REQUIREMENTS-LEVEL SEMANTICS FOR UML STATECHARTS*

Rik Eshuis†
Roel Wieringa
University of Twente, Dept. of Computer Science,
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{eshuis,roelw}@cs.utwente.nl

Abstract We propose a formal real-time semantics for UML statecharts aimed at the requirements level. A requirements-level model assumes perfect technology and has a considerably simpler semantics than an implementation level model. Our semantics is an adaptation of the STATEMATE statechart semantics, with local variables, real time, identifier addressing, point-to-point communication, synchronous communication and dynamic object creation and deletion. We start with an informal comparison of STATEMATE and UML statechart semantics and then give a formalisation of our semantics in terms of labelled transition systems.

Keywords: statecharts, UML, formal semantics

1. INTRODUCTION

Statecharts were introduced by Harel [5] to model the behaviour of activities in the structured analysis approach STATEMATE [8]. They have been adapted in many object-oriented design notations, including the UML [14], but with an informally or undefined semantics that appears to be quite different from the STATEMATE semantics.

In this paper, we define a formal semantics for UML statecharts based upon the STATEMATE semantics for two reasons. First, a formal semantics allows model checking UML models used for requirements specification. This differs from similar research done for STATEMATE [3] in that it works with OO models and it differs from similar work [11] with UML

*Partially supported by Esprit Working Group ASPIRE, contract number 22704.
An abridged version is published in the proceedings of FMOODS 2000 [4]
†Supported by NWO/SION, grant nr. 612-62-02 (DAEMON).

Table 1 Differences between STATEMATE and requirements-level UML

STATEMATE *activity chart + statechart*	*UML class diagram + statechart*
separation of data state/process and control state/process	encapsulation of data state/process and control state/process
channel addressing	identifier addressing
broadcast	point-to-point
instance level model	type-instance distinction

in that it is at the requirements level, not at the implementation level. (A requirements-level model focusses on the design and assumes perfect technology [13], whereas an implementation-level model does not, since it is especially concerned with implementing a design.) Second, such a semantics allows to classify the differences between structured and OO semantics of statecharts in two dimensions: structured versus OO models and requirements level versus implementation level.

Tables 1 summarises the difference between our OO semantics and the structured STATEMATE semantics. Firstly, object-oriented models encapsulate data manipulation, control and data state into objects (as operations, statecharts, attributes). Structured Yourdon-style models separate them into data processes, control processes and data stores, respectively. STATEMATE simulates local variables of statecharts by means of input and output flows accessible to the statechart. This means that in STATEMATE "local" data is really data present in communication channels between activities. STATEMATE models (as all other structured models) have separate activities where UML models only know of data manipulation local to an object. The absence of separate activities and the use of true local variables in the UML considerably simplifies our semantics w.r.t. the STATEMATE semantics. Secondly, to communicate information to a destination, objects in OO models must use the identifier(s) of the destination(s) whereas processes in structured models must use the identifiers of the communication channels. As a consequence, communication in OO models is point-to-point, whereas in structured models, it is broadcast. Finally, object-oriented models use the type-instance distinction, which is absent from structured models. This means that in our semantics, we can deal with dynamic creation and deletion of instances.

Table 2 summarises the differences between our semantics of UML statecharts and the semantics indicated informally in the OMG documentation [14]. The differences follow from the fact that we define a requirements-level semantics whereas the OMG defines an implementa-

Table 2 Differences between requirements and implementation-level semantics

requirements-level semantics	*implementation-level semantics*
perfect technology	imperfect technology
input is event set	• input is queue
system reacts to all events in input	• system reacts to one event in input (the first)
event exists for one step	• event exists arbitrary long
instantaneous communication	• non-instantaneous communication
communication arrives always	communication may get lost, or arrive at wrong destination
one global clock	many local clocks
no clock drift	clocks may drift
action is instantaneous	• action takes time
unlimited concurrency	• limited concurrency, i.e., threads of control with interleaving per thread (one thread per active object)

tion-level semantics. The OMG UML 1.3 semantics makes the assumptions marked by a bullet in table 2. Our semantics, like STATEMATE, makes all of the assumptions in the left-hand column.

Together, tables 1 and 2 factorise the differences between the STATEMATE and UML semantics of statecharts into two groups: the differences between structured and OO models and the difference between requirements-level and implementation-level semantics. On the other hand, Harel and Gery [6] state that the main difference is that UML statecharts use run-to-completion (RTC) whereas STATEMATE statecharts do not. RTC is one possible way to maintain atomicity of transitions at the implementation level. In an RTC semantics, an event can only be processed, if processing of the precious event input has completed (all triggered transitions have been completely taken). Sending an asynchronous message is considered to be completed when the message is sent; calling a synchronous operation is considered completed when the called operation is completed, and this requires maintaining a call stack. STATEMATE has no synchronous communication and hence no call stack. In our opinion, RTC is a minor difference compared to the differences identified by us.

We now briefly sketch some design choices we have made in defining a semantics. The following choices have to be made for any semantics of statecharts, regardless of semantic level or design paradigm (see also Von der Beeck [2]):

- We specify both a clock-synchronous and clock-asynchronous semantics [8]. In the clock-synchronous semantics, the system starts

processing its input only at the tick of the clock. In the clock-asynchronous semantics, the system starts processing its input as soon as it receives it. In the OMG semantics of UML, this issue is ignored.

- In synchronous communication, the caller must wait until the callee has finished processing the communication. In asynchronous communication, the caller continues without waiting for the receiver to finish processing the communication. STATEMATE only uses asynchronous communication. We follow the UML in defining a semantics for both. We show that in our requirements-level semantics, synchronous communication is only possible with a clock-asynchronous semantics.
- Like the UML [14], we allow no compound triggers, no negated trigger events, and only a single entry and single exit action.
- We allow for the case that the action expressions of a transition contain sequence, interleaving and parallel operators. The example action language that we use only contains the sequence operator, but our semantics can deal with the generalised case. Parallel actions that interfere lead nondeterministically to different possible states.
- Updates to variables are made at the end of a step. This way no inconsistent value can be read.
- We assume a given priority preorder on transitions, and do not commit ourselves to any particular one. Possible priority preorder are the STATEMATE one (a higher level transition has priority over a lower level one) and the UML one (a lower level transition has priority over a higher level one).
- In the UML, a distinction is made between active and passive objects. Active objects have computing resources, passive objects have not. At the requirements level, all objects have computing resources, so we only have active objects.

As most other statechart semantics, ours preserves causality (effect should not be contradictory to cause). We allow instantaneous states.

The following UML statechart constructs could be added to our requirements-level semantics without difficulty, but to simplify the exposition we omitted them:

- Deferred events. These can be simulated by regenerating an event as often as it is to be deferred.

- Parametrised events. These can be simulated by sets of parameterless events (one for every combination of parameter values). The resulting state space may be infinite (making model checking impossible).

- A taxonomy for events. This is an abbreviation mechanism that allows one to reduce the number of transitions in a statechart. It does not add expressive power.

- Activities (actions that take time). These can be simulated by instantaneous start and finish events.

- Dynamic choice points. Adding this UML construct would mean that a step is executed in two parts, separated by the dynamic choice point. Dynamic choice points can be simulated by adding an intermediary state.

- Synchronisation states. (These pseudo states are used to mimic a monitoring or semaphore construct.) We regard this to be an implementation-level construct. This construct can be simulated by a join (a transition with multiple source states) with a counter.

- History states. These can be simulated by (re)defining an entry function (see e.g. Damm *et al.* [3]).

Related work. Our LTS semantics is the UML analogon of the LTS semantics for STATEMATE statecharts defined by Damm *et al.* [3]. Due to the differences between UML and STATEMATE discussed above, our semantics is considerably simpler. Our semantics is based upon an earlier requirements-level UML semantics with simple state-transition diagrams given by Wieringa and Broersen [17]. All other approaches known to us are implementation-level semantics.

Latella *et al.* [10] use hierarchical automata to define a formal semantics for one UML statechart. The step semantics they define resembles our semantics of a step, but they do not define other aspects of behaviour (for example, communication).

Lilius and Porres [11] give an operational semantics for the original UML definition [14] by defining an algorithm for the execution of a step in pseudo-code. It is however unclear how to derive a Kripke structure from this algorithm. This is important since the authors have built a model checker based on their semantics. The authors do not define other aspects of behaviour (e.g. creation, deletion, communication).

Reggio *et al.* [16] give a semantics for one UML statechart belonging to an active object by defining a labelled transition system. They stick

as close as possible to the informal standard definition, eliminating inconsistencies, but they nevertheless have to make a number of choices mostly due to the implementation level. They do not define communication.

Harel and Kupferman [7] give an executable object model for a simplified version of a UML like system. They consider simple Mealy machines, whereas we consider statecharts. They focus on communication between different objects and on the possibilities of deadlock in a clock-asynchronous semantics. We, on the other hand, focus on communication at the requirements level where no deadlock is possible, and consider both a clock-synchronous and clock-asynchronous semantics.

In section 2, we summarise the necessary syntax definitions of UML statecharts. In section 3 we generalise the definition of a step so that it applies to structured and object-oriented design approaches and is independent from semantic level. In section 4 we then define the execution semantics of one statechart in terms of a labelled transition system. In section 5 we add creation and deletion action expressions and communication between statecharts. We end with a discussion and conclusions.

2. STATECHART SYNTAX

Most of the definitions in this section are adapted from Damm *et al.* [3] and Pnueli and Shalev [15] with minor changes to deal with final states and completion events in the UML. The paragraph on time expression is new. A statechart SC is a collection of state nodes, hierarchically related, and connected by edges. It is described by the tuple $SC = (Nodes, Edges, Events, Guards, Actions, Var)$, with $Nodes$ the set of state nodes, $Edges$ the set of edges, $Events$ the set of primitive event expressions, $Guards$ the set of guard expressions, $Actions$ the set of action expressions, and Var the set of local variables. We assume a set $Names$ to label the edges.

State nodes. The function $children : Nodes \rightarrow \mathbb{P}Nodes$ defines for each state the set of children of that state (its immediate substates). If $s \in children(s')$, we call s' the *parent* of s. If a state node has no children, it is *basic*. Otherwise, it is a *composite* state node. A composite state node is either an AND or an OR state node. When the system is in an OR state node, it is in one of its children. When the system is in an AND state node, it is in all of its children. The function $type : Nodes \rightarrow \{\text{BASIC,AND,OR}\}$ assigns to every state node its type.

Function *children* defines a partial ordering on the state nodes:

$$s \leq s \qquad \text{Reflexivity}$$

$$s \leq s' \quad \text{if } s \in \mathit{children}(s')$$
$$s \leq s' \wedge s' \leq s'' \Rightarrow s \leq s'' \qquad \text{Transitivity}$$

If $s \leq s'$ we say that s' is the *ancestor* of s. If either $s \leq s'$ or $s' \leq s$, we say s and s' are *ancestrally related*.

There are some special state nodes. We assume a unique root state node, denoted *root*. We require *root* to be an OR state node. Another special state node is the final state node. If state node s is an OR state and one if its children is a final state node, that state node is denoted $\mathbf{final}_s$. A final state node must be basic, and no edges emanate from it. Obviously, $\mathbf{final}_s \leq \mathbf{final}_s$ and $\mathbf{final}_s \leq s$.

The partial function $\mathit{default} : \mathit{Nodes} \rightarrow \mathit{Nodes}$ identifies for each OR state node s one of its children as the default state node, that is entered when s is entered.

For a set of state nodes $X \subseteq \mathit{Nodes}$, the least common ancestor $\mathit{lca}(X)$ is the state $l \in \mathit{Nodes}$ such that:

$$\forall x \in X \cdot x \leq l$$
$$\forall y \in \mathit{Nodes} \cdot (\forall x \in X \cdot x \leq y) \Rightarrow l \leq y$$

Two state nodes s, s' are *orthogonal*, written $s \perp s'$, if s and s' are not ancestrally related and their *lca* is an AND state.

A set of state nodes $X \subseteq \mathit{Nodes}$ is *consistent* if, for every every pair $x, y \in X$, either x and y are ancestrally related or x and y are orthogonal. A *configuration* is a maximal consistent set of state nodes. Configurations are valid 'states' of the statechart.

In order to deal with UML completion events, we define predicate $\mathit{stopped}(C, s)$ that is true if in configuration C (composite) state s is terminated:

$$\mathit{stopped}(C, s) = \begin{cases} \mathbf{false}, & \text{if } \mathit{type}(s) = \text{BASIC} \\ \mathbf{final}_s \in C, & \text{if } \mathit{type}(s) = \text{OR} \\ \forall x \in \mathit{children}(s) \cdot \mathit{stopped}(C, x), & \text{if } \mathit{type}(s) = \text{AND} \end{cases}$$

Edges. An edge is a relation from one set of state nodes (source) to another (target). An edge has a unique name. It is labelled with an event expression, a guard expression and an action expression. (Informally, when the event specified by the event expression occurs, and the guard expression evaluates to true, the action expression is evaluated.) Formally, $\mathit{Edges} \subseteq \mathit{Names} \times \mathbb{P}\mathit{Nodes} \times (\mathit{Events} \cup \emptyset) \times \mathit{Guards} \times \mathit{Actions} \times \mathbb{P}\mathit{Nodes}$. For every $t = (n, X, e, g, as, Y)$, let $\mathit{name}(t) = n$, $\mathit{source}(t) = X$, $\mathit{event}(t) = e$, $\mathit{guard}(t) = g$, $\mathit{action}(t) = as$ and $\mathit{target}(t) = Y$.

We require that $source(t)$ and $target(t)$ are non-empty consistent sets of state nodes.

For a given transition t whose source is composite state s, a special event that may be used to label t is $\surd_s$. This we call the completion event. It occurs if and only if in a given configuration C, $stopped(C, s)$ becomes true.

The *scope* of a transition t is the most nested OR state that contains both $source(t)$ and $target(t)$. It is the most nested state of the statechart that is not exited or entered when the transition is taken.

Two transitions are consistent if they are equal or their scopes are orthogonal. A set of transitions T is called a *consistent* set if t_1 and t_2 are consistent, for every $t_1, t_2 \in T$. Denote by $consTrans(T)$ the set of all transitions that are consistent with every $t \in T$. Note that if T is consistent, then $T \subseteq consTrans(T)$.

Next, we assume a priority ordering $\preceq$ on the transitions. We need an order to choose some transitions if we have a non-consistent set of transitions. We assume $\preceq$ to be a total pre-order. We write $t \preceq t'$ to denote that t has higher or the same priority as t'.

Syntax of action expressions. The set of basic action expressions for a given statechart SC, denoted $Actions(SC)$, is defined as follows.

`v:=exp`	where $v \in Var(SC)$
`a;a'`	where a, $a' \in Actions(SC)$
`refid:=create(Class)`	where *refid* $\in Var(SC)$
`delete(id)`	
`send id.signal`	where *signal* is a signal reception of object *id*
`id.oper`	where *oper* is an operation of object *id*

This syntax may easily be extended to incorporate more advanced features. In section 5 we will explain the meaning of *Class* and *id*.

Time expressions. In the UML there are two kinds of time expressions: absolute and relative. An absolute time expression references the current time. It is specified by the UML when(*cond*) construct (e.g. when(time=12:00:00h)). A relative time expression references the time relative to some occasion. It is specified by the UML after(*texp*) construct (e.g. after(10s), which is true 10 seconds after the source state has been entered), where *texp* is of type $\mathbb{N}$.

Below, we will define a translation of when and after into basic guard expressions (i.e. not containing $\wedge, \vee, \neg$), that we call basic clock constraints. We assume that each basic clock constraint references a clock c of type real, and that the form of the constraint is $c \leq n$ or $c \geq n$ where $n \in \mathbb{N}$.

We translate an absolute time expression when(*cond*) into a guard expression that references a global clock (the current time) by treating *cond* as guard expression. We introduce special clock variable *ct* to represent the current time. At the beginning of the execution of the statechart, this clock is initialised with the current time.

We translate a relative time expression after(*texp*) as follows into a guard expression that references a local clock. We regard after as a special kind of event label that can only label transitions that have a single source. Let $x = source(t)$. We introduce a special clock variable tmr_x that denotes a local clock (a timer), that will be reset every time state node x is entered. The event after(*texp*) is replaced by guard expression $[tmr_x \geq texp]$.

Note that a basic clock constraint of the form $tmr_x \leq n$, where tmr_x is a timer and $n \in \mathbb{N}$, does not make sense. If state node x is entered, tmr_x is set to zero; hence, the basic clock constraint will be always true then.

In STATEMATE, scheduled actions may be used. A scheduled action, in STATEMATE syntax denoted sc(a, *texp*), means that action a must be done after delay *texp*. This can be specified in the UML using the after construct: after(*texp*)/a.

3. GENERIC STATECHART STEP SEMANTICS

In this section and the next one we give a requirements-level semantics for one statechart. In this section we present the definition of a step. It is generic and independent from whether we follow a structured or object-oriented design approach. It is simpler than the STATEMATE definition of a step [8] [3] because there are neither negative nor compound events in the UML.

We define how to compute a step, given a configuration C and a set of input events I. This step must obey several constraints, such as maximality and consistency and priority. After Pnueli and Shalev [15], we define a step both declaratively and operationally. Note however that we do not define a superstep in this way, as they do, and that we take into account priorities, whereas they do not.

First, we define all enabled transitions $En(C, I)$:

$$En(C, I) = \{t \mid source(t) \subseteq C \wedge event(t) \in I \wedge guard_\sigma(t)\}$$

where $guard_\sigma(t)$ represents the evaluation of $guard(t)$ in valuation σ. (See section 4 for the definition of a valuation.)

We construct a step in an incremental way, by adding transitions that are consistent. To aid in this construction, we define function *addToStep*

Figure 1 $nextstep(C, I)$: algorithm to compute steps

```
let T = ∅;
while T ⊂ addToStep(T) do
choice t ∈ high(addToStep(T) −
T);
let T = T ∪ {t}
endwhile
return T
```

as follows. Suppose we have decided to take a set of transitions T. The function $addToStep(T)$ gives us the transitions that are enabled and are consistent with T. The function $addToStep(T)$ is defined by:

$$addToStep(T) = En(C, I) \cap consTrans(T)$$

A step T^* must satisfy the following constraints:

1 *Enabledness* All transitions of the step must be enabled.

2 *Consistency* All transitions taken in T^* are consistent with each other, i.e., each of them could be added.

3 *Maximality* The step is maximal, i.e., every transition that could be added is already part of the step.

4 *Priority* If an enabled transition is not in the step, then there must a transition in the step that is inconsistent and has higher or the same priority.

We formalise these constraints as follows:

$$T^* \subseteq En(C, I) \qquad \text{Enabledness}$$
$$T^* \subseteq consTrans(T^*) \qquad \text{Consistency}$$
$$addToStep(T^*) \subseteq T^* \qquad \text{Maximality}$$
$$\forall t \in En(C, I) - T^* \; \exists t' \in T^* \cdot t \notin consTrans(\{t'\}) \wedge t' \preceq t \qquad \text{Priority}$$

Note that the first three constraints are equivalent to the fixpoint equation $T^* = addToStep(T^*)$.

Next, we give a nondeterministic algorithm $nextstep(C, I)$ to compute the maximal, nonconflicting step given a configuration C and a set of input events I (see Fig. 1). In the definition of the algorithm, we use a set $high(T)$. This set $high(T) \subseteq T$ contains the transitions of T with the highest priority:

$$high(T) = \{t \in T \mid \forall t' \in T \cdot t \preceq t'\}$$

Now that we have defined a step in two different ways, we show the consistency of both approaches. We call a step T^* constructible if and only if T^* can be constructed via procedure $nextstep(C, I)$.

Proposition 1. *T^* satisfies the constraints 1-4 iff T^* is constructible.*

Proof. $\Rightarrow$. Let T^* be a step satisfying constraints 1,2,3, and 4. We show that T^* is constructible, by showing $T^* \cap high(addToStep(T) - T) \neq \emptyset$ for $T \subset T^*$. The claim follows from this fact, and the monotonicity of *addToStep*.

First, observe that $T^* \cap (addToStep(T) - T) \neq \emptyset$, because *addToStep* is monotone. Also, $high(T) \neq \emptyset$ if $T \neq \emptyset$.

Take arbitrary transition $t \in high(addToStep(T) - T)$. If $t \in T^*$ the claim is true. Suppose $t \notin T^*$. From constraint 4 follows, there exists a transition $t' \in T^*$ such that $t' \preceq t$. Because *addToStep* is monotone decreasing, $t' \in addToStep(T)$. Also, $t' \notin T$ (otherwise t would not be enabled). So $t' \in high(addToStep(T) - T)$ (because $t' \preceq t$ and $t \in high(addToStep(T) - T)$).

$\Leftarrow$. Constraints 1,2 and 3 follow from standard fixpoint theorems. We only prove constraint 4. Let $T^* = \{t_0 \ldots t_n\}$ be a step, with the transitions ordered by time of addition. Take an arbitrary t from $En(C, I) - T^*$. We now show there must exist a $t' \in T^*$ such that t' is inconsistent with t and $t' \preceq t$. Clearly, $t \notin T^*$ because there is another transition $t' \in T^*$ that is inconsistent with t. Let $T^{*'} = \{m_0, \ldots, m_i\} \subseteq T^*$ be the set to which t' was added. We know that $t \in addToStep(T^{*'}) - T^{*'}$ (because *addToStep* is monotone). Now $t' \in high(addToStep(T^{*'}) - T^{*'}$. So $t' \preceq t$. □

4. STATECHART LTS SEMANTICS

We now present two different ways of executing a step. In the *clock-synchronous* semantics, a step is executed when the clock ticks. In the *clock-asynchronous* semantics, a step is executed immediately when new events arrive. Then the system becomes unstable and reacts infinitely fast to become stable again. These two semantics are borrowed from the STATEMATE semantics. The major differences with STATEMATE semantics are the different treatment of termination in the UML and the absence of a separate activity model in the UML. Besides, our definition of real time is different from the STATEMATE definition.

4.1. CLOCKED LABELLED KRIPKE STRUCTURE

We assume a typed set of variables Var and a typed data domain $\mathcal{D}$. A *valuation function* $\sigma : Var \rightarrow \mathcal{D}$ assigns to every variable a value. The set of valuation functions defined on Var is denoted $\Sigma(Var)$. Every state of the clocked labelled Kripke structure is a valuation σ of the variables Var. From now on, we will use the term 'valuation' instead of 'state of a clocked labelled Kripke structure' to avoid confusion with the state nodes of statecharts. Valuations are connected by transitions labelled with actions out of set Act.

A clocked labelled Kripke structure (CLKS) is a tuple $(Var, Act, \rightarrow, ci, \sigma_0)$ with:

- Var the set of variables,
- Act the set of actions,
- $\rightarrow \subseteq \Sigma(Var) \times \mathbb{P}Act \times \Sigma(Var)$ the transition relation,
- $ci : \sigma \rightarrow CC$ a function that assigns a conjunction of basic clock constraints to every valuation, the clock invariant,
- $\sigma_0 \in \Sigma(Var)$ the initial valuation.

Set CC denotes all possible conjunctions of all basic clock constraints. Instead of $(\sigma, A, \sigma') \in \rightarrow$, we write $\sigma \xrightarrow{A} \sigma'$. The clock invariant is a conjunction of basic clock constraints of the form $c \leq n$. Given configuration $\sigma(cfg)$, for every transition t with $source(t) \subseteq \sigma(cfg)$, that has a guard expression containing a clock constraint of the form $c \geq n$, the clock invariant ci_σ contains a basic clock constraint $c \leq n$. Below, we will see that the clock-invariant helps in raising a temporal event (see the definition of $\rightarrow_{time}$). This is necessary for the clock-asynchronous semantics. It is not necessary for the clock-synchronous semantics, since at every tick of the clock input is processed, but it helps in clarifying the clock-synchronous semantics.

For readers familiar with timed automata [1] or clocked transition systems [12], note that we do not have defined clock constraints on transitions of the CLKS. Instead, clock constraints are implicitly present as guard expressions in the statechart! Equivalently, clock constraint could be defined on transitions of the CLKS, corresponding to taking a step, by taking the conjunction of the clock constraints of each transition in the statechart performed in the step. But this would considerably complicate the semantics, as some steps would no longer be valid anymore. Not all steps would be valid steps, because some transition in the CLKS

might violate a clock constraint. Our approach is more simple, since such a violating transition simply cannot occur in the step. The set of clocks to be reset is implicit; see the definition of $\rightarrow_{copy}$ below.

When interpreting a statechart in an LTS, the set of variables *Var* comprises the variables of a statechart *SC* (denoted $Var(SC)$), and a copy of these variables, denoted by priming the original variables. Updates are made to the primed variables, to ensure that actions can only read the values variables had at the beginning of a step.

Special elements of *Var* are the variables *cfg* and *inp* denoting resp. the current configuration of the statechart and the input of the statechart, represented as a set of events. Next, we have a clock variable *ct* that denotes the current time, and a set of clock variables *tmr* that contains all timers tmr_x, where x is the source of a transition that has a guard expression referencing local clock tmr_x (see section 2).

We only consider Kripke structures that satisfy the constraint that all relevant completion events are in the input:

$$\{\surd_s \mid stopped(\sigma(cfg), s) \wedge s \in \sigma(cfg)\} \subseteq \sigma(inp)$$

We let set *Act* be the union of the sets *Events*, *Actions*, and *Guards*. We assume a function $act : Edges \rightarrow Act$ that, given a set of edges, returns the set of actions that are in the set. This function will be used to label the transitions in the CLKS with actions.

Next, we sketch briefly how action expressions are evaluated. If a is an action expression and σ a valuation, then $[\![a]\!]\sigma$ is the valuation resulting from the evaluation of expression a. Updates are made to the primed versions of variables only. For example:

$$\begin{array}{lcl} [\![x := exp]\!]\sigma & = & \sigma[x'/[\![exp]\!]\sigma] \\ [\![action; actionsequence]\!]\sigma & = & [\![action]\!]\sigma; [\![actionsequence]\!]\sigma \end{array}$$

The meaning of the x/val construct on valuations is defined as follows:

$$\sigma[x/val](y) = \begin{cases} \sigma(y), & \text{if } y \neq x \\ val, & \text{if } y = x \end{cases}$$

Finally, for the initial valuation σ_0 it must hold that:

$$\begin{array}{lcl} & & \sigma_0(cfg) = dcomp(\{root\}) \\ & & \sigma_0(input) = \emptyset \\ \forall t \in tmr & \cdot & \sigma_0(t) = 0 \end{array}$$

and *ct* and the local variables must be given an appropriate value. Function *dcomp* computes the default completion of a set of state nodes (see e.g. [15], [3] for a definition).

4.2. BASIC CLOCK-SYNCHRONOUS EXECUTION

The $\rightarrow$ relation of the Kripke structure is the union of three relations $\rightarrow_{time}$, $\rightarrow_{change}$, and $\rightarrow_{step}$, defined below. Not every sequence of transitions out of these three sets satisfies the clock-synchronous semantics. Sequences in the clock-synchronous semantics are cycles, where a cycle describes the execution of the CLKS during one time-unit (from one clock tick to another). A cycle is a non-empty sequence of time steps (which represent the elapsing of time), possibly interleaved with change steps (which represent changes in the environment), followed by the taking of a step at the tick of the clock:

$$\xrightarrow{A}_{cycle} = \xrightarrow{\delta_0}_{time}; \rightarrow_{change}; \xrightarrow{\delta_1}_{time}; \rightarrow_{change} \ldots \xrightarrow{\delta_j}_{time}; \xrightarrow{A}_{step}$$

where $\Sigma_{i=0}^{j}\delta_i = 1$. Relation $\rightarrow_{time}$ corresponds to the elapsing of time. Relations $\rightarrow_{change}$ represents a change in the environment. There are three possible changes: either an external event occurs ($\rightarrow_{event}$), a value change occurs ($\rightarrow_{value}$), or a temporal event occurs ($\rightarrow_{time-out}$).

$$\begin{array}{lcl}
\sigma \xrightarrow{\delta}_{time} \sigma' & \Leftrightarrow & \sigma' = \sigma[ct/(\sigma(ct)+\delta), \\
 & & \qquad \&_{tmr_x | tmr_x \in tmr} tmr_x/(\sigma(tmr_x)+\delta)] \\
 & \wedge & \forall \varepsilon \in [0,\delta] \cdot ci_{\sigma''} \quad \text{where} \\
 & & \sigma'' = \sigma[\&_{tmr_x | tmr_x \in tmr} tmr_x/(\sigma(tmr_x)+\varepsilon)] \\
\rightarrow_{change} & = & \rightarrow_{event} \cup \rightarrow_{value} \cup \rightarrow_{time-out} \\
\sigma \rightarrow_{event} \sigma' & \Leftrightarrow & \exists E \subseteq Events \cdot \ E \neq \emptyset \\
 & & \qquad \wedge \ \sigma' = \sigma[inp/(\sigma(inp) \cup E)] \\
\sigma \rightarrow_{value} \sigma' & \Leftrightarrow & \sigma \neq \sigma' \\
 & \wedge & \sigma(cfg) = \sigma'(cfg) \\
 & \wedge & \sigma(inp) = \sigma'(inp) \\
 & \wedge & \forall tmr_x \in tmr \cdot \sigma(tmr_x) = \sigma'(tmr_x) \\
\sigma \rightarrow_{time-out} \sigma' & \Leftrightarrow & \sigma = \sigma' \\
 & \wedge & \nexists \sigma'' \cdot \sigma \rightarrow_{time} \sigma''
\end{array}$$

Relation $\xrightarrow{\delta}_{time}$ represents the increase of the global clock ct and all timers tmr_x by δ, provided that during this period all clock invariants hold. The ampersand symbol denotes that all updates to all timers are made simultaneously:

$$\sigma[\&_{id \in ID} x_{id}/val_{id}] = \sigma[x_{firstid}/val_{firstid} \ldots x_{lastid}/val_{lastid}]$$

where $ID = \{firstid \ldots lastid\}$. Relation $\rightarrow_{event}$ extends the current input set nondeterministically with a set E without changing the system state. Relation $\rightarrow_{value}$ indicates a value change of one or more local variables. Relation $\rightarrow_{time-out}$ represents that a temporal event (time-out) occurs iff a time step is not possible. Next, we define the $\rightarrow_{step}$ relation, corresponding to the execution of a step.

$$
\begin{array}{lcl}
\xrightarrow{A}_{step} & = & \xrightarrow{A}_{nextstep}; \rightarrow_{copy} \\
\sigma \xrightarrow{A}_{nextstep} \sigma' & \Leftrightarrow & A = act(T^*) \\
& \wedge & \sigma' = \sigma''[cfg/nextconfig(\sigma(cfg), T^*), inp/\emptyset] \\
& & \text{where } isStep(T^*) \text{ and } \sigma'' = [\![actions(T^*)]\!]\sigma. \\
\sigma \rightarrow_{copy} \sigma' & \Leftrightarrow & \sigma' = \sigma[\ \&_{v \in Var(SC)} v/\sigma(v') \\
& & \qquad \&_{tmr_x | tmr_x \in tmr \wedge x \in \sigma(cfg)} tmr_x/0]
\end{array}
$$

Relation $\xrightarrow{A}_{nextstep}$ relates valuation σ with σ' iff given input events $\sigma(inp)$ in configuration $\sigma(cfg)$ step T^*, satisfying $isStep(T^*)$ is taken, and the actions of T^* are evaluated, such that valuation σ'' is reached. Next, the configuration and inputs are updated, the input is removed, and valuation σ' is reached. (See e.g. [3] for a precise definition of function *nextconfig*.) Predicate $isStep(T^*)$ is true iff step T^* satisfies the consistency, maximality and priority constraints, given configuration $\sigma(cfg)$ and set of inputs $\sigma(inp)$. Whether completion transitions have priority over non-completion transitions, depends on the definition of the priority relation $\preceq$. Function $actions(st)$, given a set of transitions, returns a sequence of their actions, respecting the order of the action within each transition, but nondeterministically choosing an order between the transitions themselves:

$$actions(T) = action(t_1); \ldots; action(t_n) \Leftrightarrow \{t_1, \ldots, t_n\} = T$$

Relation $\rightarrow_{copy}$ represents that all unprimed variables are updated with the values of their primed counter parts and all relevant timers are reset.

Finally, we present two algorithms in natural language to execute a step and a cycle. These algorithms serve as an informal explanation of the semantics defined above. In Fig. 2 we present an algorithm to execute a step. It corresponds to the definition of $\xrightarrow{A}_{step}$. In Fig. 3 we present an algorithm to execute a sequence of cycles in the clock-synchronous semantics. The body of the while loop of this algorithm corresponds to the definition of $\xrightarrow{A}_{cycle}$.

Figure 2 Algorithm to execute a step

1 Compute $nextstep(C, I)$ using the algorithm in Fig. 1
2 For every transition in the step execute the actions (only updating primed variables)
3 Compute the next configuration
4 Update the current configuration with the next configuration
5 Empty the input set
6 Update the variables with the values of their primed counterparts
7 Reset all relevant timers

Figure 3 Execution algorithm for the clock-synchronous semantics

While true do
- Receive changes until clock ticks
- Execute a step (see Fig. 2)

4.3. BASIC CLOCK-ASYNCHRONOUS EXECUTION

The $\rightarrow$ relation of the CLKS is a union of the $\rightarrow_{time}$, $\rightarrow_{change}$ and $\rightarrow_{superstep}$ relations. Like in the clock synchronous semantics, an execution sequence is a sequence of cycles. Now, however, a cycle describes the execution of the CLKS from the end of one reaction until the end of the execution of the next one. A cycle is a sequence of time steps, followed by a change step, immediately followed by the reaction of the system, a superstep. A superstep is a chain of steps. The fact that a chain of steps is taken, rather than a single step, is due to the perfect technology assumption (reactions happen infinitely fast) and the assumption of the clock-asynchronous semantics that a change in the environment, including one caused by the system itself, is immediately reacted upon. This implies that all completion events are immediately reacted upon; hence in all valuations the configuration of the statechart is stable (does not contain a final state node). Another effect of this definition is that a superstep may be infinite. In that case, we say the superstep diverges.

$$\xrightarrow{A}_{cycle} \quad = \quad \rightarrow_{time}; \rightarrow_{change}; \xrightarrow{A}_{superstep}$$
$$\sigma \xrightarrow{A}_{superstep} \sigma' \quad = \quad \sigma \xrightarrow{A_1}_{step} \ldots \xrightarrow{A_n}_{step} \sigma'$$

where $A = A_1 \cup \ldots \cup A_n$ and $stable(\sigma'(cfg), \sigma'(inp))$ must hold:

Figure 4 Execution algorithm for the clock-asynchronous semantics

While true do

- Receive change (external event, value change, temporal event)
- Execute a step (see Fig. 2)
- Repeat: if the resulting state is unstable, then execute a step
  Until the state is stable

$$stable(C, I) \Leftrightarrow (\forall s \in C \cdot stopped(C, s)) \wedge I = \emptyset \wedge En(C, I) = \emptyset$$

The superstep stops if the current configuration is stable and there are no more inputs and no more enabled transitions: the state is stable. All intermediate states must be unstable. The other definitions are as before. Note that we do not require the initial valuation of the superstep to be a stable state, since this valuation is stable because the initial valuation is stable and every transition out of set $\rightarrow_{time}$, $\rightarrow_{change}$, and $\rightarrow_{superstep}$ ends up in a stable state.

In Fig. 4 we present an algorithm to execute a sequence of cycles in the clock-asynchronous semantics. The body of the while loop of this algorithm corresponds to the definition of $\xrightarrow{A}_{cycle}$.

5. COMMUNICATING STATECHARTS

We now add creation and deletion of objects, absent from STATEMATE semantics, and synchronous and asynchronous communication. STATEMATE only uses asynchronous communication. Our semantics leans heavily on the use of object identifiers.

Assume a set of Classes *Class* and a set of object identifiers *OID*. Every class c has a statechart definition associated, whose root is identified as $c.root$. Now that we have multiple objects instead of one, we index the special variables *cfg*, *inp* and *tmr* with the object id's. So $cfg[id]$ denotes the configuration *cfg* of object *id*. We assume that all variables used by objects are unique, including clock variables. (Variables can be made unique by simply putting the object id in front.) We denote the local variables of an object *id* with $Var(id)$. As before, every variable $v \in Var(id)$ has a primed version, but now, in addition, every set of input events $inp[id]$ has also its primed version $inp'[id]$, because we define communication between statecharts.

5.1. CREATION AND DELETION

The meaning of action expression `refid:=create (Class)` is that an object of class *Class* is created with a new identity whose value is assigned to variable *refid*. Action expression `destroy(id)` means that object *id* is destroyed.

In order to give a formal semantics for creation and deletion we make use of two predicates. Predicate $Exists(id)$ is true iff an object with identifier *id* exists. Predicate $Used(id)$ is true iff there exists or existed an object with identifier *id*. The obvious constraint holds that $Exists(id) \Rightarrow Used(id)$.

$$\begin{array}{lcll} [\![refid := create(Class)]\!]\sigma & = & \sigma[cfg[id]/dcomp(\{Class.root\}), & \\ & & \quad inp[id]/\emptyset, & \\ & & \quad refid/id, & \\ & & \quad Exists(id)/true, & \\ & & \quad Used(id)/true] & \text{if } \neg Used_\sigma(id) \\ [\![destroy(id)]\!]\sigma & = & \sigma[Exists(id)/false] & \text{if } Exists_\sigma(id) \end{array}$$

A new object *id* can be created only if *id* is not used before. If object *id* can be created, then it is initialised by setting its configuration and set of input events, and updating predicates *Used* and *Exists*.

Before we define the clock-synchronous and clock-asynchronous semantics, we state the constraint that for every existing object, all relevant completion events should be in the input:

$$\begin{array}{l} \forall id \mid Exists_\sigma(id) \cdot \\ \quad \{\surd_s \mid stopped(\sigma(cfg[id]), s) \wedge s \in \sigma(cfg[id])\} \subseteq \sigma(inp[id]) \end{array}$$

The initial valuation of the Kripke structure now must satisfy

$$\begin{array}{lcl} \forall id \mid Exists_\sigma(id) & \cdot & \sigma_0(cfg) = dcomp(\{id.root\}) \\ & \wedge & \sigma_0(input) = \emptyset \\ & \wedge & \forall t \in tmr[id] \cdot \sigma_0(t) = 0 \end{array}$$

We assume that all local variables of the existing objects are given an appropriate value.

5.2. COMMUNICATION

Clock-synchronous model. In the clock-synchronous model, we only allow asynchronous communication (the **send** action). The reason is that in synchronous communication, the caller must wait until the

callee returns. In the clock-synchronous model, the callee is performing its own step when it receives the call and can respond to the call only in the next step at the next tick of the clock. Since in our requirements-level semantics, a transition must be taken in zero time, the caller cannot wait for the callee to do its work. We therefore have no synchronous communication in a clock-synchronous semantics.

As in the clock-synchronous model of the previous section, the $\rightarrow$ relation of the CLKS is the union of three relation $\rightarrow_{time}$, $\rightarrow_{change}$ and $\rightarrow_{step}$. Again, every sequence of the CLKS must be a sequence of cycles, where a cycle is as defined in the previous clock-synchronous model. The three relations have to be redefined to take into account the fact that we now have a set of objects, rather than one object. Relations $\rightarrow_{time}$ and $\rightarrow_{change}$ now become:

$$
\begin{array}{lcl}
\sigma \xrightarrow{\delta}_{time} \sigma' & \Leftrightarrow & \sigma' = \sigma[\&_{id|Exists_\sigma(id)} \\
& & \qquad \&_{tmr_x|tmr_x \in tmr[id]} tmr_x/(\sigma(tmr_x)+\delta), \\
& & \qquad ct/(\sigma(ct)+\delta)] \\
& \wedge & \forall \varepsilon \in [0,\delta] \cdot (\forall id \in OID | Exists_\sigma(id) \cdot id.ci_{\sigma''}) \\
& & \text{where } \sigma'' = \sigma[\&_{id|Exists_\sigma(id)} \\
& & \qquad \&_{tmr_x|tmr_x \in tmr} tmr_x/(\sigma(tmr_x)+\varepsilon)] \\
\rightarrow_{change} & = & \rightarrow_{event} \cup \rightarrow_{value} \cup \rightarrow_{time-out} \\
\sigma \rightarrow_{event} \sigma & \Leftrightarrow & \exists ID \subseteq OID | ID \neq \emptyset \cdot \\
& & \quad (\forall id \in ID | Exists_\sigma(id) \cdot \\
& & \quad\quad (\exists E \subseteq Events | E \neq \emptyset \cdot \\
& & \quad\quad\quad \sigma' = \sigma[inp[id]/(\sigma(inp[id]) \cup E)] \\
& & \quad\quad) \\
& & \quad) \\
\sigma \rightarrow_{value} \sigma' & \Leftrightarrow & \sigma \neq \sigma' \\
& \wedge & \forall id \in OID | Exists_\sigma(id) \cdot \\
& & \quad \wedge \quad \sigma(cfg[id]) = \sigma'(cfg[id]) \\
& & \quad \wedge \quad \sigma(inp[id]) = \sigma'(inp[id]) \\
& & \quad \wedge \quad \forall t \in tmr[id] \cdot \sigma(t) = \sigma'(t) \\
& \wedge & \sigma(ct) = \sigma'(ct) \\
\sigma \rightarrow_{time-out} \sigma' & \Leftrightarrow & \sigma = \sigma' \\
& \wedge & \nexists \sigma'' \cdot \sigma \rightarrow_{time} \sigma''
\end{array}
$$

Next, we redefine the $\rightarrow_{step}$ relation:

$$
\xrightarrow{A}_{step} \quad = \quad \xrightarrow{A}_{nextstep}; \rightarrow_{copy}
$$

$$
\begin{array}{lcl}
\sigma \xrightarrow{A}_{nextstep} \sigma' & \Leftrightarrow & T^* = \bigcup_{id|Exists_\sigma(id)} T^*_{id} \\
& \wedge & A = act(T^*) \\
& \wedge & \sigma' = \sigma''[\ \&_{id|Exists_\sigma(id)} \\
& & \qquad cfg[id]/nextconfig(\sigma(cfg[id]), T^*_{id})] \\
& & \text{where for every } id,\ Exists_\sigma(id) \Rightarrow isStep(T^*_{id}), \\
& & \text{and } \sigma'' = [\![actions(T^*)]\!]\sigma \\
\sigma \rightarrow_{copy} \sigma' & \Leftrightarrow & \sigma' = \sigma[\ \&_{id|Exists_\sigma(id)} \\
& & \qquad \&_{v|v\in Var(id)} v/\sigma(v'), \\
& & \qquad inp[id]/\sigma(inp'[id]) \\
& & \qquad inp'[id]/\emptyset] \\
& & \qquad \&_{tmr_x|tmr_x\in tmr[id]\wedge x\in\sigma(cfg[id])} tmr_x/0]
\end{array}
$$

The definition of $\rightarrow_{nextstep}$ reflects that all existing objects dispatch their events at the same time. Step T^* is defined as the union of the individual steps of the objects. The remainder of the definition is a generalisation of the definition of $\rightarrow_{nextstep}$ in the previous section. The $\rightarrow_{copy}$ relation is a straightforward generalisation of the counterpart with the same name in the preceding models. Note that the set of input events after a step may be filled with events generated by **send** actions.

Next, we add the semantics of a non-blocking communication by specifying the semantics of a **send** action expression.

$$[\![\textbf{send}\ id.signal]\!]\sigma = \sigma[inp'[id]/(\sigma(inp'[id]) \cup \{signal\})]$$

We end this paragraph by giving two algorithms to execute a step $\xrightarrow{A}_{step}$ (Fig. 5) and a sequence of cycles $\rightarrow_{cycle}$ (Fig. 6) in the multi-object clock-synchronous semantics.

Clock-asynchronous model. In the clock-asynchronous model, we again define $\rightarrow$ and a cycle as in the previous clock-asynchronous model, reusing for $\rightarrow_{time}$ and $\rightarrow_{change}$ the definitions of the previous paragraph. We now define the relation $\rightarrow_{superstep}$.

First, observe that in the clock-asynchronous model both asynchronous and synchronous communication is allowed. Synchronous communication is allowed because each object instantly reacts to the events it receives and hence always is ready to synchronise with another object. We assume that only a single object is active during a single step. We can justify this by our other assumption that single steps and supersteps do not take time.

Figure 5 Algorithm to execute a step in the multi-object clock-synchronous semantics

1. For every existing object, compute its step $nextstep(C, I)$ using the algorithm in Fig. 1
2. For every existing object, execute for every transition in its step the actions (only updating primed variables)
3. For every existing object
   - Compute its next configuration
   - Update its current configuration with its next configuration
4. For every existing object
   - Update the variables with the values of their primed counterparts (including the input set)
   - Empty its primed input set (that contains the events generated in the step itself)
   - Reset all relevant timers

Figure 6 Execution algorithm for the multi-object clock-synchronous semantics

While true do

- Receive changes for existing objects until clock ticks
- Execute a step for all existing objects (see Fig. 5)

A step, executed by one object id, is defined as follows. The definition formalises RTC semantics.

$$
\begin{array}{lcl}
\sigma \xrightarrow{A,id}_{step} \sigma' & = & \sigma \xrightarrow{A,id}_{nextstep}; \xrightarrow{id}_{copy} \\
\sigma \xrightarrow{A,id}_{nextstep} \sigma' & \Leftrightarrow & A = act(T^*) \\
 & \wedge & \sigma' = \sigma''[cfg[id]/nextconfig(\sigma(cfg[id]), T^*)] \\
 & & \text{where } isStep(T^*) \text{ and } \sigma'' = [\![actions(T^*)]\!]\sigma \\
\sigma \xrightarrow{id}_{copy} \sigma' & \Leftrightarrow & \sigma' = \sigma[\ \&_{v \in Var(id)}\ v/\sigma(v'), \\
 & & \quad \&_{tmr_x | tmr_x \in tmr[id] \wedge x \in \sigma(cfg[id])} tmr_x/0, \\
 & & \quad \&_{i | Exists_\sigma(i)} \\
 & & \quad\quad inp[i]/\sigma(inp'[i]), \\
 & & \quad\quad inp'[i]/\emptyset]
\end{array}
$$

The above definition is a straightforward adaptation of the previous definitions of $\rightarrow_{step}$, $\rightarrow_{nextstep}$ and $\rightarrow_{copy}$. The change w.r.t. the clock-

synchronous model of this section is that we now define a step for one object only. A superstep S of a finite set of objects O will be a nondeterministic interleaving of supersteps of the each object in O. The next definition says that in each single step of S we nondeterministically choose a step of some object in O that is unstable.

$$\begin{array}{lcl}
\sigma \xrightarrow{A}_{singlestep} \sigma' & \Leftrightarrow & \exists id \in OID \mid Exists_\sigma(id) \cdot \\
& \wedge & \sigma \xrightarrow{A,id}_{step} \sigma' \\
& \wedge & \neg stable(\sigma(cfg[id]), \sigma(inp[id])) \\
\sigma \xrightarrow{A}_{superstep} \sigma' & \Leftrightarrow & \sigma \xrightarrow{A_1}_{singlestep} \ldots \xrightarrow{A_n}_{singlestep} \sigma' \\
& \wedge & \forall id \in OID \mid Exists_{\sigma'}(id) \cdot \\
& & \qquad stable(\sigma'(cfg[id]), \sigma'(inp[id]))
\end{array}$$

where $A = A_1 \cup \ldots \cup A_n$.

What remains is to define the meaning of synchronous calls as part of the execution of $actions(T^*)$. If an object calls operation *oper* of object *id*, then first of all *id* should exist, and second, *id* should process the call. Since *id* itself may have nonempty input, this old input should be remembered and placed back after *id* has finished executing. Together with $\rightarrow_{step}$, this formalises RTC semantics, because it says that a call action is executed only when the called action is executed.

$$\begin{array}{lll}
[\![id.oper]\!]\sigma = \sigma' & \text{if} & Exists_\sigma(id) \\
& & \wedge\ \sigma_1 \xrightarrow{A,id}_{step} \sigma_2 \\
& & \wedge\ \sigma' = \sigma_2[inp[id]/(\sigma_2(inp[id]) \cup \sigma(inp[id]))] \\
& & \text{where } \sigma_1 = \sigma[inp[id]/oper]
\end{array}$$

Finally, we present two algorithms, one to execute a single step $\xrightarrow{A}_{singlestep}$ (Fig. 7), and one to execute a sequence of cycles in the clock-asynchronous semantics (Fig. 8).

Figure 7 Algorithm to execute a single-object step

1 For the object do

- Compute $nextstep(C, I)$ using the algorithm in Fig. 1
- For every transition in the step execute the actions (only updating primed variables). This requires maintaining a call stack for synchronous calls.
- Compute the next configuration
- Update the current configuration with the next configuration
- Empty the input set
- Update the variables with the values of their primed counterparts
- Reset all relevant timers

2 For all objects do

- Update the input set with the value of the primed input set (may contain events generated in current step)
- Empty the primed input set

Figure 8 Execution algorithm for the multi-object clock-asynchronous semantics

While true do

- Receive change (external event, value change, temporal event)
- Repeat: if there is an object with an unstable state, then execute a step for this object (see Fig. 7)
  Until all objects have a stable state

6. CONCLUSION

This paper presents a formal real-time requirements-level semantics for UML statecharts. The step relation is specified both declaratively and operationally and is parametrised with respect to the priority definition. The most important aspect of the semantics is the definition of communication between several co-existing and cooperating objects. The semantics supports dynamic instantiation of objects. As explained in the introduction, our semantics makes it possible to identify the differences of UML statecharts with STATEMATE statecharts.

Because it is defined in terms of a transition system, our semantics can be used to define an execution semantics for UML models at the re-

quirements level. In addition, it is the semantic basis for model checking UML requirements models. First results in model checking are reported elsewhere [9].

Future work includes, next to model checking, the study of inheritance in OO systems by applying techniques from concurrency (similar to e.g. [7]).

Acknowledgments

The authors would like to thank Maarten Fokkinga and David Jansen for commenting on previous parts of this paper.

References

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Beeck, M. von der. A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 863, pages 128–148. Springer-Verlag, 1994.

[3] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proc. COMPOS 97*, Lecture Notes in Computer Science 1536. Springer-Verlag, 1997.

[4] R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts. In *Proc. FMOODS 2000, IFIP TC6/WG6.1*. Kluwer, 2000.

[5] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[6] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[7] D. Harel and O. Kupferman. On the behavioral inheritance of state-based objects. Technical Report MCS99-12, Weizmann Institute of Science, 1999.

[8] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[9] D. N. Jansen and R. J. Wieringa. Extending CTL with actions and real-time. 2000. Submitted.

[10] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1*. Kluwer, 1999.

[11] J. Lilius and I. Porres Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *Proc. UML'99*, Lecture Notes in Computer Science 1723. Springer-Verlag, 1999.

[12] Z. Manna and A. Pnueli. Clocked transition systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering*. World Scientific, 1996.

[13] S. McMenamin and J. Palmer. *Essential Systems Analysis*. Yourdon Press, New York, New York, 1984.

[14] OMG. *Unified Modeling Language version 1.3*. OMG, July 1999.

[15] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, pages 244–265. Springer-Verlag, 1991.

[16] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines – a lightweight formal approach. In T.S.E. Maibaum, editor, *Proc. FASE 2000*, Lecture Notes in Computer Science 1783. Springer-Verlag, 2000.

[17] R. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class- and behavior diagrams. In M. Broy, D. Coleman, T.S.E. Maibaum, and B. Rumpe, editors, *Proc. PSMT'98*. Technische Universität München, TUM-I9803, 1998.