INSTITUT FÜR INFORMATIK

Semantics of UML 2.2 State Machines in Rewriting Logic

Jens Schönborn
Marcel Kyas

Bericht Nr. 1202
January 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Semantics of UML 2.2 State Machines in Rewriting Logic

Jens Schönborn
Marcel Kyas

Bericht Nr. 1202
January 2012
ISSN 2192-6247

e-mail: jes@informatik.uni-kiel.de, marcel.kyas@fu-berlin.de

Abstract While the semantics of (labeled) transition systems and the relations between these are well understood. However, the same has not yet been achieved for UML 2.2 state machines. Their many semantics often is defined in terms of labeled transition systems, because the standard document is ambiguous. A formal, modular core semantics for UML 2.2 state machines is given in rewriting logic.

Keywords. UML, state machine, semantics.

1 Introduction

UML state machines [1] are a graphical formalism used for specifying behavior of objects. They are derived from Harel statecharts [2] and their object-oriented version [3].

The specification of UML state machines (from now on just *state machines*) is informal and leaves parts open (asking the user to fill in details), contains semantic variation points (asking the user to decide between many options), and is in other places ambiguous, contradictory, or incomplete. There are many publications on the semantics of UML state machines, a.o., [4–16]. Their differences are the result of decisions on the variation points and the open parts. This multitude not only reflects the ambiguity of the UML standards, but also the evolution of the standard. For example, many problems identified by Fecher et al. [17] have been rectified, while new issues have been introduced.

Our contribution is identifying a common core semantics for state machines from the standard [1]. We present a formal and modular core semantics for state machines in rewriting logic [18]. Due to the modularisation we can treat all the above mentioned problems as semantics variations.

The rewriting logic semantics is executable in Maude [19] and allows simulation and analysis of the state machine[1]. This enables us to check whether the differences between the many semantics are revealed by a particular state machine. To this end, the formalization is modular: Modules with explicit interfaces allow to supply parts that were left open, select modules to decide on semantic variation points, or experiment with other aspects of the semantics.

[1] The complete code can be found at `http://trac.rtsys.informatik.uni-kiel.de/trac/kieler/raw-attachment/wiki/Projects/UMLSim/UML_SM_Maude.zip`

Among the difficult parts of the semantics are a correct,i. e., meaningful, formalization of history pseudo states and the priority rule used to determine which set of transitions fire. We discuss this in our previous article [17] in detail. While the former is seldom treated, the latter is invariably in conflict with the official description in the standard.

2 Syntax

This section describes the syntax used as input for the Maude interpreter. In the following all examples are based on this state machine. The basic concepts of UML 2.2 state machines are states and labeled transitions between them. The state machine is just a simple example, it starts execution in `S1` and moves to `S2` when a `failure` occurs. Then it tries to `reconnect`. Finally it moves to a final state `Final0` when it is done.

2.1 Events

Events are encoded as strings with the constructor depicted in Listing 6, e.g., the reconnect event in Fig. 1 is encoded as follows `ev:"reconnect"`.

2.2 States and Regions

Vertices describe the states and regions of a state machine. Their identifiers are strings. For every type of vertex there is an own constructor for identifiers, e.g., `R "region0"`. The hierarchical structure of the state machine is encoded into the states, e.g., `root "R0" : C "S1" : R "R1" : F "Final0"`. The final state `Final0` is directly contained by the top level state `S1`. The constructors for all types of vertices can be found in Table 7. Note that the only constructor for names of vertices with a single argument is `root` all others require the name of their containing vertex as a second argument. This ensures that the hierarchical structure forms a tree. Also note that we only use `ModState` instead of `State` as identifier for the sort of state machine states because `State` is already defined in the model-checker module.

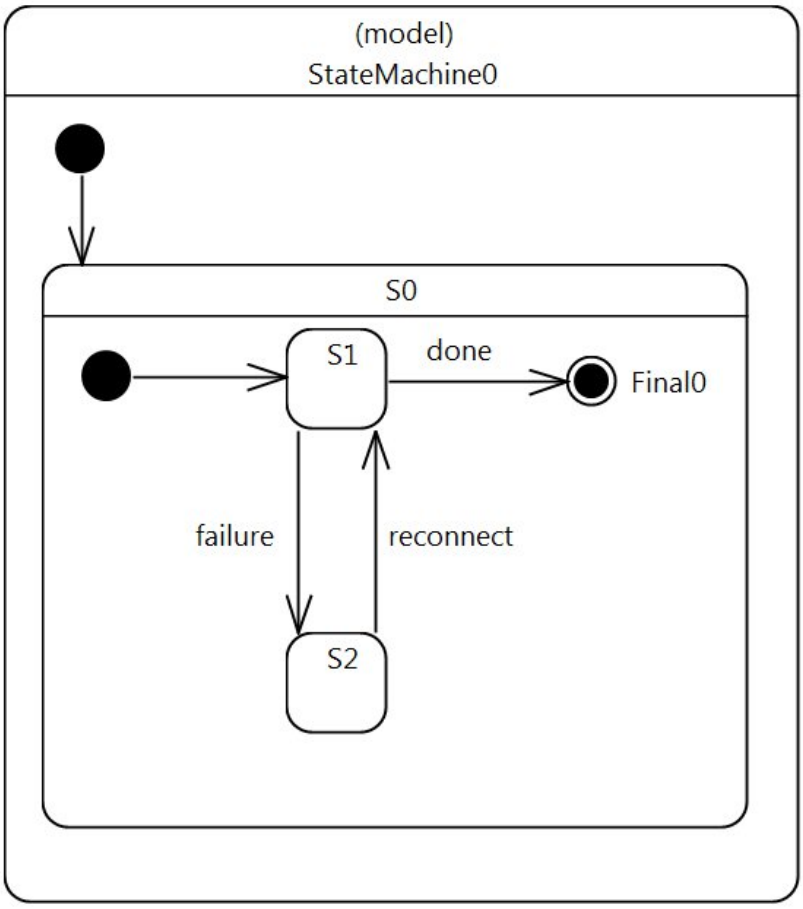

Figure 1. Example state machine

2.3 Actions

Actions are modeled as variable reference sets Listing 8, e.g., for `x:=x+y action Var-x, Var-y read Var-x write`. Interfering actions are detected as depicted in Listing 9.

2.4 Transitions

Basic transitions Listing 10 connect states and pseudostates. They are restricted to exactly one source and one target, e.g., `basicTrans "trS22-S1" (S2) true skip (S1).`

Compound transition Listing 11 are composed of basic transitions, e.g., `simpleTrans "ctrS22-S1" reconnect trS22-S1 R1`. The last argument stores the transitions scope. This can be statically determined during the transformation form the model top the input code and is only added for efficiency.

2.5 State Machines

Entry and exit actions of states are encoded as mappings from states to actions Listing 12, e.g., `entryAc S1 (action Var-x, Var-y read Var-x write)` . The information about default states determined by deep and shallow history as well as initial pseudo states is encoded as mappings from regions to a directly contained states, e.g., `defR1 = R1 default S1.`

A state machine is defined by a set of vertices, mappings for the default states and actions of the states and a set of compound transitions as shown in Listing 13. Note that we have omitted some of the `include` statements for the sake of brevity.

The semantics syntax module for state machines simply defines the states of execution of a state machine with additional information where necessary (see Listing 14). This is the case

1. when the state machine is in a stable configuration `stableC`, or
2. when an event has been selected for execution `eventSelC`, or
3. when all sets of fireable transitions are determined `fireC`, or
4. when a set of fireable transitions is chosen and when then execution, or of a single compound transition is finished `unstableC`
5. when states are left for a compound transition `leaveC`, or
6. when the basic transitions of a compound, or transition are executed `execTransC` or `execChoiceOut` if a choice pseudostate is involved, or
7. when states are entered for a compound transition `enterC`, or

8. when the execution of a run-to-completion step is done `doneC`.

Note that the last state of execution `doneC` is only added in order to simplify observation of the end of a run-to-completion step. Now one can define propositions on the states of execution. Listing. 1 presents two examples: This first one asks whether the state machine makes progress, the second one asks whether the state machine reaches a configuration where a set of states is active.

Listing 1. Propositions on sates of execution

```
139 op isDone_ : MState -> Bool .
140 eq isDone (maState (doneC<STATEC> V <HISTC> HC <ENDCONF>) E) = true .
141 eq isDone msst = false [owise] .
142
143 op statesActive : Verts -> Prop .
144 ceq maState (stableC<STATEC> V <HISTC> HC <ENDCONF>) E |=
145         statesActive(V2) = true if (V2 subset V) .
146 eq msst |= statesActive(V) = false [owise] .
```

3 Semantics

Since the different parts of the semantic, e.g., actions and transition priority, are partitioned into different modules these can be easily exchanged to make different decisions on the semantic variation points.

The first rule in Listing 15 selects an event triggering transitions which avoids discarding. The second rule determines the sets of fireable transitions according to the event. The third rule selects one of these for execution. This corresponds to the standard where the selection is defined to be non-deterministic. Then one compound transition is selected for execution and the states left are determined by the last two rules. Again the non-deterministic selection is according to the standard [1, p. 566].

The first two rules in Listing 16 leave the regions and states according to the standard, i.e., innermost first. They do not observe whether the last state is left. The set of states to be left has to be empty when the third applied finishing the exiting of states and regions for a transition.

Thereafter the associated basic transitions incoming into a pseudo state are processed as depicted in Listing 17. For choice pseudo states the current state config-

uration has to be considered. Note that transitions not involving a pseudo state are modeled with no incoming and one outgoing basic transition. The last rule finishes execution of basic transitions.

The first two rules in Listing 18 enter the regions and states according to the standard, i.e., outermost first. This is completed by the third rule similar to exiting states and region.

The first rule in Listing 19 finishes the run-to-completion step when there are no more transitions to be executed. At this point the succeeding state and history configurations are determined. When the state machine is done it changes into a stable state called stable configuration in the UML standard.

3.1 State Machine to Semantics Interface

In order to keep the semantic rules simple and short the static information of the state machine under investigation defined as an equation `SMINT` as shown in Listing 3. Sets of elements are encoded as equations, e.g., see Fig 2 where the states and regions of the state machine are encoded.

Listing 2. Encoding of states and regions

```
26 ***// States //
27 ops S0-1248987457 S1--1717774261 T--546184256 S2--453151192 : ->
28         ModState .
29 eq S0-1248987457 = root(R "R-346887101") : C "S0-1248987457" .
30 eq S1--1717774261 = root(R "R-346887101") : C "S0-1248987457" :
31         R "R1075236835" : C "S1--1717774261" .
32 eq T--546184256 = root(R "R-346887101") : C "S0-1248987457" :
33         R "R1075236835" : C "T--546184256" .
34 eq S2--453151192 = root(R "R-346887101") : C "S0-1248987457" :
35         R "R1075236835" : C "S2--453151192" .
36
37 ***// Regions //
38 ops R-346887101 R1075236835 : -> Region .
39 eq R-346887101 = root(R "R-346887101") .
40 eq R1075236835 = root(R "R-346887101") : C "S0-1248987457" :
41         R "R1075236835" .
42 ***// allVerts //
43 op allVerts : -> Verts .
44 eq allVerts = R-346887101, R1075236835 , S0-1248987457,
45         S1--1717774261, T--546184256, S2--453151192 .
```

Listing 3. Semantics State Machine interface.

```
107 op SMINT : -> StateMachine .
```

```
108    eq SMINT = (allVerts allDefs allSHDefs allDHDefs allEntryActions allExitActions allTrans) .
```

This information can then be accessed as described in Listing 20. Now the state machine can be simulated and model checked with commands shown in Listing 4 and Listing 5. Listing 5 shows how reachability of the final state can be checked.

Listing 4. Simulation of a state machine with Maude

```
1 maState stableC<STATEC> S0-1248987457, S1--1717774261, R1075236835,
2     R-346887101 <HISTC> empty <ENDCONF> (failure, done, reconnect) =>*
3     mastate such that isDone mastate = true .
```

Listing 5. Model checking of a state machine with Maude

```
1 reduce modelCheck(maState stableC<STATEC> S0-1248987457, S1--1717774261,
2   R-346887101, R1075236835 <HISTC> empty <ENDCONF> (failure, done, reconnect),
3         <> statesActive(T--546184256)) .
```

The numbers behind dash in the ids of the states and regions are used for automated parsing of results in the Eclipse integration implementation which we address next.

4 Eclipse Integration

The approach for simulating and model checking of state machines presented in this paper is implemented and integrated in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[2] framework. It is here used to automatically generate a state machines encoding from the graphical representations.

Generally speaking KIELER is a test-bed for enhancing the pragmatics, i. e., the user interaction, of model-based system design as described in our previous paper [20]. The KIELER framework is a set of open source Eclipse plug-ins that integrate with common Eclipse modeling projects, such as the Graphical Modeling Framework (GMF), the Textual Modeling Framework (TMF), and especially the Eclipse Modeling Framework (EMF) as the modeling backbone.

[2] http://www.informatik.uni-kiel.de/en/rtsys/kieler/, last visited: Jan 15, 2012

State machines are represented in our Maude syntax for the analysis, whereas they are represented graphically in KIELER and encoded conforming to the official UML2 meta model.

Simulating and model checking of state machines is fully integrated into the KIELER Eclipse based framework to abstract from the concrete syntax of a model checker. State machines can be defined by modeling them graphically using the most common Eclipse based UML2 state machine editor that is part of the Papyrus MDT project[3].

Utilizing model transformation techniques in Eclipse, a state machine can be transformed into any textual representation using a simulator's, e. g., Maude, or a model checker's, e. g., Maude or PROMELA, syntax. Here, we transform the official abstract UML2 syntax of state machines into the concrete syntax used by our Maude implementation.

An interface to a simulator and model checker completes the integration. Using generated identifiers for the elements, i. e., for states, transitions, events, and actions, the output of the model checker can then be mapped back to the corresponding elements in the editor and, e. g., be used for visualization. Fig. 2 shows an example where the active states are grey.

Listing 6. Events

```
1  fmod EVENT is
2    protecting STRING .
3    sort Event .
4    op ev:_ : String -> Event [ctor] .
5  endfm
```

Listing 7. States and regions as vertices

```
1  fmod VERTEXSYNTAX is
2    sorts Vert VertID .
3  endfm
4
5  view Vert from TRIV to VERTEXSYNTAX is
6    sort Elt to Vert .
7  endv
8
9  view VertID from TRIV to VERTEXSYNTAX is
10   sort Elt to VertID .
11 endv
12
13 fmod VERTSSYNTAX is
```

[3] http://www.eclipse.org/modeling/mdt/, last visited: Jan 15, 2012

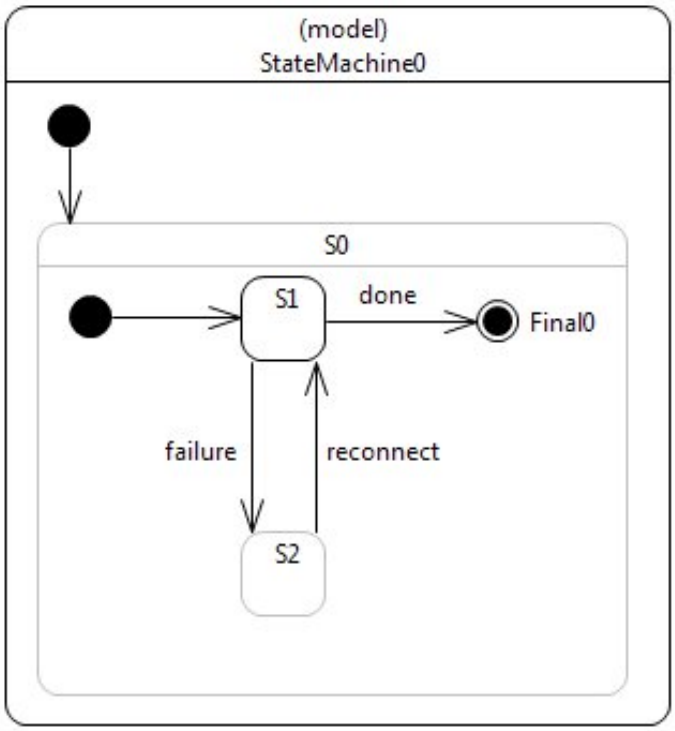

Figure 2. Visualization in KIELER where active states are grey

```
14  including VERTEXSYNTAX .
15  including STRING .
16  including SET{Vert} * (sort Set{Vert} to Verts, sort  NeSet{Vert} to NeVerts) .
17  including SET{VertID} * (sort Set{VertID} to VertIDs,
18          sort NeSet{VertID} to NeVertIDs) .
19
20  sorts RegionID StateID CompositeID FinalID HistDeepID
21    JoinID ForkID ChoiceID HistShallowID .
22  subsorts CompositeID FinalID HistDeepID JoinID
23    ForkID ChoiceID HistShallowID < StateID .
24  subsorts RegionID StateID < VertID .
25
26  sorts Region ModState Composite Final HistDeep Join Fork Choice HistShallow .
27  subsorts Composite Final HistDeep Join Fork Choice HistShallow < ModState .
28
29  subsort ModState Region < Vert .
30
31  op F_ : String -> FinalID [ctor] .
32  op C_ : String -> CompositeID [ctor] .
33  op R_ : String -> RegionID [ctor] .
34  op join_ : String -> JoinID [ctor] .
35  op fork_ : String -> ForkID [ctor] .
36  op choice_ : String -> ChoiceID [ctor] .
```

```
37    op H : -> HistShallowID [ctor] .
38    op H* : -> HistDeepID [ctor] .
39    op root_ : RegionID -> Region [ctor] .
40    op _:_ : Composite RegionID -> Region [ctor] .
41    op _:_ : Region CompositeID -> Composite [ctor] .
42    op _:_ : Region FinalID -> Final [ctor] .
43    op _:_ : Region HistShallowID -> HistShallow [ctor] .
44    op _:_ : Region HistDeepID -> HistDeep [ctor] .
45    op _:_ : Region JoinID -> Join [ctor] .
46    op _:_ : Region ForkID -> Fork [ctor] .
47    op _:_ : Region ChoiceID -> Choice [ctor] .
```

Listing 8. Actions are modeled as reference sets

```
1  fmod VARIABLE is
2    including STRING .
3    sort Variable .
4    op Var-_ : String -> Var [ctor] .
5  endfm
6
7  view Variable from TRIV to VARIABLE is
8    sort Elt to Variable .
9  endv
10
11 fmod ACTION is
12   including SET{Variable} * (sort Set{Variable} to Variables,
13         sort NeSet{Variable} to NeVariables) .
14   sorts Action .
15   op skip : -> Action .
16   op action_read_write : Variables Variables -> Action .
17 endfm
18
19 view Action from TRIV to ACTION is
20   sort Elt to Action .
21 endv
22
23 fmod ACTIONSSYNTAX is
24   including ACTION .
25   including LIST{Action} * (sort List{Action} to Actions,
26         sort NeList{Action} to NeActions) .
27
28   op (seq_) : Actions -> Action .
29   op (par_) : Actions -> Action .
30
31 endfm
```

Listing 10. Basic transitions and guards

```
1  fmod TRANSITIONSYNTAX is
2    including EVENTSET .
3    including VERTSSYNTAX .
4    including ACTIONSSYNTAX .
5
6    sorts Guard BasicTransition Fireset .
7
8    op basicTrans_______ : String Vert Guard Action Vert ->
9           BasicTransition [ctor]  .
33
```

Listing 9. Detecting interference

```
fmod ACTIONS is
  including ACTIONSSYNTAX .
  including VARIABLE .

  vars a a2 : Action .
  vars A A2 : Actions .
  var NeA : NeActions .
  vars Vars Vars1 : Variables .
  op refers : Actions -> Actions .
  eq refers (par A) = refers A .
  eq refers (seq A) = refers A .
  eq refers a NeA = (refers a) (refers A) .
  eq refers action Vars read Vars1 write = Vars Vars1 .

  op interferes_ : Actions -> Bool .
  eq interferes (seq A) = interferesSeq A .
  eq interferes (par A) = interferesPar A .
  eq interferes a = false [owise] .

  op interferesSeq_ : Actions -> Bool .
  eq interferesSeq (a A) = (interferes a) or (interferesSeq A) .
  eq interferesSeq nil = false .

  op interferesPar_ : Actions -> Bool .
  eq interferesPar (a A) = (interferes a) or
       (neIntersection ((refers a), (refers A))) or interferesPar A .
  eq interferesPar nil = false .
endfm
```

```
endfm

view BasicTransition from TRIV to TRANSITIONSYNTAX is
  sort Elt to BasicTransition .
endv
```

Listing 11. Compound transitions

```
fmod COMPOUNDTRANSITIONSYNTAX is
  including EVENTSET .
  including VERTSSYNTAX .
  including ACTIONSSYNTAX .
  including SET{BasicTransition} * (sort Set{BasicTransition} to BasicTransitions,
       sort NeSet{BasicTransition} to NeBasicTransitions) .

  sort Compoundtransition Menge .

  op joinTrans_____ : String Event BasicTransitions BasicTransition Region ->
       Compoundtransition .
  op forkTrans_____ : String Event BasicTransition BasicTransitions Region ->
       Compoundtransition .
  op choiceTrans_____ : String Event BasicTransition BasicTransitions Region ->
       Compoundtransition .
  op simpleTrans_____ : String Event BasicTransition Region ->
       Compoundtransition .
```

Listing 12. Properties of states

```
fmod STATE−ACTION is
  including VERTSSYNTAX .
  including ACTIONSSYNTAX .
  sort EntryAction ExitAction .
  op entryAc___ : ModState Action −> EntryAction .
  op exitAc___ : ModState Action −> ExitAction .
endfm
fmod STATE−DEFAULT is
  including VERTSSYNTAX .
  sort Default .
  op _default_ : Region ModState −> Default .
endfm
```

Listing 13. state machines

```
fmod STATEMACHINESYNTAX is
  including SET{ExitAction} * (sort Set{ExitAction} to ExitActions,
        sort NeSet{ExitAction} to NeExitActions) .

  sorts StateMachine .
  op __________ : Verts Defaults ShallowDefaults DeepDefaults
    EntryActions ExitActions Compoundtransitions
    −> StateMachine .

endfm
```

Listing 14. state machines states of execution

```
view Fireset from TRIV to COMPOUNDTRANSITIONSYNTAX is
  sort Elt to Fireset .
endv

fmod SEMANTICSSYNTAX is
  including STATEMACHINESYNTAX .
  including COMPOUNDTRANSITIONSSYNTAX .
  including SET{Fireset} * (sort Set{Fireset} to Firesets,
        sort NeSet{Fireset} to NeFiresets) .

  sorts LtsState .
  sort MState .

  op maState___ : Configuration EventSet −> MState .
  op fireset_ : Compoundtransitions −> Fireset .

  sort Configuration .

op stableC<STATEC>_<HISTC>_<ENDCONF> : Verts HistoryConfs −>
        Configuration .
op eventSelC<STATEC>_<HISTC>_<EVENT>_<ENDCONF> : Verts
        HistoryConfs Event −> Configuration .
op fireC<STATEC>_<HISTC>_<TRANSSETS>_<ENDCONF> : Verts
        HistoryConfs Sets −>
        Configuration .
op unstableC<STATEC>_<HISTC>_<TRANSITIONS>_<ACTIONS>_<ENDCONF> :
        Verts HistoryConfs Compoundtransitions Action −> Configuration .
op leaveC<STATEC>_<HISTC>_<TRANSITIONS>_<NEXTTR>_
        <INCOMINGBASICTR>_<OUTGOINGBASICTR>_<LEAVESTATES>_
```

```
                <ACTIONS>_<ENDCONF> :
        Verts HistoryConfs Compoundtransitions Compoundtransition
                BasicTransitions BasicTransitions Verts Action -> Configuration .
op enterC<STATEC>_<HISTC>_<TRANSITIONS>_<NEXTTR>_<ENTERSTATES>_
  <ACTIONS>_<ENDCONF> :
        Verts HistoryConfs Compoundtransitions Compoundtransition
                Verts Action -> Configuration .
op execTransC<STATEC>_<HISTC>_<TRANSITIONS>_<NEXTTR>_
        <INCOMINGBASICTR>_<OUTGOINGBASICTR>_<ACTIONS>_<ENDCONF> :
        Verts HistoryConfs Compoundtransitions Compoundtransition
                BasicTransitions BasicTransitions Action -> Configuration .
op execChoiceOut<STATEC>_<HISTC>_<TRANSITIONS>_<NEXTTR>_
        <INCOMINGBASICTR>_<OUTGOINGBASICTR>_<ACTIONS>_<ENDCONF> :
        Verts HistoryConfs Compoundtransitions Compoundtransition
                BasicTransitions BasicTransitions Action -> Configuration .
op doneC<STATEC>_<HISTC>_<ACTIONS>_<ENDCONF> : Verts HistoryConfs
        Action -> Configuration .

  op readyMachine_ : MState -> Bool .
  eq readyMachine (maState (doneC<STATEC> verts <HISTC> hc
        <ENDCONF>) evs) = true .
  eq readyMachine mstate = false [owise] .
```

Listing 15. Event dispatch and selection of fireing transitions

```
crl : maState (stableC<STATEC> V <HISTC> HC <ENDCONF>) (e2, ES) =>
  maState (eventSelC<STATEC> V <HISTC> HC <EVENT> e2 <ENDCONF>)
        (ES, e2) if not((enabled V e2) == empty) .
crl : maState (eventSelC<STATEC> V <HISTC> HC <EVENT> e2 <ENDCONF>)
        (ES) =>
  maState (fireC<STATEC> V <HISTC> HC <TRANSSETS>
    (fireableSets
        (remLowPrio (enabled V e2)))
    <ENDCONF>) (ES) .
rl : maState (fireC<STATEC> V <HISTC> HC <TRANSSETS>(mm(NeT), MN)
        <ENDCONF>) ES =>
  maState (unstableC<STATEC> V <HISTC> HC <TRANSITIONS> NeT
        <ACTIONS> nil <ENDCONF>) ES .
rl : maState (unstableC<STATEC> V <HISTC> HC <TRANSITIONS> (t, T)
        <ACTIONS> A <ENDCONF>) ES =>
  maState (leaveC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
    <INCOMINGBASICTR> (inTrans t) <OUTGOINGBASICTR> (outTrans t)
     <LEAVESTATES> (leave V t) <ACTIONS> A <ENDCONF>) ES .
rl : maState (unstableC<STATEC> V <HISTC> HC <TRANSITIONS> t
        <ACTIONS> A <ENDCONF>) ES =>
  maState (leaveC<STATEC> V <HISTC> HC <TRANSITIONS> empty <NEXTTR> t
    <INCOMINGBASICTR> (inTrans t) <OUTGOINGBASICTR> (outTrans t)
     <LEAVESTATES> (leave V t) <ACTIONS> A <ENDCONF>) ES .
```

Listing 16. Leaving states and regions for a compound transition

```
crl : maState (leaveC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
 <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1
  <LEAVESTATES> (v, leaveStates) <ACTIONS> A <ENDCONF>) ES =>
  maState (leaveC<STATEC> (V \ v) <HISTC> HC <TRANSITIONS> T
   <NEXTTR> t <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1
    <LEAVESTATES> leaveStates <ACTIONS> A <ENDCONF>) ES
   if (((intersection (leaveStates, (getSubVerts v))) == empty)
```

```
131           and not(typeRegion v)) .
135 crl : maState (leaveC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
136  <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1
137  <LEAVESTATES> (v, leaveStates) <ACTIONS> A <ENDCONF>) ES =>
138  maState (leaveC<STATEC> (V \ v) <HISTC> HC <TRANSITIONS> T <NEXTTR> t
139   <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1
140    <LEAVESTATES> leaveStates <ACTIONS> A <ENDCONF>) ES
141   if ((intersection (leaveStates, (getSubVerts v)) == empty)
142    and (typeRegion v)) .
146 rl : maState (leaveC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
147  <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1
148   <LEAVESTATES> empty <ACTIONS> (seq A) <ENDCONF>) ES =>
149   maState (execTransC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
150    <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1
151     <ACTIONS> (seq A (gatherExitAc (region t) (leave V t)))<ENDCONF>) .
```

Listing 17. Basic transitions execution

```
155 crl : maState (execTransC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
156  <INCOMINGBASICTR> empty <OUTGOINGBASICTR> sT1 <ACTIONS> A <ENDCONF>) ES =>
157   maState (execChoiceOut<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
158    <INCOMINGBASICTR> empty <OUTGOINGBASICTR> sT1 <ACTIONS> A <ENDCONF>) ES
159    if (isChoice t) .
163 rl : maState (execChoiceOut<STATEC> V <HISTC> HC
164  <TRANSITIONS> T <NEXTTR> t <INCOMINGBASICTR> empty <OUTGOINGBASICTR> sT1
165   <ACTIONS> A <ENDCONF>) ES =>
166   maState (enterC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
167   <ENTERSTATES> (enterH HC (chooseChoice t V)) <ACTIONS> A <ENDCONF>) ES .
171 crl : maState (execTransC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
172  <INCOMINGBASICTR> empty <OUTGOINGBASICTR> (st, sT1) <ACTIONS> A <ENDCONF>) ES =>
173   maState (execTransC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
174    <INCOMINGBASICTR> empty <OUTGOINGBASICTR> sT1 <ACTIONS> A <ENDCONF>) ES
175    if not(isChoice t) .
179 rl : maState (execTransC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
180  <INCOMINGBASICTR> (st, sT) <OUTGOINGBASICTR> sT1 <ACTIONS> A <ENDCONF>) ES =>
181   maState (execTransC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
182    <INCOMINGBASICTR> sT <OUTGOINGBASICTR> sT1 <ACTIONS> A <ENDCONF>) ES .
186 rl : maState (execTransC<STATEC> V <HISTC> HC
187  <TRANSITIONS> T <NEXTTR> t <INCOMINGBASICTR> empty <OUTGOINGBASICTR> empty
188   <ACTIONS> (seq A) <ENDCONF>) ES =>
189   maState (enterC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
190   <ENTERSTATES> (enterH HC t) <ACTIONS> (seq A (gatherActions t)) <ENDCONF>) ES .
```

Listing 18. Entering states and regions for a compound transition

```
194 crl : maState (enterC<STATEC> V <HISTC> HC
195  <TRANSITIONS> T <NEXTTR> t <ENTERSTATES> (v, enterStates)
196   <ACTIONS> A <ENDCONF>) ES =>
197   maState (enterC<STATEC> (V, v) <HISTC> HC <TRANSITIONS> T <NEXTTR> t
198    <ENTERSTATES> enterStates <ACTIONS> A <ENDCONF>) ES
199   if (((intersection (enterStates, (start v))) == empty) and not(typeRegion v)) .
203 crl : maState (enterC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
204  <ENTERSTATES> (v, enterStates) <ACTIONS> A <ENDCONF>) ES =>
205   maState (enterC<STATEC> (V, v) <HISTC> HC <TRANSITIONS> T <NEXTTR> t
206    <ENTERSTATES> enterStates <ACTIONS> A <ENDCONF>) ES
207   if ((intersection (enterStates, (start v)) == empty) and (typeRegion v)) .
211 rl : maState (enterC<STATEC> V <HISTC> HC <TRANSITIONS> T <NEXTTR> t
212  <ENTERSTATES> empty <ACTIONS> (seq A) <ENDCONF>) ES =>
```

```
213     maState (unstableC<STATEC> V <HISTC> HC <TRANSITIONS> T
214             <ACTIONS> (seq A (gatherEntryAc (region t) V)) <ENDCONF>) ES .
```

Listing 19. Finishing the run-to-completion step.

```
218  rl : maState (unstableC<STATEC> V <HISTC> HC <TRANSITIONS> empty
219    <ACTIONS> A <ENDCONF>) ES =>
220     maState (doneC<STATEC> V <HISTC> (succHC V HC) <ENDCONF>) ES .
224  rl : maState (doneC<STATEC> V <HISTC> HC <ENDCONF>) ES =>
225     maState (stableC<STATEC> V <HISTC> HC <ENDCONF>) ES .
```

Listing 20. Use of the state machine interface

```
49     op getEntryAc___ : EntryActions Vert -> Action [format (g o d d)] .
50     eq getEntryAc ((entryAc v a), ENA) v = a .
51     eq getEntryAc ENA v = nil [owise] .
52
53     op getSMEntryAc_ : Vert -> EntryActions .
54     eq getSMEntryAc v = getEntryAc ($getSMEntryAc SMINT) v .
55
56     op $getSMEntryAc_ : StateMachine -> EntryActions .
57     eq $getSMEntryAc (V D SH DH ENA EXA T) = ENA .
```

References

1. OMG: UML 2.2 Superstructure Specification. (February 2009) `http://www.omg.org/spec/UML/2.2/Superstructure/`.
2. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (1987) 231–274
3. Harel, D., Gery, E.: Executable object modeling with Statecharts. Computer **30**(7) (1997) 31–42
4. Damm, W., Josko, B., Hungar, H., Pnueli, A.: A compositional real-time semantics of statemate designs. In de Roever, W.P., Langmaack, H., Pnueli, A., eds.: COMPOS. Volume 1536 of LNCS., Springer (1997) 186–238
5. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in Promela/SPIN. In: WIFT, IEEE Computer Society Press (1998) 90–101
6. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML Statechart diagrams. In Ciancarini, P., Gorrieri, R., eds.: FMOODS, Kluwer (1999) 331–347
7. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. **23**(3) (2001) 273–303
8. Eshuis, R., Jansen, D.N., Wieringa, R.: Requirements-level semantics and model checking of object-oriented statecharts. Requirements Engineering **7**(4) (December 2002) 243–263
9. Hooman, J., van der Zwaag, M.: A semantics of communicating reactive objects with timing. In Graf, S., Haugen, O., Ober, I., Selic, B., eds.: SVERTS. Verimag technical report 2003/10/22, Verimag (2003) Available online at `http://www-verimag.imag.fr/EVENTS/2003/SVERTS/`.
10. Kyas, M., Fecher, H., de Boer, F.S., van der Zwaag, M., Hooman, J., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. In: SFEDL. ENTCS, Elsevier (2004)
11. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of UML state machines. In Davies, J., Schulte, W., Barnett, M., eds.: ICFEM. Volume 3308 of LNCS., Springer (2004) 434–448
12. Lam, V.S.W., Padget, J.A.: Symbolic model checking of UML statechart diagrams with an integrated approach. In: ECBS, IEEE Computer Society Press (2004) 337–347

13. Mrowka, R., Szmuc, T.: UML statecharts compositional semantics in LOTOS. In: ISPDC, IEEE Computer Society Press (2008) 459–463
14. Dubrovin, J., Junttila, T.A.: Symbolic model checking of hierarchical UML state machines. In Billington, J., Duan, Z., Koutny, M., eds.: ACSD, IEEE Computer Society Press (2008) 108–117
15. Gagnon, P., Mokhati, F., Badri, M.: Applying model checking to concurrent UML models. Journal of Object Technology **7**(1) (2008)
16. Zhao, Y., Yang, Z., Xie, J.: Formal semantics of uml state diagram and automatic verification based on kripke structure. In: CCECE, IEEE (2009) 974–978
17. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.P.: 29 new unclarities in the semantics of UML 2.0 state machines. In Lau, K.K., Banach, R., eds.: ICFEM. Volume 3785 of LNCS., Springer (2005) 52–65
18. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96** (1992) 73–155
19. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285** (2002) 187–243
20. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). Volume 6394 of LNCS., Springer (October 2010) 196–210