ρ , by $H(\rho)_n$ the n th set of events in ρ , and by $\rho(n)$ the n th pair of dynamic basis and set of events in the system run ρ . The empty system run is denoted by ε .

Given a system run ρ , we define:

- the partition of a system run ρ^* as the set of pairs (ρ_1, ρ_2) with ρ_1 and ρ_2 system runs such that $\forall j \in \mathbb{N}. \omega(\rho_i)_j = \omega(\rho)_j$ ($i = 1, 2$) and $\forall j \in \mathbb{N}. \{H(\rho_1)_j, H(\rho_2)_j\}$ is a partition of the set of events $H(\rho)_j$.
- $\text{rem}(\rho)$ as the system run ρ' resulting from removing every pair (ω_i, \emptyset) from ρ .
- concurrence ρ_{\parallel} as the set of pairs of system runs $(\text{rem}(\rho_1), \text{rem}(\rho_2))$ such that $(\rho_1, \rho_2) \in \rho^*$.
- strict sequencing $\rho;$ as the set of pairs $(\rho_1, \rho_2) \in \rho_{\parallel}$ such that $\exists k \in \mathbb{N}. \forall i \in \mathbb{N}. 0 \leq i \leq k \implies H(\rho_1)_i = H(\rho)_i$ and

- $k < i \Rightarrow H(\rho_2)_{i-k} = H(\rho)_i$.
- weak sequencing $\rho;_{\times}$ as the set of pairs of system runs $(\rho_1, \rho_2) \in \rho_{\parallel}$ such that $\forall \eta_1 \in \bigcup_n H(\rho_1)_n \setminus \{\tau\}, \eta_2 \in \bigcup_n H(\rho_2)_n \setminus \{\tau\} \cdot \eta_1 \times \eta_2 \Rightarrow at(\eta_1) < at(\eta_2)$.
 - $filter(M)(\rho)$, with $M \in \mathbb{M}$, as the set of system runs that result of the removal of some events from ρ whose messages are labelled by elements in M and subsequently eliminating pairs with empty set of events, i.e., $rem(\rho') \in filter(M)(\rho)$ if $\forall i \in \mathbb{N} \cdot \omega(\rho)_i = \omega(\rho')_i \wedge H(\rho')_i \subseteq H(\rho)_i$, and any $\eta \in H(\rho)_i \setminus H(\rho')_i$ is of the form either $snd(s, r, m)$ or $rcv(s, r, m)$ with $m \in M$.

B. The Positive Fragment

The semantics of the positive fragment of the language (with no occurrences of negation and assertion) is defined by an inductive positive satisfaction relation between system runs and interactions, denoted by $\rho \models_p S$ and read system run ρ positively satisfies interaction S , as shown in Figure 4.

$$\begin{aligned}
 \rho \models_p B & \text{ if } \text{lin}_{at}(\bigcup_i H(\rho)_i \setminus \{\tau\}) \subseteq \text{lin}(B) \\
 \rho \models_p \text{strict}(S_1, S_2) & \text{ if } \exists (\rho_1, \rho_2) \in \rho; \cdot \rho_i \models_p S_i \ (i = 1, 2) \\
 \rho \models_p \text{seq}(S_1, S_2) & \text{ if } \exists (\rho_1, \rho_2) \in \rho;_{\times} \cdot \rho_i \models_p S_i \ (i = 1, 2) \\
 \rho \models_p \text{par}(S_1, S_2) & \text{ if } \exists (\rho_1, \rho_2) \in \rho_{\parallel} \cdot \rho_i \models_p S_i \ (i = 1, 2) \\
 \rho \models_p \text{loop}(0, 0, S) & \text{ if } \bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset \\
 \rho \models_p \text{loop}(0, n + 1, S) & \text{ if } \bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset \vee \\
 & \rho \models_p \text{seq}(S, \text{loop}(0, n, S)) \\
 \rho \models_p \text{loop}(m + 1, n + 1, S) & \text{ if } \rho \models_p \text{seq}(S, \text{loop}(m, n, S)) \\
 \rho \models_p \text{loop}(m, \infty, S) & \text{ if } \exists n \geq m \cdot \rho \models_p \text{loop}(m, n, S) \\
 \rho \models_p \text{ignore}(M, S) & \text{ if } \exists \rho_1 \cdot \rho_1 \in filter(M)(\rho) \wedge \rho_1 \models_p S \\
 \rho \models_p \text{alt}(S_1, S_2) & \text{ if } \rho \models_p S_1 \vee \rho \models_p S_2 \\
 \rho \models_p \text{constraint}(\varphi, S) & \text{ if } (\rho, 0); \emptyset \vdash \varphi \downarrow \text{true} \wedge \rho \models_p S
 \end{aligned}$$

Figure 4. Semantics of the positive fragment

The base case considers a system run possibly containing one or more occurrences of the silent event τ . Silent events are ignored in order to verify the satisfaction of a given system run. A system run satisfies a basic interaction if every possible linearization of all its events but τ (with respect to their occurrence time) is included in the linearization of the basic interaction (with respect to its partial ordering).

In order to satisfy `strict`, `seq` and `par` operators, the system run is partitioned in two and the satisfaction is verified for each resulting system run. Performing a partition of a system run may produce empty set of events H_i in the resulting system runs. Those sets are removed, with their corresponding dynamic bases, since they represent transitions taken by some other parts of the interaction.

The `ignore` operator is evaluated by filtering some events of the system run. The `loop` operator is equivalent to its unfolding as weak sequencing of its interaction arguments. The `alt` operator is evaluated as the disjunction of the satisfaction of the system run in both interaction fragments.

Finally, there is a satisfaction rule to evaluate whether a system run positively satisfies an OCL/RT-constrained interaction. The `constraint` fragment is interpreted as a conjunction

where both the constraint and the interaction must be satisfied in order to consider a system run positive. In this case, it is not possible to consider positive a system run where constraints are not valid, not even the empty system run.

Consider the example in Figure 1 and the system runs

$$\begin{aligned}
 \rho_1 &= (\omega_0, \{snd(a, b, n, at_1), rcv(a, b, n, at_2)\}) \text{ and} \\
 \rho_2 &= (\omega_0, \{snd(a, b, n, at_1), rcv(a, b, n, at_2)\}) \\
 &\quad (\omega_1, \{snd(b, a, m, at_3), rcv(b, a, m, at_4)\})
 \end{aligned}$$

Both system runs positively satisfy $\text{seq}(B_1, \text{loop}(0, N, B_2))$. In the case of ρ_1 , it satisfies the interaction since the operator `loop` accepts the empty system run and $(\rho_1, \varepsilon) \in \rho_{1;_{\times}}$. However, the system runs are valid for the whole interaction depending on the assignments for at_i .

Table I
TIME ASSIGNMENTS FOR THE EXAMPLE

at_i	Assignment 1	Assignment 2	Assignment 3
at_1	0	0	0
at_2	2	4	3
at_3	3	5	7
at_4	4	6	9

With the first and third time assignments in Table I, the reception of n is done before 3 time units since the sending of n . In consequence, φ_1 is satisfied and both system runs are valid. The proof of this is developed as follows:

$$\frac{\rho_2(0) \models_p B_1 \quad (\rho_2(0), 0); \emptyset \vdash \varphi_1 \downarrow \text{true}}{\rho_2(0) \models_p \text{constraint}(\varphi_1, B_1)}$$

then,

$$\frac{\rho_2(0) \models_p \text{constraint}(\varphi_1, B_1) \quad \rho_2(1) \models_p \text{loop}(0, N, B_2) \quad (\rho_2(0), \rho_2(1)) \in \rho_{2;_{\times}}}{\rho_2 \models_p \text{seq}(\text{constraint}(\varphi_1, B_1), \text{loop}(0, N, B_2))}$$

If we consider the second assignment, the constraint is not satisfied and consequently the system run is not positive but negative as we show next.

C. The Negative Fragment

The semantics of the negative fragment of the language (with occurrences of negation and assertion) is defined by an inductive negative satisfaction relation between system runs and interactions, denoted by $\rho \models_n S$ and read system run ρ negatively satisfies interaction S , as shown in Figure 5.

As in [4] we regard the empty system run and also those with events besides possibly τ as being positive for `neg`(S). We change the negative satisfaction rule following the same idea as sequential operators: after traversing a negative fragment a system run will always be negative no matter what happens afterwards. A system run positively satisfying S is positive for `assert`(S), otherwise the system run is negative.

For the operators `strict` and `seq` we adopt the view that the only system runs that are negative are those that either run through the first operand negatively or fulfil the first operand positively and the second operand negatively.

$$\begin{aligned}
 \rho \models_p \text{neg}(S) & \text{ if } \bigcup_i H(\rho)_i \setminus \{\tau\} = \emptyset \\
 \rho \models_p \text{assert}(S) & \text{ if } \rho \models_p S \\
 \rho \models_n \text{strict}(S_1, S_2) & \text{ if } \exists(\rho_1, \rho_2) \in \rho; \cdot (\rho_1 \models_n S_1 \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2)) \\
 \rho \models_n \text{seq}(S_1, S_2) & \text{ if } \exists(\rho_1, \rho_2) \in \rho; \times (\rho_1 \models_n S_1 \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2)) \\
 \rho \models_n \text{par}(S_1, S_2) & \text{ if } \exists(\rho_1, \rho_2) \in \rho \parallel \cdot ((\rho_1 \models_n S_1 \wedge \rho_2 \models_n S_2) \vee (\rho_1 \models_n S_1 \wedge \rho_2 \models_p S_2) \vee (\rho_1 \models_p S_1 \wedge \rho_2 \models_n S_2)) \\
 \rho \models_n \text{loop}(0, n+1, S) & \text{ if } \rho \models_n \text{seq}(S, \text{loop}(0, n, S)) \\
 \rho \models_n \text{loop}(m+1, n+1, S) & \text{ if } \rho \models_n \text{seq}(S, \text{loop}(m, n, S)) \\
 \rho \models_n \text{loop}(m, \infty, S) & \text{ if } \exists n \geq m. \rho \models_n \text{loop}(m, n, S) \\
 \rho \models_n \text{ignore}(M, S) & \text{ if } \exists \rho_1. \rho_1 \in \text{filter}(M)(\rho) \wedge \rho_1 \models_n S \\
 \rho \models_n \text{alt}(S_1, S_2) & \text{ if } \rho \models_n S_1 \wedge \rho \models_n S_2 \\
 \rho \models_n \text{neg}(S) & \text{ if } \exists(\rho_1, \rho_2) \in \rho; \cdot \rho_1 \models_p S \\
 \rho \models_n \text{assert}(S) & \text{ if } \rho \not\models_p S \\
 \rho \models_n \text{constraint}(\varphi, S) & \text{ if } (\rho, 0); \emptyset \vdash \varphi \downarrow \text{false} \vee \rho \models_n S
 \end{aligned}$$

Figure 5. Semantics of the negative fragment

A similar instance is taken towards `par` where at least one operand has to be run through negatively in order to make a run negative. The `loop` and `ignore` operators follow the same idea as in the positive fragment, while for the `alt` operator both operands have to be run through negatively.

Finally, there is a satisfaction rule to evaluate whether a system run negatively satisfies an OCL/RT constrained interaction. The negative rule is interpreted as the “negation” of the positive satisfaction rule: either the constraint evaluates to `false` or the system run negatively satisfies the interaction.

We had that the system runs ρ_1 and ρ_2 of above satisfy the interaction of Figure 1 depending on the value of at_i . For the second time assignment, φ_1 is not satisfied and consequently the system run is negative for the interaction:

$$\frac{\begin{array}{l} \rho_2(0) \models_n \text{constraint}(\varphi_1, B_1) \\ \rho_2(1) \models_p \text{loop}(0, N, B_2) \quad (\rho_2(0), \rho_2(1)) \in \rho_2; \times \end{array}}{\rho_2 \models_n \text{seq}(\text{constraint}(\varphi_1, B_1), \text{loop}(0, N, B_2))}$$

D. Summary of Constraints Satisfaction

The satisfaction rule for constraints, defined in the previous section, states that a system run ρ is positive for an interaction $\text{constraint}(\varphi, S)$ if the system run positively satisfies the interaction S and also φ evaluates to `true`. If ρ negatively satisfies the interaction S or φ evaluates to `false`, the system run ρ negatively satisfies the interaction $\text{constraint}(\varphi, S)$. In any other case, the system run ρ is considered inconclusive. Finally, a system run can be both positive and negative. In this case, we speak of an *overspecified* interaction.

In the following, by $\rho \vdash \varphi \downarrow v$ abbreviates $(\rho, 0); \emptyset \vdash \varphi \downarrow v$.

IV. IMPLEMENTATION AND REFINEMENT

Refinement means adding information to a specification to make it closer to an implementation. This can be formally defined using the notion of refinement by model inclusion:

a concrete specification refines an abstract specification if any model of the concrete specification is also a model of the abstract one. Below, we adapt the set of definitions originally given in [4] for implementation and refinement of an interaction to our semantic domains.

Definition 1 (Implementation of Interactions). A *process* is an arbitrary set of system runs. A process I is an implementation of an interaction S , written $I \models S$, if

1. $\exists \rho \in I. \rho \models_p S$
2. $\forall \rho \in I. \rho \not\models_n S$

An interaction S is *implementable* if there is a process I such that $I \models S$.

Definition 2 (Refinement). An interaction S' refines an interaction S , written $S \rightsquigarrow S'$, if any implementation of S' is also an implementation of S , i.e., $\forall I. I \models S' \Rightarrow I \models S$.

With these definitions we can increase the set of refinement rules given in [4]. We show one of them in the following lemma. It is straightforward to prove that $S \rightsquigarrow \text{constraint}(\varphi, S)$ for any constraint φ and interaction S .

Lemma 1. *Let S and S' be interactions with S' implementable and not overspecified, let φ be a constraint. If $S \rightsquigarrow S'$, then $\text{constraint}(\varphi, S) \rightsquigarrow \text{constraint}(\varphi, S')$.*

Proof: Let I be a process with $I \models \text{constraint}(\varphi, S')$.

(a) $\exists \rho \in I. \rho \models_p \text{constraint}(\varphi, S')$. Thus (by the positive satisfaction relation) $\rho \vdash \varphi \downarrow \text{true} \wedge \rho \models_p S'$, and thus $\rho \not\models_n S'$ as S' is not overspecified. Then $\rho \models_p S$ by definition of \rightsquigarrow , and thus, $\rho \models_p \text{constraint}(\varphi, S)$.

(b) $\forall \rho \in I. \rho \not\models_n \text{constraint}(\varphi, S')$. Thus (by the negative satisfaction relation) $\rho \not\vdash \varphi \downarrow \text{false} \wedge \rho \not\models_n S'$. Now, let I' be a process such that $I' \models S'$. Then, $I' \cup \{\rho\} \models S'$, and thus $I' \cup \{\rho\} \models S$ because $S \rightsquigarrow S'$. Finally, $\rho \not\models_n S$, and hence, $\rho \not\models_n \text{constraint}(\varphi, S)$.

From (a) and (b) we conclude that $I \models \text{constraint}(\varphi, S)$, thus $\text{constraint}(\varphi, S) \rightsquigarrow \text{constraint}(\varphi, S')$. ■

In analogy to Definition 2, we define constraint refinement.

Definition 3 (Constraint Refinement). A constraint φ' refines a constraint φ , written $\varphi \rightsquigarrow \varphi'$ if, for any system run ρ ,

1. $\rho \vdash \varphi' \downarrow \text{true} \Rightarrow \rho \vdash \varphi \downarrow \text{true}$
2. $\rho \vdash \varphi' \downarrow \text{undef} \Rightarrow \rho \not\vdash \varphi \downarrow \text{false}$

We can also define a further refinement rule.

Lemma 2. *Let φ and φ' be constraints, let S be an interaction. If $\varphi \rightsquigarrow \varphi'$, then $\text{constraint}(\varphi, S) \rightsquigarrow \text{constraint}(\varphi', S)$.*

Proof: Let I be a process with $I \models \text{constraint}(\varphi', S)$.

(a) $\exists \rho \in I. \rho \models_p \text{constraint}(\varphi', S)$. Thus (by the positive satisfaction relation) $\rho \vdash \varphi' \downarrow \text{true} \wedge \rho \models_p S$. Then (by Definition 3) $\rho \vdash \varphi \downarrow \text{true}$, and finally, $\rho \models_p \text{constraint}(\varphi, S)$.

(b) $\forall \rho \in I. \rho \not\models_n \text{constraint}(\varphi', S)$. Thus (by the negative satisfaction relation) one of the following cases occur: (1) $\rho \vdash \varphi' \downarrow \text{true} \wedge \rho \not\models_n S$. Then (by Definition 3) $\rho \vdash \varphi \downarrow \text{true}$, and finally $\rho \not\models_n \text{constraint}(\varphi, S)$;

(2) $\rho \vdash \varphi' \downarrow \text{undef} \wedge \rho \not\llcorner_n S$. Then (by Definition 3) $\rho \not\llcorner \varphi \downarrow \text{false}$ and in consequence $\rho \not\llcorner_n \text{constraint}(\varphi, S)$

From (a) and (b) we conclude that $I \models \text{constraint}(\varphi, S)$, and thus the thesis holds. ■

The interaction of the example in Figure 1 can be refined by adding a new constraint φ_2 which states that every message m must be sent within 3 time units since the reception of n :

```
context B
def: sndM : Set(Event) =
  events->select(e | e.oclIsTypeOf(SndMsgEvent)
    and e.message.name = "m")
def: rcvN : Event =
  events->select(e | e.oclIsTypeOf(RcvMsgEvent)
    and e.message.name = "n")->any()
constr: always(sndM->forall(e |
  e.at < rcvN.at + 3))
```

The third assignment of Table I makes the system run invalid for $\text{constraint}(\varphi_2, \text{seq}(\text{constraint}(\varphi_1, B_1), \text{loop}(0, N, B_2)))$ while it is valid for $\text{seq}(\text{constraint}(\varphi_1, B_1), \text{loop}(0, N, B_2))$. The interactions can also be refined by a constraint refinement. If we restrict the duration of n to $\{0..2\}$, only the first assignment makes the system run valid.

V. IN PRACTICE

A practical analysis is mandatory for a fine-tuning and a careful utilization of the proposal. [13] presents some examples, taken from the literature, that apply our approach; these examples are not repeated here due to space reasons.

UML 2.0 Interactions can graphically specify basic timing constraints, as defined in [1]. These kind of constraints are simple time and duration observations and/or constraints. They can be expressed with OCL/RT assigning to them a concrete meaning in the evaluation of a system-run. Besides, OCL/RT allows specifying more complex constraints involving also system instances, message events and global variables. Consequently our approach is one of the most powerful proposals in order to specify real-time constraints in UML 2.0 Interactions.

As far as we know, only the work in [12] analyzes an OCL extension to be used with interactions. There, the problem of synchronization and the specification of liveness properties (normally expressed at the level of state diagrams) are addressed. Our work does not address the problem of synchronization but the use of OCL/RT allows specifying some liveness properties. We think that our work and that in [12] are complementary, and a fusion of both can be made in order to investigate the relation between constrained interactions and timed state machines.

VI. CONCLUSIONS AND OUTLOOK

This work contributes to the clarification on the use of time constraints in inter-component behavioral specifications. Based on previous works, we developed a semantics for an enhanced UML 2.0 Interactions language with OCL/RT constraints support. The resulting denotational semantics evaluates the positivity/negativity of a system run within an interaction, considering both interaction structure and constraints satisfaction. To the best of our knowledge, our semantics is the only one

with an expressive support of complex constraints in interactions. We also reviewed the notions of implementation and refinement of interactions, we extended the set of refinement rules intended for formal verification, and we introduced the notion of constraint refinement.

Since some kind of constraints are not relevant for the interactions, OCL/RT is not used in its full potential. Future work is needed in order to extend the language to specify state-based constraints suited for temporal logic reasoning. Moreover, system runs do not consider termination events for operations nor synchronization of messages. Both concepts are needed to develop an integrated behavioral specification model connecting other development activities as intra-component specifications and design by contract.

Our approach tends to ensure software quality attributes by verification of the generated models at early development stages. A formal background helps to prevent from ambiguous, imprecise, contradictory and error-prone specifications. This approach allows a tighter control of the processes and improves software reliability. Our intention is not capricious since software modeling and formal verification are strongly related in growing engineering approaches nowadays.

ACKNOWLEDGMENT

The authors are indebted to Alexander Knapp for fruitful discussions on the subject.

REFERENCES

- [1] "UML 2.0 Superstructure," Object Management Group, Formal Specification formal/05-07-04, 2005. [Online]. Available: <http://www.omg.org/docs/formal/05-07-04>
- [2] "UML 2.0 Object Constraint Language," Object Management Group, Formal Specification formal/06-05-01, 2006. [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/06-05-01>
- [3] M. V. Cengarle and A. Knapp, "Towards OCL/RT," *Lecture Notes in Computer Science*, vol. 2391, pp. 390–409, 2002.
- [4] M. V. Cengarle and A. Knapp, "UML 2.0 Interactions: Semantics and Refinement," in *Proc. 3rd Intl. Work. on Critical Systems Development with UML*. Technische Universität München, 2004, pp. 85–99.
- [5] B. P. Douglass, *Real-Time UML: Advances in the UML for Real-Time Systems*, 3rd ed. Addison Wesley, 2004.
- [6] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. Wiley, 1994.
- [7] H. Störrle, "Semantics of Interactions in UML 2.0," *Proc. 2003 IEEE Symp. on Human Centric Computing Languages and Environments*, pp. 129–136, 2003.
- [8] Ø. Haugen, R. Kobro, K. Husa, and K. Stølen, "Why Timed Sequence Diagrams Require Three-Event Semantics," *Lecture Notes in Computer Science*, vol. 3466, pp. 1–25, 2005.
- [9] R. Grosu and S. Smolka, "Safety-Liveness Semantics for uml 2.0 sequence diagrams," in *Proc. 5th Intl. Conf. on Application of Concurrency to System Design*. IEEE Computer Society, 2005, pp. 6–14.
- [10] Y. Hammal, "Branching Time Semantics for UML 2.0 Sequence Diagrams," *Lecture Notes in Computer Science*, vol. 4229, pp. 259–274, 2006.
- [11] A. Cavarra and J. Küster-Filipe, "Formalizing Liveness-Enriched Sequence Diagrams Using ASMs," *Lecture Notes in Computer Science*, vol. 3052, pp. 62–77, 2004.
- [12] A. Cavarra and J. Küster-Filipe, "Combining Sequence Diagrams and OCL for Liveness," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 19–38, 2005.
- [13] D. Calegari, "UML 2.0 Interactions with OCL/RT Constraints," InCOPEDECIBA, Master Thesis Report TR07-17, 2007. [Online]. Available: <http://www.fing.edu.uy/inco/pedeciba/biblioteca/tesis/tesis-calegari.pdf>
- [14] V. Pratt, "Modelling Concurrency with Partial Orders," *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33–71, 1986.