Semantics of UML State Machines

Alexander Knapp

Institut für Informatik, Ludwig-Maximilians-Universität München
knapp@pst.ifi.lmu.de

Abstract. The abstract syntax and semantics of a simplified subclass of UML state machines is defined.

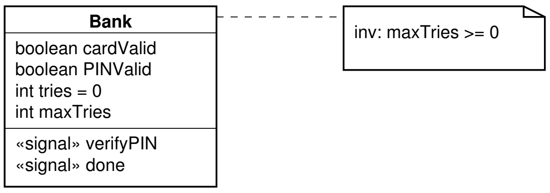
1 UML State Machines

We illustrate the main concepts of UML state machines by a simple UML model of an automatic teller machine (ATM), shown in Fig. 1: The class diagram in Fig. 1(a) specifies an (active) class Bank. Classes define *attributes*, i.e., local variables of its instances, and *operations* and *signals* that may be invoked on instances by call and send actions, respectively.

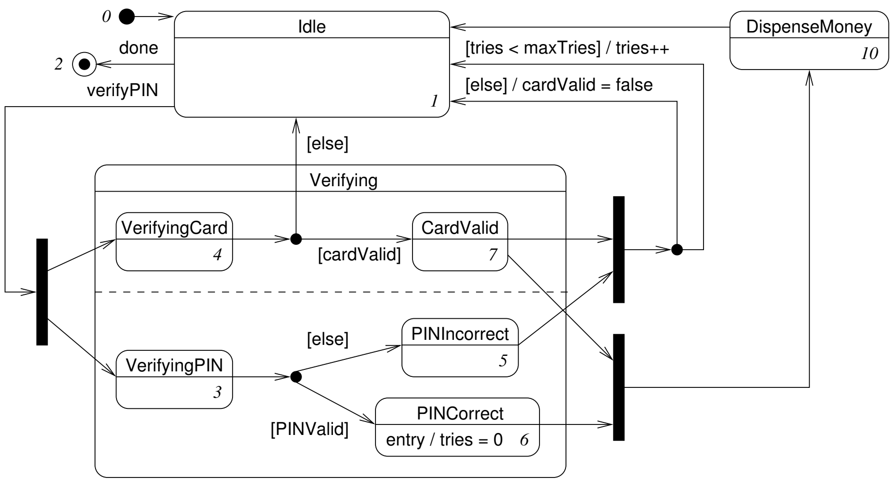
The state machine for class Bank is shown in Fig. 1(b), consisting of *states* and *transitions* between states (we number the states for short reference later on). States can be *simple* (such as Idle and DispenseMoney) or *composite* (such as Verifying); a *concurrent* composite state contains several *orthogonal regions*, separated by dashed lines. Moreover, *fork* and *join* (pseudo-)states, shown as bars, synchronize several transitions to and from orthogonal regions; *junction* (pseudo-)states, represented as filled circles, chain together multiple transitions. Transitions between states are triggered by *events*. Transitions may also be guarded by conditions and specify actions to be executed or events to be emitted when the transition is fired. For example, the transition leading from state Idle to the fork pseudostate requires signal verifyPIN to be present; the transition branch from VerifyingCard to CardValid requires the guard cardValid to be true; the transition branches to Idle set the Bank attributes tries and cardValid. Events may also be emitted by *entry* and *exit* actions that are executed when a state is activated or deactivated. Transitions without an explicit trigger (e.g. the transition leaving DispenseMoney), are called *completion transitions* and are triggered by *completion events* which are emitted when a state completes all its internal activities.

The actual state of a state machine is given by its *active state configuration* and by the contents of its *event queue*. The active state configuration is the tree of active states; in particular, for every concurrent composite state each of its orthogonal regions is active. The event queue holds the events that have not yet been handled by the machine. The *event dispatcher* dequeues the first event from the queue; the event is then processed in a *run-to-completion* (RTC) step. First, a maximally consistent set of enabled transitions is chosen: a transition is *enabled* if all of its source states are contained in the active state configuration, if its trigger is matched by the current event, and if its guard is true; two enabled transitions are *consistent* if they do not share a source state. For each transition in the set, its *least common ancestor* (LCA) is determined, i.e. the lowest composite state that contains all the transition's source and target states. The transition's main source state, that is the direct substate of the LCA containing the source states, is deactivated, the transition's actions are executed, and its target states are activated.

The example state machine simulates card and PIN validation of a bank computer. After initialization the bank computer is in state Idle. The reception of signal done leads to finalizing the state machine, whereas on reception of signal verifyPIN the verification process is started in state Verifying. If the card is invalid, the bank computer immediately returns to state Idle. If the PIN is invalid, it is checked whether the maximum number of trials is exceeded. If this is the case, the card is marked invalid; otherwise the number of trials is incremented by one. In both cases, the bank computer returns to state Idle. If the PIN is valid, the number of trials is reset to zero. If both the PIN and the card are valid, state DispenseMoney is entered from which the bank computer returns to state Idle.



(a) Class diagram



(b) State machine diagram for class Bank

Fig. 1. UML model of an ATM

2 Semantics of UML State Machines

The semantics of UML state machines is defined by an execution algorithm. This algorithm forms the basis for embedding UML state machines into temporal logic and, in particular, the symbolic execution technique. Our semantical account of UML state machines follows the semantics definition of the UML 1.5 specification [2] as closely as possible, but fills in some of the gaps of the specification. We mainly follow the ideas presented by Lilius and Porres [1]. However, we refine the definition of compound transitions and maximally consistent sets of compound transitions.

We first define the abstract syntax of the sublanguage of UML state machines from the UML specification [2] for which our semantics is valid. Apart from those language constructs which we do not discuss here (i.e. history, sync, and choice pseudostates, call and deferred events, and internal transitions), this definition introduces the following restriction: A transition from an initial pseudostate must target a non-pseudostate contained in the same composite state as the initial pseudostate. Abandoning this restriction would lead to a more intricate definition of compound transitions. In particular, when an initial pseudostate could target a junction pseudostate, this would amount to treating dynamic choice points as the guards on the transitions from such a junction pseudostate have to be evaluated only after the state containing the initial pseudostate has been entered. However, such a transition could leave the initial pseudostate's container and thus render the choice of transitions to be executed invalid. This case has no relevance in most practical applications.

The semantics of UML state machines is defined in two steps: First, we present a procedure for statically computing the configurations and the compound transitions of a state machine. Due to our syntactical restriction, compound transitions are trees of state machine transitions (describing the chaining of junctions, forks, and entries of states) with a possible fan-in prefix (describing joins) that represent a move from a configuration into another configuration. Second, we define an algorithm for run-to-completion steps which first computes a maximally consistent set of compound transitions for a given event and then executes the set of compound transitions.

2.1 Abstract Syntax of UML State machines

We assume an expression language Exp that at least includes boolean expressions (like true, false, $e_1 \wedge e_2$, etc.) and an action language Act that at least includes a skip statement, and sequential (;) and parallel (||) composition of statements. Furthermore, we assume a set of events $Event$ which includes $*$ denoting a completion event.

A *state* s has a kind $kind(s) \in \{\mathsf{initial}, \mathsf{final}, \mathsf{simple}, \mathsf{composite}, \mathsf{concurrent}, \mathsf{junction}, \mathsf{join}, \mathsf{fork}\}$, an entry action $entry(s) \in Act$, and an exit action $exit(s) \in Act$. A *pseudostate* is a state s with $kind(s) \in \{\mathsf{initial}, \mathsf{junction}, \mathsf{join}, \mathsf{fork}\}$; we require that $entry(s) = \mathsf{skip}$ and $exit(s) = \mathsf{skip}$ for each pseudostate s. A *composite* state is a state s with $kind(s) \in \{\mathsf{composite}, \mathsf{concurrent}\}$.

A *state hierarchy* is given by a tree (S, E) where S is a finite set of states and $E \subseteq S \times S$ a non-empty substate relation such that the constraints below are satisfied. We write $substates(s) = \{s' \in S \mid (s, s') \in E\}$ for the substates of state s:

1. If $\mathit{substates}(s) \neq \emptyset$ then $\mathit{kind}(s) \in \{\textsf{composite}, \textsf{concurrent}\}$.
2. If $\mathit{kind}(s) = \textsf{concurrent}$ then $\#\mathit{substates}(s) \geq 2$ and $\mathit{kind}(s') = \textsf{composite}$ for all $s' \in \mathit{substates}(s)$.
3. If $\mathit{kind}(s) = \textsf{composite}$ then $\#\{s \in \mathit{substates}(s) \mid \mathit{kind}(s') = \textsf{initial}\} \leq 1$.

We further write $\mathit{container}(s)$ for the container state of state s if s is not the root state; $\mathit{substates}^+(s) = \{s' \in S \mid (s, s') \in E^+\}$ for the set of transitive substates of state s; $\mathit{substates}^*(s) = \mathit{substates}^+(s) \cup \{s\}$ for the set of reflexive, transitive substates of s; and $\mathit{initial}(s)$ for the initial state contained in the composite state s if it exists. The *least common ancestor* of a set of states $M \subseteq S$ not containing the root state, denoted by $\mathit{lca}(M)$, is the least composite state c w.r.t. E such that $M \subseteq \mathit{substates}^+(c)$; the *least common reflexive ancestor* of $M \subseteq S$, written $\mathit{lca}^=(M)$, is the least state s w.r.t. E such that $M \subseteq \mathit{substates}^*(s)$.

Given a state hierarchy $H = (S, E)$, a *transition* t over H has a source state $\mathit{source}(t) \in S$, a target state $\mathit{target}(t) \in S$, a triggering event $\mathit{trigger}(t) \in \mathit{Event}$, a guard expression $\mathit{guard}(t) \in \mathit{Exp}$, and an effect action $\mathit{effect}(t) \in \mathit{Act}$, such that the following constraints are satisfied:

1. $\mathit{kind}(\mathit{source}(t)) \neq \textsf{final}$.
2. $\mathit{kind}(\mathit{target}(t)) \neq \textsf{initial}$.
3. If $\mathit{kind}(\mathit{source}(t)) = \textsf{initial}$ then $\mathit{target}(t)$ is not a pseudostate.
4. If $\mathit{kind}(\mathit{source}(t)) = \textsf{initial}$ then $\mathit{container}(\mathit{target}(t)) = \mathit{container}(\mathit{source}(t))$.
5. If $\mathit{kind}(\mathit{source}(t)) = \textsf{fork}$ then $\mathit{target}(t)$ is not a pseudostate.
6. If $\mathit{kind}(\mathit{target}(t)) = \textsf{join}$ then $\mathit{source}(t)$ is not a pseudostate.
7. If $\mathit{kind}(\mathit{source}(t)) = \textsf{composite}$ then $\mathit{kind}(\mathit{container}(\mathit{source}(t))) \neq \textsf{concurrent}$.
8. If $\mathit{kind}(\mathit{target}(t)) = \textsf{composite}$ then $\mathit{kind}(\mathit{container}(\mathit{target}(t))) \neq \textsf{concurrent}$.
9. If $\mathit{kind}(\mathit{source}(t)) \in \{\textsf{initial}, \textsf{fork}, \textsf{join}\}$ then $\mathit{guard}(t) = \textsf{true}$.
10. If $\mathit{kind}(\mathit{target}(t)) = \textsf{join}$ then $\mathit{guard}(t) = \textsf{true}$.
11. If $\mathit{kind}(\mathit{source}(t)) = \textsf{initial}$ then $\mathit{effect}(t) = \textsf{skip}$.
12. If $\mathit{source}(t)$ is a pseudostate, then $\mathit{trigger}(t) = *$.

A *state machine* (for a class C) is given by a pair (H, T) where $H = (S, E)$ is a state hierarchy and T a finite set of transitions over H such that the constraints below are satisfied for all $t \in T$. We write $\mathit{outgoings}(s)$ for the set $\{t \in T \mid \mathit{source}(t) = s\}$; $\mathit{incomings}(s)$ for the set $\{t \in T \mid \mathit{target}(t) = s\}$; $\mathit{sources}(M)$ for the set $\{\mathit{source}(t) \mid t \in M\}$; and $\mathit{targets}(M)$ for the set $\{\mathit{target}(t) \mid t \in M\}$:

1. If $\mathit{kind}(s) = \textsf{initial}$ then $\#\mathit{outgoings}(s) = 1$.
2. If $\mathit{kind}(s) = \textsf{junction}$ then $\#\mathit{incomings}(s) = 1$ and $\#\mathit{outgoings}(s) \geq 1$.
3. If $\mathit{kind}(s) = \textsf{fork}$ then $\#\mathit{incomings}(s) = 1$ and $\#\mathit{outgoings}(s) \geq 2$.
4. If $\mathit{kind}(s) = \textsf{join}$ then $\#\mathit{outgoings}(s) = 1$ and $\#\mathit{incomings}(s) \geq 2$.
5. If $\mathit{kind}(s) = \textsf{fork}$ then there is an $s' \in S$ with $\mathit{kind}(s') = \textsf{concurrent}$ such that $\mathit{targets}(\mathit{outgoings}(s)) \subseteq \mathit{substates}^+(s') \setminus \mathit{substates}(s')$ and the following holds: if $t, t' \in \mathit{outgoings}(s)$ such that $\{\mathit{target}(t), \mathit{target}(t')\} \subseteq \mathit{substates}^+(s'')$ for some $s'' \in \mathit{substates}^+(s')$ then $t = t'$.
6. If $\mathit{kind}(s) = \textsf{join}$ then there is an $s' \in S$ with $\mathit{kind}(s') = \textsf{concurrent}$ such that $\mathit{sources}(\mathit{incomings}(s)) \subseteq \mathit{substates}^+(s') \setminus \mathit{substates}(s')$ and the following holds: if $t, t' \in \mathit{incomings}(s)$ such that $\{\mathit{source}(t), \mathit{source}(t')\} \subseteq \mathit{substates}^+(s'')$ for some $s'' \in \mathit{substates}^+(s')$ then $t = t'$.

Conditions (5) and (6) require forks and joins to come from and go to different orthogonal regions of a concurrent composite state.

2.2 Configurations and Compound Transitions

The *configurations* of a state machine $((S, E), T)$ are given by the smallest subsets C of S that satisfy the following conditions:

1. The root state of S is in C.
2. No state $s \in C$ is a pseudostate.
3. If $kind(s) = \mathsf{composite}$ then there is a single $s' \in C$ such that $container(s') = s$.
4. If $kind(s) = \mathsf{concurrent}$ then all states $s' \in S$ with $container(s') = s$ are in C.

In particular, composite states are or-states, concurrent states and-states.

The *compound transitions* of a state machine $((S, E), T)$ represent semantically complete transition paths that originate from a set of non-pseudostates and target a set of simple states. More precisely, a compound transition consists of three parts: The optional tail part of a compound transition may have multiple transitions in T originating from a set of mutually orthogonal regions that are joined by a join pseudostate. The middle part of a compound transition is a finite chain of transitions in T joined via junction pseudostates. Finally, the optional head part of a compound transition is a tree of transitions in T: If a transition in the middle part of a compound transition or in its head part itself targets a composite state the head part continues at the initial state of this composite transition; if a transition targets a concurrent composite state the head part continues at all initial states of the orthogonal regions of the concurrent composite state; if a transition targets a fork pseudostate the head part continues with the transitions outgoing from the fork pseudostate which target mutually orthogonal regions and simultaneously continues at the initial states of all those orthogonal regions that are not targeted by transitions outgoing from the fork pseudostate.

In the ATM example, the compound transitions outgoing from VerifyingCard just consist of middle parts:

$$\langle\{\}, \langle \mathsf{VerifyingCard} \rightarrow \mathsf{junction}, \mathsf{junction} \rightarrow \mathsf{Idle}\rangle, \{\}\rangle,$$
$$\langle\{\}, \langle \mathsf{VerifyingCard} \rightarrow \mathsf{junction}, \mathsf{junction} \rightarrow \mathsf{CardValid}\rangle, \{\}\rangle.$$

The fork transition from Idle consists of a middle part and a tail part:

$$\langle\{\}, \langle \mathsf{Idle} \rightarrow \mathsf{fork}\rangle, \{\langle \mathsf{fork} \rightarrow \mathsf{VerifyingCard}\rangle, \langle \mathsf{fork} \rightarrow \mathsf{VerifyingPIN}\rangle\}\rangle\ .$$

The join transition to DispenseMoney consists of a head part and a middle part:

$$\langle\{\langle \mathsf{CardValid} \rightarrow \mathsf{join}\rangle, \langle \mathsf{PINCorrect} \rightarrow \mathsf{join}\rangle\}, \langle \mathsf{join} \rightarrow \mathsf{DispenseMoney}\rangle, \{\}\rangle\ .$$

We present an algorithm for computing the compound transitions of a state machine $((S, E), T)$ in two steps: The procedure compounds in Fig. 2 for computing the compound transitions outgoing from a non-pseudostate in S relies on the procedure forwardTrees in Fig. 3 that computes the middle and head parts of compound transitions,

```
compounds($state$) $\equiv$
  $compounds \leftarrow \emptyset$
  $joinTargets \leftarrow \emptyset$
  for $forwardTree \in$ forwardTrees($state$) do
    if join $\in \{kind(s) \mid s \in targets(forwardTree)\}$
      then $joinTarget \leftarrow choose(\{s \mid s \in targets(forwardTree) \wedge kind(s) =$ join$\})$
        if $joinTarget \notin joinTargets$
          then $joinTargets \leftarrow joinTargets \cup \{joinTarget\}$
            for $joinFT \in$ forwardTrees($joinTarget$) do
              $compounds \leftarrow compounds \cup \{\langle incomings(joinTarget), joinFT\rangle\}$
            od
        fi
      else $compounds \leftarrow compounds \cup \{\langle\emptyset, forwardTree\rangle\}$
    fi
  od
  $compounds \rfloor$
```

Fig. 2. Compound transitions algorithms (1).

also called *forward trees*, outgoing from an arbitrary state in S. Herein, the (simple) target states of the leaf transitions of a forward tree f are denoted by $targets(f)$.

Note that our definition of compound transitions deviates from the explanations in the UML specification [2]. There, compound transitions are not required to target simple states only, but may as well stop at composite states. The proper initialization of composite and concurrent composite states is left to the entry procedure for composite states. In fact, as mentioned above, this is necessary for handling choice pseudostates and initial in conjunction with junction pseudostates.

The notions of source states, target states, trigger, guard, and effect are transferred from transitions to compound transitions in the following way: The *source states* of a compound transition τ, written $sources(\tau)$, are the source states of the transitions in the tail part of τ, if τ shows a tail part, and the source state of the middle part, otherwise. Analogously, the target states of τ, written $targets(\tau)$, are the target states of the transitions in the head part of τ, if τ shows a head part, and the target state of the middle part, otherwise. The *trigger* of τ is the set of triggers of the transitions in the tail part of τ, if τ shows a tail part, and the trigger of the first transition in the middle part otherwise. The *guard* of τ is the conjunction of all guards of transitions in τ. Finally, the *effect* of τ is the sequential composition of the effects of the tail, the middle, and the head part of τ, where the effects in the tail and the head are conjoined in parallel whereas the effects in the middle part are composed sequentially. These definitions are naturally extended to sets of compound transitions which show the same trigger.

We recall some notions on compound transitions τ from the UML specification that will be used for the definition of the execution semantics of state machines, in particular, when computing maximally conflict free sets of compound transitions in a given configuration C: The *main source state* of τ, $mainSource(\tau)$, is given by the state $s = lca(lca^{=}(sources(\tau)), lca^{=}(targets(\tau)))$ if $kind(s) =$ concurrent, and it is given by the state $s' \in substates(s)$ with $lca^{=}(sources(\tau)) \in substates^{*}(s')$, oth-

```
forwardTrees($state$) $\equiv$
  $forwardTrees \leftarrow \emptyset$
  for $outgoing \in outgoings(state)$ do
    $target \leftarrow target(outgoing)$
    if $kind(target) =$ junction
      then for $targetFT \in$ forwardTrees($target$) do
             $forwardTrees \leftarrow forwardTrees \cup \{\langle outgoing, targetFT\rangle\}$
           od
    elsif $kind(target) =$ fork
      then $container \leftarrow lca(targets(outgoings(target)))$
           $regionsFTs \leftarrow \langle\rangle$
           for $region \in substates(container)$ do
             $regionTs \leftarrow \{t \in outgoings(target) \mid target(t) \in substates(region)\}$
             if $regionTs \neq \emptyset$
               then $transition \leftarrow choose(regionTs)$
                    if $kind(target(transition)) \in \{$composite, concurrent$\}$
                      then $regionFTs \leftarrow \langle\rangle$
                           for $regionFT \in$ forwardTrees($target(transition)$) do
                             $regionFTs \leftarrow regionFTs \cup \{\langle transition, regionFT\rangle\}$
                           od
                           $regionsFTs \leftarrow regionsFTs :: \langle regionFTs\rangle$
                      else $regionsFTs \leftarrow regionsFTs :: \langle\{\langle transition\rangle\}\rangle$
                    fi
               else $regionsFTs \leftarrow regionsFTs :: \langle$forwardTrees($initial(region)$)$\rangle$
             fi
           od
           $n \leftarrow \#substates(container)$
           for $\langle regionFTs_i\rangle_{1\leq i\leq n} \in regionsFTs$ do
             $forwardTrees \leftarrow forwardTrees \cup \{\langle outgoing, \langle regionFTs_i\rangle_{1\leq i\leq n}\rangle\}$
           od
    elsif $kind(target) =$ composite
      then for $initialFT \in$ forwardTrees($initial(target)$) do
             $forwardTrees \leftarrow forwardTrees \cup \{\langle outgoing, initialFT\rangle\}$
           od
    elsif $kind(target) =$ concurrent
      then $regionsFTs \leftarrow \langle\rangle$
           for $region \in substates(target)$ do
             $regionsFTs \leftarrow regionsFTs :: \langle$forwardTrees($initial(region)$)$\rangle$
           od
           $n \leftarrow \#substates(target)$
           for $\langle regionFTs_i\rangle_{1\leq i\leq n} \in regionsFTs$ do
             $forwardTrees \leftarrow forwardTrees \cup \{\langle outgoing, \langle regionFTs_i\rangle_{1\leq i\leq n}\rangle\}$
           od
      else $forwardTrees \leftarrow forwardTrees \cup \{\langle outgoing\rangle\}$
    fi
  od
  $forwardTrees\rfloor$
```

Fig. 3. Compound transitions algorithms (2).

erwise. The *main target state* of τ, $\mathit{mainTarget}(\tau)$ is defined analogously, but exchanging $\mathit{sources}(\tau)$ and $\mathit{targets}(\tau)$. The set of states *exited* by τ in configuration C, $\mathit{exited}(C, \tau)$, consists of $\mathit{substates}^*(\mathit{mainSource}(\tau)) \cap C$. The set of states *entered* by τ in configuration C, $\mathit{entered}(\tau)$, is $\mathit{substates}^*(\mathit{mainTarget}(\tau)) \cap C$. Again, the definitions for $\mathit{entered}$ and exited are naturally extended to sets of compound transitions.

Two compound transitions τ_1 and τ_2 are *in conflict* in configuration C, written $\tau_1 \sharp_C \tau_2$, if $\mathit{exited}(C, \tau_1) \cap \mathit{exited}(C, \tau_2) \neq \emptyset$; more generally, a compound transition τ is in conflict with a set of compound transitions T in configuration C, written $\tau \sharp_C T$, if $\tau \sharp_C \tau'$ for some $\tau' \in T$. If $\tau_1 \sharp_C \tau_2$ let S_1 and S_2 be the sets of states in $\mathit{sources}(\tau_1)$ and $\mathit{sources}(\tau_2)$, resp., that show the maximal numerical distance from the root state of (S, E); τ_1 is *prioritized* over τ_2 in configuration C, written $\tau_1 \prec_C \tau_2$, if $S_1 \subseteq \mathit{substates}^+(S_2)$. Again, $\tau \prec_C T$ for a compound transition τ and a set of compound transitions T in configuration C if $\tau \prec_C \tau'$ for some $\tau' \in T$.

2.3 Run-To-Completion Semantics

The execution semantics of a UML state machine is described in the UML specification as a sequence of *run-to-completion steps*. Each such step is a move from a configuration of the state machine to another configuration. The sequence of steps starts in the *initial configuration* of the state machine, i.e., the configuration that is targeted by the forward tree outgoing from the initial state of the root state of the state machine's state hierarchy. In a run-to-completion step from some configuration, first, an event is fetched from the event queue. Second, a maximally consistent set of enabled compound transitions outgoing from the states of the current configuration and whose guards are satisfied is chosen. If such a set, also called a step, exists, all its compound transitions are fired simultaneously: First, all states that are exited by the step are left in an inside-out manner, executing the exit actions of these states; each such that is marked to be not completed, as it is not part of the configuration any more. Second, the gathered effect of the step is executed. Third, all states that are entered by the step are entered in an outside-in manner, executing the entry actions of these states. Furthermore, after executing the entry action of a state this state is marked as complete, i.e. a completion event for this state is generated.

More formally, let $((S, E), T)$ be a state machine. We assume a structure of *environments* η for state machines that provides the following primitive operations: An event can be fetched by `fetch`(η); the completion of a state s can be recorded by `complete`(η, s); the revocation of a state s from being completed can be recorded by `uncomplete`(η, s); a statement a can be executed by `exec`(η, a); given a configuration C and an event e all compound transitions of $((S, E), T)$ that are triggered by e can be computed by `enabled`(η, C, e); and, finally, the validity of an expression g can be checked by $\eta \models g$.

The enabledness of compound transitions in a configuration C by an event e is indeed solely defined on the basis of the triggers of compound transitions and thus only involves the completed states that have been previously recorded with the environment. The fireable sets of compound transitions, which are maximally consistent sets of enabled compound transitions are computed by the steps algorithm in Fig. 4(a). The execution of a state machine in some configuration and some environment is defined by the

```
steps$(env, conf, event)$ $\equiv$
  $\lceil transitions \leftarrow \texttt{enabled}(env, conf, event)$
   $\{step \mid \langle guard, step\rangle \in \mathsf{steps}(conf, transitions) \wedge env \models guard\}\rfloor$

steps$(conf, transitions)$ $\equiv$
  $\lceil steps \leftarrow \{\langle \mathsf{false}, \emptyset\rangle\}$
   for $transition \in transitions$ do
      for $\langle guard, step\rangle \in \mathsf{steps}(transitions \setminus \{transition\})$ do
         if $transition \sharp_{conf}\ step$
           then if $transition \prec_{conf} step$
                  then $guard \leftarrow guard \wedge \neg\, guard(transition)$
                fi
           else $step \leftarrow step \cup \{transition\}$
                $guard \leftarrow guard \wedge guard(transition)$
         fi
         $steps \leftarrow steps \cup \{\langle guard, step\rangle\}$
      od
   od
   $steps\rfloor$
```

(a) Transition selection algorithm.

```
RTC$(env, conf)$ $\equiv$
  $\lceil\langle event, env\rangle \leftarrow \texttt{fetch}(env)$
   $steps \leftarrow \mathsf{steps}(env, conf, event)$
   if $steps \neq \emptyset$
     then choose $step \in steps$
          $\langle env, conf\rangle \leftarrow \mathsf{fire}(env, conf, step)$
   fi
   $\langle env, conf\rangle\rfloor$
```

(b) Run-to-completion step algorithm.

```
fire$(env, conf, step)$ $\equiv$
  $\lceil$for $state \in insideOut(exited(conf, step))$ do
      $env \leftarrow \texttt{exec}(env, exit(state))$
      $conf \leftarrow conf \setminus \{state\}$
      $env \leftarrow \texttt{uncomplete}(env, state)$
   od
   $env \leftarrow \texttt{exec}(env, effect(step))$
   for $state \in outsideIn(entered(conf, step))$ do
      $env \leftarrow \texttt{exec}(env, entry(state))$
      $conf \leftarrow conf \cup \{state\}$
      $env \leftarrow \texttt{complete}(env, state)$
   od
   $\langle env, conf\rangle\rfloor$
```

(c) Transition firing algorithm.

Fig. 4. State machine execution algorithms.

RTC algorithm in Fig. 4(b) which uses the algorithm for firing a compound transitions step in Fig. 4(c).

References

1. Johan Lilius and Iván Porres Paltor. Formalising UML State Machines for Model Checking. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 430–445. Springer, Berlin, 1999.

2. Object Management Group. Unified Modeling Language Specification, Version 1.5. Specification, OMG, 2003. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`.