

A structural operational semantics for UML-statecharts

Shadi Al-Dehni, Jan Jürjens

*Institute for Informatics, Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 Munich, Germany*

Abstract

Object-oriented methods are widely accepted for software development in the business application domain and have also been advertised for the design of embedded and real time systems. The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based application and even to hard real time embedded systems, UML2.0 incorporates an action semantics, which adds to UML the syntax and semantics of executable actions and procedures. Action semantics refers to the ability to formally describe actions that can be analysed by a computer and executed. Statecharts are a powerful visual formalism for capturing complex behavior and apply well to both functionally decomposed systems and to object-oriented ones. In this paper, we focus on the notation of UML-statecharts. Starting with a precise textual syntax definition, we develop a precise structured operational semantics (SOS) for UML-statecharts. Besides the support of interlevel transitions and in contrast to related work, our semantics definition supports characteristic UML-statechart features like the history mechanism as well as entry and exit actions.

Keywords: object-oriented development, unified modelling language UML,

activity Diagram.

1 Introduction

UML (the Unified Modelling Language) is a non-proprietary, third generation modeling language [3]. The Unified Modelling Language is an open method used to specify, visualise, construct and document the artifacts of an object-oriented software-intensive system under development. The UML represents a compilation of "best engineering practices" which have proven successful in modelling large, complex systems, especially at the architectural level. The UML defines nine diagram types, which allow different aspects (static, behavioural, interaction, and implementation) and properties of a system design to be expressed. These diagrams are explained as following:

A use case diagram shows a set of use cases and actors (a special kind of a class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. it consists of two diagrams; **sequence diagram** and a **collaboration diagram**. a sequence diagram shows the explicit sequence of communications and is well-suited for real-time specification and for complex scenarios. A collaboration diagram shows an interaction organized around the roles in the interaction and their relationships. It does not show time as a separate dimension, so the sequence of communications and the concurrent threads must be determined using sequence numbers [1].

Activity Diagrams shows the flow from activity to activity. An activity is an ongoing nonatomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of the value. Actions encompass calling another operation, sending a signal, creating or destroying

an object, or some pure computation, such as evaluating an expression. Graphically, an activity diagram is a collection of vertices and arcs.

State Diagram shows the change of an object through time. Based upon events that occur, the state diagram shows how the object changes from start to finish.

Class Diagram shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

Component Diagram A component Diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaborations.

2 UML-statecharts Diagram

We use state machine to model the dynamic aspects of the system. For the most part, this involves specifying the lifetime of the instances of a class, a use case, or an entire system. These instances may respond to such events as signals, operation or the passing of time. When an event occurs, some activity will take place, depending in the current state of the object. An activity is an ongoing nonatomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the model or a return of a value. The state of an object is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

We can visualize a state machine in two ways: by emphasizing the flow of control from activity to activity (using activity diagrams), or by emphasizing the potential states of the objects and the transitions among those states (using statechart diagrams).

Figure 41. State diagram

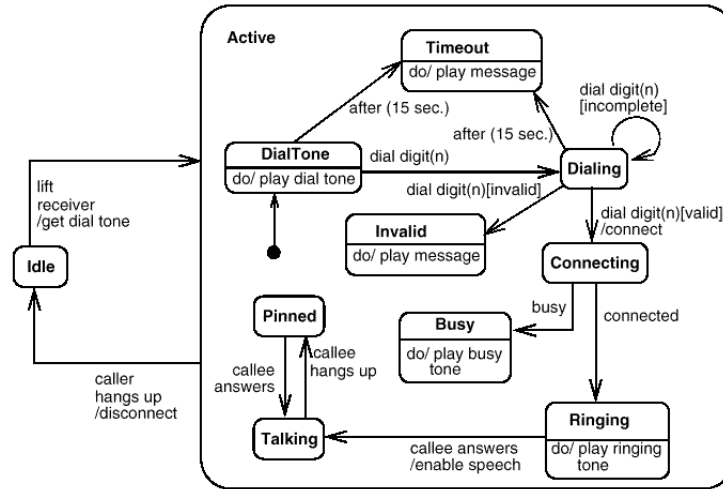


Figure 1: UML- Statechart example

2.1 UML-statecharts Properties

Some of the problems inherent in classical statecharts also exist in UML-statecharts [2]. However, there are some important differences which simplify or complicate the semantics definition, respectively. In contrast to classical statecharts, the UML-statechart language shows the following features:

- Only one input event is processed at each point of time. Especially, a generated event is not sensed within the same 'step', i.e. it can not trigger a transition within the same step.
- The trigger part of a transition label neither contains conjunctions of events nor negated events.
- If an event is taken from the input event queue which does not enable any transitions in the currently active state, then this event is simply ignored.
- If several conflicting transitions, i.e. transitions which must not be taken simultaneously, are simultaneously enabled, then a transition on a higher

level (for short: lower-first priority) [6]. In the case of classical statecharts either no priority or upper-first priority (e.g. in the STATEMATE tool) is given.

- Entry / exit action associated to states exist which are executed whenever the corresponding state is entered or exited, respectively.

The first two restrictions considerably simplify the semantics definition of UML-statecharts in comparison with classical statecharts. For example the problem of achieving global consistency does not occur. In general, causality conflicts -caused by cyclic transition executions- do not exist.

3 Syntax

UML-statechart terms. Let N, τ, π, A be countable sets of state names, transition names, events, and actions, respectively, with $\pi \subseteq A$. We denote events and actions by a, b, c, \dots and sequences of events as well as sequences of actions by α, β, \dots . For a set M let M^* denote the set of finite sequences over M . Then, the set UML-SC of *UML-statechart terms* [4] is inductively defined to be the least set satisfying the following conditions, where $n \in N$ and $en, ex \in A^*$. (The interpretations of n, en and ex will be given later.)

- **Basic term:** $s = n, (en, ex)$ is a UML-statechart term with $\text{type}(s) = \text{basic}$. Therefore s is also called a *basic term*.
- **Or-term:** If s_1, \dots, s_k are UML-statechart terms for $k > 0, \rho = 1, \dots, k, l \in \rho, HT = \{ \text{none, deep, shallow} \}$, and $T \subseteq TR =_{df} T \times \rho \times 2^N \times \prod \times A^* \times 2^N \times \rho \times HT$, then $s = [n, (s_1, \dots, s_k), l, T, (en, ex)]$ is a UML-Statechart term with $\text{type}(s) = \text{or}$. Therefore, s is also called an Or-term. Here, s_1, \dots, s_k are the subterms of s , T is the set of *transition* between the *subterms* of s , s_1 is the *default subterm* of s , l is called the *action state index* of s (or for short: the index of s), and s_l is active. Note that active state index $l \in 1, \dots, K$ denotes the l -th term within the k -tuple (s_1, \dots, s_k) of the subterms of s .

For each transition $t = (\hat{a}, i, sr, e, \alpha, td, j, ht) \in T$ we require that additional constraints are fulfilled, namely $sr \in confAll(s_i)$ and $td \in confAll(s_j)$. Furthermore, we define $name(t) =_{df} \hat{t}$, $sou(t) =_{df} s_i$, $souRes(t) =_{df} sr$, $ev(t) =_{df} e$, $act(t) =_{df} \alpha$, $tarDet(t) =_{df} td$, $tar(t) =_{df} s_j$, and $historyType(t) =_{df} ht$. $name(t)$ is called the *transition name* of t , $ev(t)$ and $act(t)$ are called the *trigger part* and *action part* of t , respectively, $sou(t)$ and $tar(t)$ are called the *source* and *target* of t , respectively, whereas $souRes(t)$ and $tarDet(t)$ are called the *source* and *target determinator*. Source restriction and target determinator provide a means for modelling an interlevel transition by a simple transition on the level of the uppermost states the interlevel transition exits and enters. The source and target of the interlevel transition are represented as additional level information by the source restriction and the target determinator. Transition t is called an interlevel transition, if its source restriction sr or its target determinator td differ from the empty set. Finally, $historyType(t)$ is called the *historytype* of t .

- **And-term:** If s_1, \dots, s_k are UML-statechart terms for $k > 0$, then $s = [n, (s_1, \dots, s_k), (en, ex)]$ is a UML-statechart term with $type(s) = \text{and}$. Thus, s is also called an And-term. Here, s_1, \dots, s_k are the (parallel) subterms of s .

In all three cases we refer to n as the root name of s and write $root(s) =_{df} n$. Furthermore, en and ex are the sequence of *entry* and the *sequence of exit actions* of s , respectively. If a_1, \dots, a_k are actions, then the sequence of actions a_1, \dots, a_k is denoted by $\langle a_1, \dots, a_k \rangle$. In particular, the empty sequence is denoted by $\langle \rangle$. We assume that all root names and transition names are mutually disjoint, so that terms and transitions within UML-statechart terms are uniquely referred to by their names. For convenience, we sometimes write "state" instead of "term" and abbreviate (s_1, \dots, s_k) by $(s_{1..k})$.

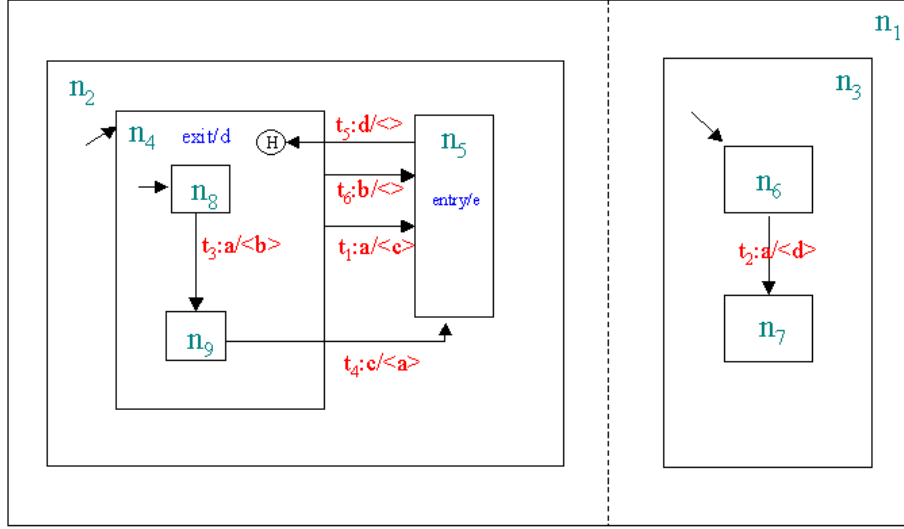


Figure 2: UML-statechart example

We explain our term syntax with Fig. 1. The term syntax of the UML-statechart of Fig.1 is as follows:

$$s_1 = [n_1, (s_2, s_3, (\langle \rangle, \langle \rangle))]$$

$$s_2 = [n_2, (s_4, s_5), l, t_1, t_4, t_5, t_6, (\langle \rangle, \langle \rangle)]$$

$$s_3 = [n_3, (s_6, s_7), l, t_2, (\langle \rangle, \langle \rangle)]$$

$$s_4 = [n_4, (s_8, s_9), l, t_3, (\langle \rangle, \langle d \rangle)]$$

$$s_5 = [n_5, (\langle e \rangle, \langle \rangle)]$$

$$s_i = [n_i, (\langle \rangle, \langle \rangle)] (6 \leq i \leq 9)$$

where s_1 is an And-state, s_2 , s_3 , and s_4 are Or-states, and s_5, \dots, s_6 are basic states. Only s_4 has an exit action and only s_5 has an entry action. For the Or-states, we have selected the default substate as the currently active substate. Therefore, the active state index equals 1 for all the Or-states considered before. A UML-statechart transition $(\hat{t}, i, sr, e, \alpha, td, j, ht)$ with $sr = \emptyset = td$ and $ht = none$ is represented by an arrow from state s_i to s_j with label $t : e/\alpha$. Therefore, the transitions t_1, t_2, \dots, t_6 are given as follows:

$$t_1 = (\hat{t}_1, 1, \emptyset, a, \langle c \rangle, \emptyset, 2, none)$$

$$t_2 = (\hat{t}_2, 1, \emptyset, a, \langle d \rangle, \emptyset, 2, none)$$

$$\begin{aligned}
t_3 &= (\hat{t}_3, 1, \emptyset, a, \langle b \rangle, \emptyset, 2, none) \\
t_4 &= (\hat{t}_4, 1, \{ng\}, c, \langle a \rangle, \emptyset, 2, none) \\
t_5 &= (\hat{t}_5, 2, \emptyset, d, \langle \rangle, \emptyset, 1, shallow) \\
t_6 &= (\hat{t}_6, 1, \emptyset, b, \langle \rangle, \emptyset, 2, none)
\end{aligned}$$

Note that components two and seven of a transition $t = (\hat{t}, i, sr, e, \alpha, td, j, ht) \in T$ - Namely i and j - of an Or-term $s = [n, (s_1, \dots, s_k), l, T, (en, ex)]$ refer to the i -th and j -th term of the k -tuple (s_1, \dots, s_k) , respectively, but not to the indexes of the state names in the k -tuple. Otherwise, e.g, transition t_1 would be given by $t_1 = (\hat{t}_1, 4, \emptyset, \alpha, \langle c \rangle, \emptyset, 5, none)$.

Transition t_4 is the only interlevel transition, because its source restriction n_9 differs from the empty set. Furthermore, t_5 is the only transition which uses the history mechanism, if its history type is shallow or deep (i.e. $historyType(t) \in \{deep, shallow\}$).

Configuration. In the following we consider different kinds of configurations. Function $conf: UML - SC \longrightarrow 2^N$ which is inductively defined along the structure of UML-statechart terms computes the (complete) *current configuration* of a given UML-statechart term s , i.e. the set of the root names of all currently active substates within s also including the root name of s .

$$\begin{aligned}
conf([n, -]) &=_{df} \{n\} \\
conf([n, (s_{1..k}, l, T, -)]) &=_{df} \{n\} \cup conf(s_l) \\
conf([n, (s_{1..k}, -)]) &=_{df} \{n\} \cup \bigcup_{i=1}^k conf(s_i)
\end{aligned}$$

For example in the UML-statechart of Fig.1 we have:

$$\begin{aligned}
conf(s_1) &= \{n_1, n_2, n_4, n_8, n_3, n_6\} \\
conf(s_2) &= \{n_2, n_4, n_8\}
\end{aligned}$$

For the sake of brevity, in the following we sometimes use subconfigurations instead of (complete) configurations. Intuitively, a subconfiguration of a UML-statechart term s is the set of all root names in the configuration of s which denote basic states. Formally, we define a function $subconf: UML-SC \longrightarrow 2^N$ by :

$$\begin{aligned}
\text{subconf}([n, -]) &=_{df} \{n\} \\
\text{subconf}([n, (s_{1..k}), l, T, -]) &=_{df} \text{subconf}(s_l) \\
\text{subconf}([n, (s_{1..k}, -)]) &=_{df} \bigcup_{i=1}^k \text{subconf}(s_i)
\end{aligned}$$

For example, in the UML-statechart of Fig.1 we have:

$$\text{subconf}(s_1) = n_8, n_6 \quad \text{subconf}(s_2) = n_8$$

Function $\text{confAll}: UML - SC \rightarrow 2^{2^N}$ applied to UML-statechart s computes the set of all *potential* configurations of s , which can be complete or *incomplete*. The term "potential" denotes that not only the currently active substate of each Or-state s' . This difference between conf and subconf on the one side and confAll on the other side implies that $\text{conf}([n, (s_{1..k}), l, T, -])$ as well as $\text{subconf}([n, (s_{1..k}), l, T, -])$ depend on active state index l , whereas $\text{confAll}([n, (s_{1..k}), l, T, -])$ does not.

The term "incomplete" denotes a configuration which results from an application of confAll to state s , where the results from an application of confAll to state s , where the recursion within confAll terminates before the basic states of s are reached. Therefore, an incomplete configuration is upward-closed with respect to the state hierarchy, but not downward-closed, whereas a complete configuration is both.

Function confAll is also defined along the structure of UML-SC:

$$\begin{aligned}
\text{confAll}([n, -]) &=_{df} \{\{n\}\} \\
\text{confAll}([n, (s_{1..k}), l, T, -]) &=_{df} \{\{n\}\} \cup c | \exists j \in \{1..K\}. c \in \text{confAll}(s_j) \cup \{\{n\}\} \\
\text{confAll}([n, (s_{1..k}, -)]) &=_{df} \{\{n\}\} \cup \bigcup_{i=1}^k c_i | c_i \in \text{confAll}(s_i) \cup \{\{n\}\}
\end{aligned}$$

Incomplete configurations are realized in the second and third case of the definition of confAll by the union with term $\{\{n\}\}$. Note that $\text{conf}(s)$ is an element of $\text{confAll}(s)$ for each UML-statechart term s , formally $\forall s \in UML - SC : \text{conf}(s) \in \text{confAll}(s)$. We explain the notion of confAll by the UML-statechart of Fig.1:

$$\begin{aligned}
\text{confAll}(s_1) &\supseteq \{\{n_1, n_2, n_4, n_8, n_3, n_6\}, \{n_1, n_2, n_4, n_9, n_3, n_6\}, \\
&\{n_1, n_2, n_4, n_8, n_3, n_7\}, \{n_1, n_2, n_4, n_9, n_3, n_7\}, \{n_1, n_2, n_5, n_3, n_7\}\} \\
\text{confAll}(s_2) &\supseteq \{\{n_2, n_4, n_8\}, \{n_2, n_4, n_9\}, \{n_2, n_5\}\}
\end{aligned}$$

In this example, we have only listed the set of all complete potential config-

urations, but not the incomplete potential ones. Having defined `confAll` and assuming that transition $t = (-, i, sr, -, -, j, -)$ with source s_i , target s_j , source restriction $sr \neq \emptyset$, and target determinant $td \neq \emptyset$ is given, now the reason for the constraints $sr \in confAll(s_i)$ and $td \in confAll(s_j)$ stated in the definition of transitions in Or-terms becomes clear. The source restriction and the target determinant are possibly incomplete) configurations of s_i and s_j and specify that transition t models an interlevel transition as follows: Source restriction sr of t specifies the source state of the interlevel transition. Target determinant td of t specifies the target state of the interlevel transition.

Let us consider an example: The only interlevel transition of UML-statechart term s_1 of Fig. 1 is $t_4 = (\hat{t}, 1, \{n_9\}, c, \langle a \rangle, \emptyset, 2, none)$, because in all other transitions both the source restriction as well as the target determinant equal the empty set. Due to the source restriction $\{n_4\}$ of t_4 this transition is represented in Fig.1 by an arrow from state s_9 (instead of s_4) to s_5 with label $t_4 : c/\langle a \rangle$.

Note that our definition of `confAll` which allows *incomplete* configuration enables -together with the conditions $sr, td \in confAll(s_i)$ - a more liberal modeling of interlevel transitions than the definition of transitions in the work of Mikk et al. [5], where source restriction and target determinant must be *complete* configurations.

As can be seen from our UML-statecharts term syntax and as in the work of [4] we do not consider the following feature of UML-statechart:

- initial, final, and junction pseudostates,
- deferred, time, and change events,
- branch segments and completion transitions,
- guards, variables, and data dependencies in transition labels,
- termination, creation, destruction of objects and send clauses within actions as well as do actions, and
- dynamic choicepoints.

4 Semantics

This section is organized as follows: At first we recall the intuitive semantics definition of UML-statechart. Afterwards we formalize the treatment of entry and exit actions. Then we define how the state resulting from transition execution is computed. Finally, we use the achieved results to formally define the semantics of UML-statecharts.

4.1 Intuition

We recall the intuitive semantics definition of UML-statecharts by considering again our example of Fig.1. Initially, UML-statechart term s_1 is in subconfiguration $\{n_8, n_6\}$. If the input event queue offers event

- b , then transition t_6 can be taken, so that the next subconfiguration of s_1 will be $\{n_5, n_6\}$ and the sequence $\langle d, e \rangle$ of actions is executed, because $\langle d \rangle$ is the sequence of exit actions of state s_4 , t_6 has the empty sequence $\langle \rangle$ as action part, and $\langle e \rangle$ is the sequence entry actions of state s_5 ,

- a , then transitions t_3 and t_2 can simultaneously be taken, so that $\{n_9, n_7\}$ will be the next subconfiguration and the sequence $\langle b, d \rangle$ of action will be executed. In this case it is not allowed that only one of both transitions is taken. Note that due to the lower-first priority in UML-statecharts the transition t_1 can not be taken if state s_8 is active, because the source s_8 of transition t_3 is on a lower level than the source s_4 of transition t_1 ,

- $e \in \prod\{a, b\}$, then the configuration does not change, because no transition can be taken.

4.2 History mechanism

The History mechanism allows to reenter an Or-state s , such that the same substate of s becomes the currently active substate as it has been the case, when s has been active before, the default substate of the Or-state becomes the currently active substate.

The history mechanism allows two kinds of scoping: the "memory" effect can be restricted to the direct substates of the considered Or-state ("Shallow", graphical symbol:"H") or can be unrestricted ("deep", graphical symbol: H^*), such that it recursively remembers the active substates along the state hierarchy down to the basic states. We will consider both cases in our semantics.

In the UML-statecharts of Fig.1 only transition t_5 uses the history mechanism with $historyType(t_5) = shallow$. If t_5 is performed, then state s_4 becomes active, if s_8 (s_9) has been active, when s_4 has been active last time before, then the default state s_8 of s_4 becomes active.

4.3 Semantic problem

The main problem in our semantics that it shall support the history mechanism which exhibits quite intricate dependencies with interlevel transitions. To clarify this point let us assume that transition t_5 in Fig.1 which uses the history mechanism (cf. $historyType(t_5) = shallow$) is modified to an interlevel transition t'_5 by changing its target determinator from empty set \emptyset to $\{n_8\}$. If t'_5 is taken one could argue that due to the history mechanism the next currently active subterm of s_4 . Alternatively, one could argue that due to the target determinator $\{n_8\}$ of t'_5 the next currently active subterm of s_4 could be s_8 . However, we will define the semantics such that the target determinator information of a transition has priority over the history mechanism. For the above-mentioned example this means that the second possibility is chosen.

4.4 Entry and Exit actions

When a UML-statechart transition t is taken, a (possibly empty) set of actions is executed: at first the sequence of exit actions of the source of t is executed (according to an inner-first approach), then the action part $arc(t)$, and finally the sequence of entry actions of t (according to an outer-first approach).

We explain the execution of entry and exit actions with the UML-statechart example of Fig.1. If transition t_1 is taken then the action sequence $\langle d, c, e \rangle$

is generated, because at first the sequence $\langle d \rangle$ of exit actions of source s_4 of transition t_1 is executed, then the action part $\langle c \rangle$ of t_1 , and finally the sequence $\langle e \rangle$ of entry actions of target s_5 of t_1 .

In general, if a transition $(-, l, -, -, \alpha, -, i, -)$ from state s_l to state s_i with action part α is taken, then a sequence $ex :: \alpha :: en$ of actions is executed, with $ex \in exit(s_l), en \in entry(s_i)$. Intuitively, $exit(s_l)$ is the set of all possible sequences of exit actions of s_l and $entry(s_i)$ is the set of all possible sequences of entry actions of s_i . Here, we assume that $::$ is an operator in infix-notation which concatenates action sequences. The mentioned intuitive meanings of functions exits and entry already indicate that we introduce a sort of nondeterminism when defining the UML-statecharts semantics. The UML definition in [6] does not define in which order the entry actions of the substates of an And-state $[n, (s_{1..k}, (en, ex))]$ have to be executed with respect to each other. The same is valid for the exit actions of the substates of an And-state. Therefore we allow each permutation of the exit actions as possible execution sequences. This nondeterminism is achieved by defining entry and exit as two set-valued functions $entry : UML - SC \longrightarrow 2^{A^*}$ and $exit : UML - SC \longrightarrow 2^{A^*}$ which are inductively defined along the structure of UML-SC as follows:

$$\begin{aligned}
entry([n, (en, ex)]) &=_{df} \{en\} \\
entry([n, (s_{1..k}, l, T, (en, ex))]) &=_{df} \{en :: en' \mid en' \in entry(s_l)\} \\
entry([n, (s_{1..k}, (en, ex))]) &=_{df} \{en :: m_1 :: \dots :: m_k \mid \\
\exists bijection b : \{1..k\} &\longrightarrow \{1..k\}. \\
m_i \in entry(s_{b(i)}) \forall i \in 1..k \\
\text{and for the exit action:} \\
exit([n, (en, ex)]) &=_{df} \{ex\} \\
exit([n, (s_{1..k}, l, T, (en, ex))]) &=_{df} \{ex' :: ex \mid ex' \in exit(s_l)\} \\
exit([n, (s_{1..k}, (en, ex))]) &=_{df} \{m_1 :: \dots :: m_k :: ex \mid \\
\exists bijection b : \{1..k\} &\longrightarrow \{1..k\}. m_i \in exit(s_{b(i)}) \forall i \in \{1..k\}\}
\end{aligned}$$

In the definition of both functions entry and exit the nondeterminism is realized by existential quantification over bijections in order to consider all possible

permutations of the corresponding sequences of entry actions and exit actions, respectively.

4.5 Computing the next state

If a UML-statechart transition t is executed, particularly its history type and - if t is an interlevel transition- its target determinator have to be considered. Therefore we define function $next$ which computes the state which results from a transition execution. Later on this function is used in the SOS rule which handles transition execution (in an OR-state)

Given a UML-statechart transition t with target s , history type $ht = historyType(t)$, and target determinator $N = tarDet(t)$ of t the function $next : HT \times N \times UML - SC \longrightarrow UML - SC$ computes the UML-statechart term

$s' = next(ht, tarDet(t), s)$ which results after execution of transition t . Note that the terms s and s' have identical static structure, only their dynamic information - specifying the currently active substates - may differ. In order to simplify the presentation of several subsequent definitions (functions $next_stop$, $default$, and the SOS rules), in the following we sometimes abstract from entry and exit actions within UML-statechart terms: for example, we write $[n]$ instead of $[n, (en, ex)]$, we write $[n, (s_{1..k}), l, T]$ instead of $[n, (s_{1..k}), l, T, (en, ex)]$, and we write $[n, (s_{1..k})]$ instead of $[n, (s_{1..k}, (en, ex))]$. Furthermore, we use the substitution notion $_{[./.]}$ as follows: If t is a term, then $t_{[a/b]}$ is the term which results from replacing all occurrences of a in t by b . Finally, for $l \in \{1, \dots, K\}$ we abbreviate $(s_1, \dots, s_{l-1}, s'_l, s_{l+1}, \dots, s_k)$ by $(s_{1..k})_{[s_l/s'_l]}$.

$$next(ht, N, [n]) =_{df} [n]$$

$$next(ht, N, [n, (s_{1..k}), l, T]) =_{df} \begin{cases} [n, (s_{1..k})_{[s_j/next(ht, N, s_j)]}, j, T] & \text{if } \exists n' \in N, j \in \{1, \dots, k\}. n' = name(s_j) \\ next_stop(ht, [n, (s_{1..k}), l, T]) & \text{otherwise} \end{cases}$$

$$next(ht, N, [n, (s_{1..k})]) =_{df} [n, (next(ht, N, s_1), \dots, next(ht, N, s_k))]$$

The second case of definition -application of function $next$ to an Or-state $[n, (s_{1..k}), l, T]$

- requires some explanation:
- If N contains a name n' of one of the state names of the substates s_1, \dots, s_k , i.e. $\exists n' \in N, j \in \{1, \dots, K\}$, such that $n' = name(s_j)$ (denoted as condition *), then active state index l will be replaced by active state index j and function *next* is recursively applied to s_j . Therefore, if $N = tarDet(t)$, then the target determinator information of t is exploited in function *next* when zooming into the state hierarchy as long as condition * is fulfilled, i.e. as long as adequate target determinator information exists.
- Otherwise, i.e. if N does not contain a name n' of one of state names of the substates s_1, \dots, s_k , function *next_stop* : $HT \times UML - SC - OR$ is called which uses the history type information to determine currently active substates of a state, with $UML - SC - OR =_{df} \{s | s \in UML - SC, type(s) = or\}$, i.e. $UML - SC - OR$ is the set of all UML-statechart terms which are Or-states. In the definition of function *next_stop* the following case distinction occurs:
 - If history type $ht = deep$, then the state $[n, (s_{1..k}), l, T]$ does not change at all.
 - If history type $ht = none$, then active state index l is replaced by active state index 1 and function *default* is used to initialize substate s_1 .
 - If history type $ht = shallow$, then active state index l does not change, but function *default* initializes all lower levels of s_l .

$$\begin{aligned}
& next_stop(ht, [n, (s_{1..k}), l, T]) \\
& =_{df} \begin{cases} [n, (s_{1..k}), l, T] & \text{if } ht = deep \\ [n, (s_{1..k})_{s_1/default(s_1)}, 1, T] & \text{if } ht = none \\ [n, (s_{1..k})_{s_l/default(s_l)}, l, T] & \text{if } ht = shallow \end{cases}
\end{aligned}$$

Note the difference between the second and third case of the definition of *next_stop*:

- The second case ($ht = none$) is independent from l , i.e. from the active state index of the currently active substate of the Or-state. *Default* (s_1) becomes the new currently active substate of s_1 .
- In contrast the third case ($ht = shallow$) depends on l , because *default*(s_l)

becomes the new currently active substate of the Or-state.

The definition of `next_stop` uses function `default`:

UML – SCUML – SC which defines for an Or-state that its currently active substate is given by its default substate.

$$\mathit{default}([n]) =_{df} [n]$$

$$\mathit{default}([n, (s_{1..k}, l, T)]) =_{df} [n, (s_{1..k})_{[s_1/default(s_1)]}, \mathbf{1}, T]$$

$$\mathit{default}([n, (s_{1..k})]) =_{df} [n, (\mathit{default}(s_1), \dots, \mathit{default}(s_k))]$$

4.6 Semantics definition

To develop a comprehensible -though price -UML-statecharts semantics definition we will modularize it as follows: First of all we will define the semantics in two phases: In the first phase we will define an auxiliary UML-statecharts semantics which only deals with processing single input events, but not with sequences of input events. In a second phase we use this auxiliary semantics to define the (complete) UML-statechart semantics dealing with processing sequences of input events. This separation already supports modularity. Furthermore, for the first phase we follow the SOS-approach of Plotkin: We take labeled transition systems as semantic domain and use SOS-rules to define the (auxiliary) semantics of UML-statecharts- restricted to the processing of single input events - in a modular way. More precisely, we will define the auxiliary semantics by a function $[[\cdot]]_{aux} : UML - SC \longrightarrow LTS$, where *LTS* is the set of *labeled transition systems* and where the (semantic) transitions work on single input events $e \in \mathbb{I}$. for the second phase we use Kripke structures as semantic domain, because this selection simplifies the processing of event sequences considerably, since Kripke structures are very appropriate for modeling that the output of one step serves as (part of) the input of the next step.

Both phases constitute an operational and modular approach, such that comprehension as well as flexibility (e.g. with respect to subsequent enhancements) are supported - without restricting preciseness.

Auxiliary semantics The auxiliary semantics $[[s]]_{aux}$ of a UML-statechart term

$s \in UML - SC$ is given by the labeled transition system $(UML - SC, L, \longrightarrow, s) \in LTS$, where

- UML-SC is the set of states,
- $L = \prod A^* \times \{0, 1\}$ is the set of labels,
- $\longrightarrow \subseteq UML - SC \times L \times UML - SC$ is the transition relation, and
- s is the start state.

For the sake of simplicity, we write $\xrightarrow[\alpha]{e, f}$ instead of $(s, (e, \alpha, f), s') \in \longrightarrow$ and $s \xrightarrow[\alpha]{e, f}$ instead of $\exists s', \alpha. s \xrightarrow[\alpha]{e, f} s'$, where s and s' are called the *source* and the *target* of these (semantic) transitions, respectively, and f is called the *stuttering flag* (or for short *flag*). We say that term s may perform a (semantic) transition with input e , output α , and flag f to term s' . If appropriate, we do not mention the input, output, and/or target of the transition. Intuitively, stuttering flag f states whether a semantic transition is performed,

- either because at least one UML-statechart transition is taken (in this case $f = 1$, denoted as *positive flag*)
- or without taking any UML-statechart transition (in this case $f = 0$, denoted as *negative flag*). In this case only target are identical. This is usually denoted as a *stuttering step*.

The flag is needed to assure that stuttering steps can only occur, if no non-stuttering step is possible.

In contrast to the work of Latella, we do not need to annotate a semantic transition with the explicit set of UML-statechart transitions which are taken when the semantic transition is performed. Instead, in our case it suffices to annotate the boolean information whether at least one UML-statechart transition is taken. This simplification reduces the complexity of the semantics and therefore could ease the implementation of the semantics. Furthermore, a better performance of the implemented semantics could result.

Transition relation \longrightarrow is defined as presented in Tabel 1 by five SOS rules using the following rule format:

name $\frac{premise}{conclusion}$ (*condition*)

$$\begin{aligned}
& \text{BAS} \frac{\text{true}}{[n] \xrightarrow[\langle \rangle]{e} [n]} \\
& \text{OR-1} \frac{(-, l, sr, e, \alpha, td, i, ht) \in T, sr \subseteq \text{conf}(s_l), s_l \xrightarrow{e} s_i}{[n, (s_{1..k}, l, T)] \xrightarrow[\text{ex}::\alpha::\text{en}]{e} [n, (s_{1..k})_{[s_i/\text{next}(ht, td, s_i)]}, i, T]} \left(\begin{array}{l} ex \in \text{exit}(s_l), \\ en \in \text{entry}(\text{next}(ht, td, s_i)) \end{array} \right) \\
& \text{OR-2} \frac{s_l \xrightarrow[\alpha]{e} s'_l}{[n, (s_{1..k}, l, T)] \xrightarrow[\alpha]{e} [n, (s_{1..k})_{[s_l/s'_l]}, l, T]} \\
& \text{OR-3} \frac{s_l \xrightarrow[\langle \rangle]{e} s_l, [n, (s_{1..k}, l, T)] \xrightarrow{e} s_i}{[n, (s_{1..k}, l, T)] \xrightarrow[\langle \rangle]{e} [n, (s_{1..k}, l, T)]} \\
& \text{AND} \frac{\forall j \in \{1, \dots, k\}. s_j \xrightarrow[\alpha_j]{e} s'_j}{[n, (s_{1..k})] \xrightarrow[\alpha]{e} [n, (s'_{1..k})]} \left(\begin{array}{l} \alpha \in \{b(1) :: \dots :: \alpha_n(k) \mid \\ \exists \text{bijection } b : \{1..k\} \rightarrow \{1..k\}\} \end{array} \right)
\end{aligned}$$

Explanation of the SOS rules:

- BAS (stuttering)

A basic state may perform a semantic transition with arbitrary input e , empty output, and negative flag such that the state does not change, i.e. that the input event is just consumed.

-OR-1 (progress) If t is a UML-statechart transition of an Or-state s with trigger part e , then s can perform a semantic transition with input e and positive flag.

- if the source restriction sr of t is a subset of the complete current configuration of currently active substate s_l of s ($sr \subseteq \text{conf}(s_l)$) and

- if s_l cannot perform a semantic transition with input e and positive flag ($s_l \not\xrightarrow{e} s_i$)

The former condition assures the enabledness of transition t , whereas the last condition assures the lower-first priority of UML-statecharts. Rule OR-1 treats the execution of entry and exit action: it is only rule in which additional entry and exit actions, i.e. entry and exit actions not already occurring in the output part of the transition in the rule's premise, can occur in the output part of the transition in the conclusion. Note that the source restriction sr and the target terminator td of a UML-statechart transition only appear in this rule: $sr \subseteq \text{conf}(s_l)$ assures the enabledness of the considered transition, whereas td - used within $\text{next}(ht, td, s_i)$ - precisely defines the target state and therefore also the entry actions to be executed. The target of the semantic transition differs from its source by changing the currently active substate from s_l to s_i ,

because s_l and s_i are the source and target of the UML-statechart transition t , respectively. Furthermore, the dynamic information of s_i is updated according to the history type ht and the target determinator td of t using function $next$. This update is performed by the substitution $(s_{1..k})_{[s_i/next(ht,td,s_i)]}$. Finally, the output of the semantic transition is given by concatenating a sequence ex of the exit actions of the old currently active substate s_l with the output part α of the UML-statechart transition t and with a sequence en of the entry actions of the new currently active substate s'_i , where s'_i is the result of updating s_i using function $next$ as explained before.

- OR-2 (propagation of progress) If a substate of an Or-state may perform a semantic transition with a positive flag, then the Or.state may perform a semantic transition with the same label.
- OR-3 (propagation of suttering) If a substate of an Or-state may perform a semantic transition with a negative flag (i.e. no UML-statechart transition can be taken within the Or-state) and if the Or-state cannot perform a semantic transition with positive flag, then the Or-state may also perform a semantic transition with the same label (in particular with negative flag). The condition, that the Or-state cannot perform a semantic transition with positive flag supports the maximality condition which will be dealt with later on in more detail.
- AND (composition)

If every substate s_j of an And-state s can perform a semantic transition with input e , output α_j , and flag b_j , then And-state s can also perform a semantic transition with the same input e , but with output α resulting from concatenating the substate outputs α_j in an arbitrary order, and with flag $\bigvee_{j=1}^k f_j$ given by the logical disjunction of all flags f_j . Here, we identify “0“ and “1“ with the boolean values “false“ and “true“, respectively, to evaluate term $\bigvee_{j=1}^k f_j$. Summing up, the SOS rules define that for every input event $e \in \prod$ and for every state $s \in UML - SC$.

- either a semantic transition $s \xrightarrow[\alpha]{e}_1 s'$ with output $\in A^*$ and state $s' \in UML - SC$ exists or

- a semantic transition $s \xrightarrow[\langle \rangle]{e} s$ exists -with empty output and without a state change.

In particular our semantics definition fulfills the maximality condition of UML-statecharts : a maximal number of non-conflicting UML-statechart transitions is taken, when a semantic transition is performed. This condition is assured by the following facts:

- The AND-rule assures that an And-term can only perform a semantic transition, if all of its (parallel) substates perform a (semantic) transition.
- The set of Or-rules make sure that performing a semantic transition with positive flag (denoting the execution of at least one UML statechart transition) has priority over performing a semantic transition with negative flag (denoting the execution of no UML-statechart transition) the premise of rule OR-3 (propagation of stutterung) can only become true, if the premise of rule OR-1 (progress) evaluates to false.

In the SOS rules we have used stutterung steps in order to simplify the formulation of (parallel) composition in rule AND: by allowing stuttering steps we can (and in fact must) assume that all its parallel substates perform a semantic transition. If we would not use stuttering steps, it would be difficult to define the semantics such that the aforementioned maximality condition will be satisfied.

As an example Fig.2 presents the auxiliary semantics, i.e. a labeled transition system lts , for the UML-statechart of Fig.1 in a graphical way as state transition diagram std , where each std -state represents an lts -state s by the current sub-configuration of s . For the sake of simplicity, Fig 2 only presents those semantics transitions which have positive flags. Semantic transitions with negative flags would have to be presented by cyclic transitions at every state. More precisely for every state s of the transition diagram of Fig 2. for which there does not exist an outgoing transition with input $e \in \prod$ a cyclic transition with label $e/\langle \rangle$ at state s would have to be drawn. Furthermore, Fig.2 only shows those states which are eachable from the start state (n_8, n_6) .

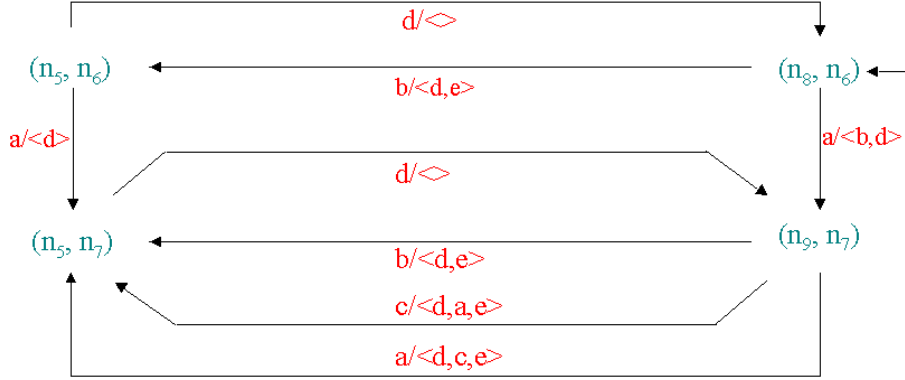


Figure 3: Auxiliary semantics of UML-statechart from Fig.1

4.7 Complete Semantics

In order to define the UML-statechart semantics in a more complete way, we have to consider that-given a sequence of input event - a UML-statechart performs a sequence of steps, such that during each of these steps

- one event of the current sequence of input events is consumed and therefore deleted from this sequence and
- a sequence of actions is generated which is added to the shorted sequence of input events resulting in a new sequence of input events to be used in the following step.

Kripke structures can be used to model this processing. In particular, they are very appropriate for modelling that the output of one step serves as the input for the following step. Therefore, after having defined the auxiliary semantics $[[\cdot]]_{aux}$ in a first phase, in a second phase we use Kripke structures and the (auxiliary) semantics $[[\cdot]] : UML-SC \rightarrow K$ for UML-statechart terms, where K is the set of Kripke structures.

The (complete) semantics $[[s]]$ of a UML-statechart term $s \in UML-SC$ is given by a Kripke structure $K = (S, st, \rightarrow) \in K$, where

- $S = UML-SC \times \prod^*$ is the set of Kripke states of K ,
- $st = (s, \varepsilon_0) \in S$ is the start state of K with $\varepsilon_0 \in \prod^*$

- $\longrightarrow \subseteq S \times S$ is the transition relation of K .

Due to the choice of Kripke structures as semantic domain we have to require $\prod = A$, because the "output" of a step of a Kripke structure serves as the "input" of the next step.

For the sake of simplicity, we write $(s,) \rightarrow (s', \alpha)$ instead of $(s, \epsilon, s', \alpha) \in \rightarrow$.

The following SOS rule (called get-inp) defines transition relation \rightarrow of the complete semantics by use of transition relation \rightarrow of the auxiliary semantics.

$$get - inp \quad \frac{s \xrightarrow[\alpha f]{\epsilon} s'}{(s, \epsilon) \rightarrow (s', \epsilon'') \mid (\exists(\epsilon, \epsilon') \in sel, \exists(\alpha, \epsilon', \epsilon'') \in join)}$$

Explanation of SOS rule get-inp:

If a sequence ϵ of events is given, such that relation sel can separate it in a single event e and the rest sequence ϵ' and if state s may perform a transition according to transition relation \rightarrow with input e (being a single event), output α , and flag f to state s' , then Kripke state (s, ϵ) may also perform a transition according to transition relation \rightarrow to state (s', ϵ'') , if relation $join$ can compose output α and rest sequence ϵ' to sequence ϵ'' .

According to the UML definition which does not define the scheduling strategy of the input event queue of UML-statecharts (and thereby following [10]) we use two relations $sel \subseteq \prod^* \times (\prod \times \prod)$ and $join \subseteq (\prod^* \times \prod^*) \times \prod^*$ which still have to be defined accordingly for a concrete scheduling strategy of the input event queue.

After having defined the semantics we will discuss the advantages and disadvantages of interlevel transitions.

- On the one hand, interlevel transitions are included in Harel's classical statecharts, as well as in UML-statecharts of all UML versions 1.x, and also in the UML 2.0 Proposal of the U2-partners [7].

- On the other hand interlevel transitions imply that UML-statecharts are not defined in a modular way, so that the definition of a compositional (UML)-statecharts semantics is impeded. Therefore interlevel transitions are not allowed in most work related to the formalization of statecharts semantics [8,9,10] (also in our previous work [11,12]).

- However, in the work presented above we have incorporated interlevel transitions within our UML-statechart term syntax, although their prohibition would have significantly simplified our syntax and semantics definition of UML-statecharts:

- Syntax:

Transitions of Or.states would neither contain source restriction nor target determinator information. Therefore also function confAll would not be necessary any more.

- Semantics:

We could skip function conf which is used in SOS-rule Or-1. Furthermore, the definition of function next would be significantly less complex.

5 Conclusion and further work

A formal semantics of UML-statechart is presented including UML-statechart features like the history mechanism as well as entry and exit actions. Furthermore, the use of the semantic namely UML-statechart terms as well as the approach of defining an SOS-style semantics results in quite a succinct and well adaptable semantics which could be used as the basis for formal analysis techniques like model checking, equivalence checking, refinement checking, consistency checking or for defining transformations between tools which support different UML-statechart dialects. In future, another additional features of UML-statecharts -e.g. guards and deferred events - within semantic definition could be considered. Another major point of future work is given by the development of adequate formal notions for equivalence, refinement, and consistency based on the semantics definition of UML behavioural notations providing the basis of a development method for UML behavioural notations.

References

- [1] Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide, Addison Wesley Longman, Inc. 1999.
- [2] Harel, D.: Statechart: A visual formalism for complex systems. *Science of computer Programming*, 8: 231-274, 1987
- [3] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 1998
- [4] Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML statechart diagrams. In: *Formal Methods for open object-based Distributed Systems*. Chapman Hall, 1999
- [5] Mikk, E., Lakhnech, Y., Siegel, M.: Hierarchical automata as modelö for Statecharts. In: *Proceedings of Asian Computing Science Conference (ASIAN '97)*. Lecture Notes in Computer Science, vol. 1345. Springer-Verlag, December 1997
- [6] OMG. OMG Unified Modeling Language Specification. Version 1.4, 2001
- [7] U2 Partners. Unified Modeling Language 2.0 Proposal, version 0.671 (draft). <http://www.u2-partners.org>, 2002
- [8] Scholz, P.: Design of Reactive Systems and their Distributed Implementation with Statecharts. PhD thesis, Munich University of Technology, Munich, Germany, August 1998
- [9] Pnueli, a., Shalev, M.: What is in a step: On the semantics of statecharts. In: Ito, T., Meyer, A.(eds.) *Theoretical Aspects of Computer Spftware (TACS '91)* Lecture Notes in Computer Science, vol. 526. Springer-Verlag, Sendai, Japan, September 1991, pp. 244-264
- [10] Levi, F.: Verification of Temporal and Real-Time Properties of Statecharts. PhD thesis, University of Pisa-Genova-Udine, Pisa, Italy, February 1997

- [11] Luetzgen, G., von der Beeck, M., Cleaveland, R.: A Compositional Approach to Statecharts Semantics. In: Proc. of ACM SIGSOFT Eighth Int. Symp. on the Foundations of Software Engineering (FSE-8). ACM, 2000, pp.120-129

- [12] von der Beeck, M.: Formalization of UML-Statecharts. In: Gogolla, M., Kobryn, C.(eds.) UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts, and Tools. Lecture Notes in Computer Science, vol. 2185. Springer-Verlag, 2001, pp. 406-421