iState: A Statechart Translator [*]

Emil Sekerinski and Rafik Zurob

McMaster University, Department of Computing and Software
Hamilton, Ontario, Canada, L8S 4K1
emil|zurobrs@cas.mcmaster.ca

Abstract. We describe formal steps in the design of *iState*, a tool for translating statecharts into programming languages. Currently *iState* generates code in either Pascal, Java, or the Abstract Machine Notation of the B method. The translation proceeds in several phases. The focus of this paper is the formal description of the intermediate representations, for which we use class diagrams together with their textual counterparts. We describe how the class diagrams are further refined. The notions of representable, normalized, and legal statecharts are introduced, where normalized statecharts appear as an intermediate representation and code is generated only for legal statecharts.

1 Introduction

Statecharts, conceived as a visual formalism for the design of reactive systems [3], extend finite state diagrams by *hierarchy*, *concurrency*, and *communication*. These three extensions allow the specification of complex reactive systems that would be impractical without them. Because of the appeal of the graphical notation, statecharts have gained popularity and are now part of object-oriented modeling techniques [4, 8, 9].

The concepts of hierarchy, concurrency and communication, while intuitive on their own, interact in intricate ways. As a result, various formal semantics of statecharts and statechart tools interpret their interaction in different ways or impose different constraints, e.g. [2, 5]. Our approach is to use a semantics of statecharts defined by a direct translation into a (nondeterministic) programming language, with the goal of the generated code being *comprehensible*: Having readable code allows us to get confidence in the translator and in the original statechart. Furthermore, we can use the translation scheme for explaining statecharts and illustrate this by the translator, rather than having to use a "third domain" for the definition. Finally, in order to allow the generated code to be further analyzed, it must be understood in the first place.

Such a translation scheme from statecharts into the Abstract Machine Notation of the B method [1], an extension of Dijkstra's guarded commands, was presented in [10]. The use of AMN is motivated by three aspects: first, AMN supports nondeterminism, allowing nondeterministic statecharts to be translated to nondeterministic AMN machines (that can be further refined into executable machines). This way, the nondeterminism is being preserved rather than eliminated by the translation, as other tools typically do. Secondly, AMN supports the *parallel* or *independent* composition of statements, which allows a simple translation of concurrent states. Thirdly, the B method allows invariants to be stated and checked. Invariants can express safety conditions and checking the invariant will then ensure that every event is going to preserve all safety conditions.

[*] To appear in proceedings of UML 2001.

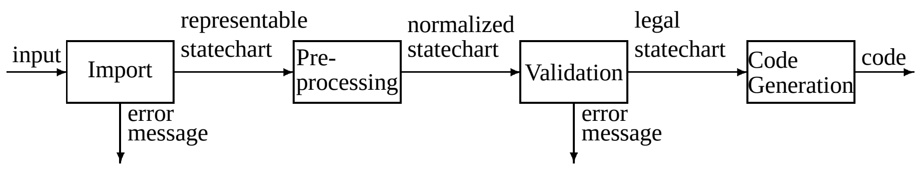


Fig. 1. Phases and intermediate representations of *iState*.

Compared to translation schemes used by other tools, ours can be characterized as *event-centric* rather than *state-centric*, as the main structure of the code is that of events. The scheme is suitable for those kind of reactive systems where events are processed quickly enough so that no queuing of events is necessary and where blocking of events is undesirable. To our experience so far, the resulting code is not only comprehensible, but compact and efficient as well.

The tool we built for implementing this translation scheme, *iState*, operates in four phases as shown in Figure 1. In this paper we focus on formally describing the intermediate representations, namely the *representable*, *normalized*, and *legal* statecharts. We refer to [10] for an illustration of the translation scheme and to [11] for details on the translation algorithms and for larger examples.

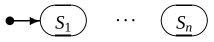
The input to *iState* is a textual representation of statecharts that is defined by a grammar. This representation can also be visualized through a LATEX package. However, the *Strategy* pattern has been used for allowing other importers to be added, for example a graphical front end. Currently *iState* generates code in AMN, Pascal, and Java. The code generator uses an intermediate representation (which is similar to AMN) so that other languages can be added as needed. All complete statecharts in this paper have been processed and drawn by *iState* and its accompanying LATEX package.

Section 2 presents briefly all statechart elements and the event-centric translation scheme. This prepares for the formalization of statecharts by class diagrams, together with their textual counterpart, in Section 3. This section illustrates a general way of translating class diagrams into a textual form. Normalized and flawed statecharts are characterized in Section 4 and legal statecharts are characterized in Section 5. The statechart model is refined by eliminating associations and basic algorithms on statecharts are discussed in Section 6. We give an example in Section 7 and conclude with a discussion in Section 8.

2 An Event-Centric Translation Scheme

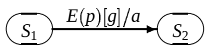
We give the translation scheme of statecharts to AMN first for plain state diagrams, then for hierarchy, concurrency, and communication. Programs in AMN, called *machines*, consists of a section declaring sets (types), a section declaring variables, an invariant section, an initialization section, and a section with operations. The initialization has to establish the invariant and all operations have to preserve it.

State diagrams consists of a finite number of *states*, *transitions* between states, and an *initial state*. Its state is represented in AMN by a variable of an enumerated set type.



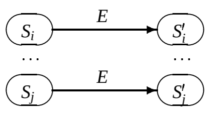
SETS $S = \{S_1, \dots, S_n\}$
VARIABLES s
INVARIANT $s \in S$
INITIALISATION $s := S_1$

Upon an *event*, a state machine can make a transition from one state to another. Events may be *generated* by the environment. Transition arrows are labeled with an event and optionally labeled with a *parameter*, a *guard*, or an *action*. Events are translated to operations that may be called from the environment. The guard is an expression over the global variables and the action is a statement over the global variables.



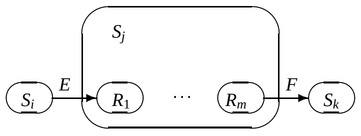
OPERATIONS
$E(p)$ $\widehat{=}$ **IF** $s = S_1 \wedge g$ **THEN** $s := S_2 \parallel a$ **END**

The operator $P||Q$ stand for the parallel composition of statements P and Q: only the variables assigned by P and Q must be distinct. For brevity, we omit parameters, guards, and actions from now on. In general, several transitions can be labeled with the same event. Let $S_i, \dots, S_j$ and $S'_i, \dots, S'_j$ be all (not necessarily distinct) elements of S. If some of the *source states* of the transitions below coincide, several of the conditions in the select statement may be true, leading to a nondeterministic choice.



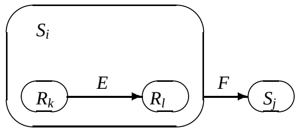
```
OPERATIONS
  E  ≙  SELECT s = S_i THEN s := S'_i
        ...
        WHEN s = S_j THEN s := S'_j
        ELSE skip
        END
```

Hierarchy. States can have two kinds of *substates* or *children*, *XOR states* and *AND states*. If a state has XOR children, then whenever the statechart is in that state, it is also in exactly one of its child states. We represent child states by an extra variable for each parent state. This generalizes to children having further offsprings. Transitions can break this hierarchy.



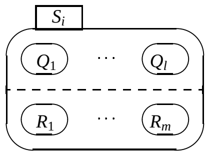
SETS $S = \{S_1, \dots, S_n\}; R = \{R_1, \dots, R_m\}$
VARIABLES s, r
INVARIANT $s \in S \wedge r \in R$
OPERATIONS
E $\widehat{=}$ **IF** $s = S_i$ **THEN** $s := S_j \parallel r := R_1$ **END**
F $\widehat{=}$ **IF** $s = S_j \wedge r = R_m$ **THEN** $s := S_k$ **END**

Transitions between child states are only taken if the statechart is in all their parent states. Transitions leaving a parent state also leave all the child states.



OPERATIONS
E $\widehat{=}$ **IF** $s = S_i \wedge r = R_k$ **THEN** $r := R_l$ **END**
F $\widehat{=}$ **IF** $s = S_i$ **THEN** $s := S_j$ **END**

Concurrency. If a state has AND children, then if the statechart is in that state, it is also in all its child states. AND states always have XOR children. AND states or *concurrent states* are separated by a dashed line in statecharts.



SETS $S = \{S_1, \dots, S_n\}; Q = \{Q_1, \dots, Q_l\}; R = \{R_1, \dots, R_m\}$
VARIABLES s, q, r
INVARIANT $s \in S \wedge q \in Q \wedge r \in R$

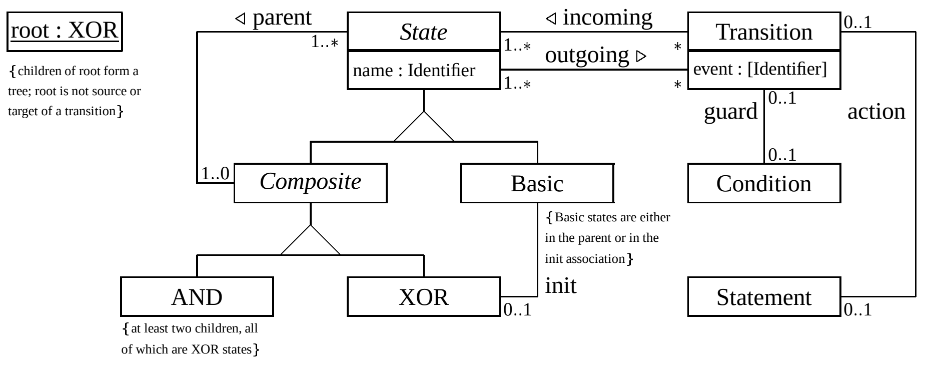
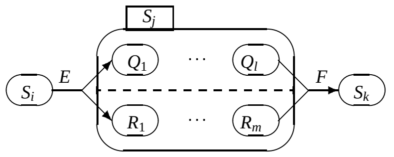

root : XOR
{children of root form a tree; root is not source or target of a transition}
◁ parent
1..∗
State
name : Identifier
◁ incoming
1..∗
outgoing ▷
∗
1..∗
∗
Transition
event : [Identifier]
0..1
guard
0..1
0..1
action
1..0
Composite
Basic
Condition
{Basic states are either in the parent or in the init association}
init
AND
XOR
0..1
Statement
0..1
{at least two children, all of which are XOR states}

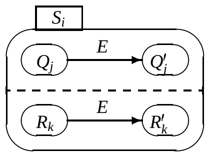
Fig. 2. Representable statecharts defined by a class diagram.

Transitions can *fork* to several AND states and can *join* from several AND states.



```
OPERATIONS
  $E \;\widehat{=}\;$ IF $s = S_i$ THEN $s := S_j \parallel q := Q_1 \parallel r := R_1$ END
  $F \;\widehat{=}\;$ IF $s = S_j \wedge q = Q_l \wedge r = R_m$ THEN $s := S_k$ END
```

Transitions in concurrent states labeled with the same event can be taken simultaneously. This is expressed by using parallel composition.



```
OPERATIONS
  $E \;\widehat{=}\;$ BEGIN
      IF $s = S_i \wedge q = Q_j$ THEN $q := Q'_j$ ELSE skip END $\parallel$
      IF $s = S_i \wedge r = R_k$ THEN $r := R'_k$ ELSE skip END
    END
```

Communication. Guards may contain *state tests* of the form **in** Q_i. These are translated to conditions of the form $q = Q_i$. Actions may contain statements that *broadcast* an event. These are translated by calling the the operation corresponding to that event. Section 7 presents an example.

3 The Statechart Model

We define *representable* statecharts in two ways, first graphically by the class diagram in Figure 2 and then in an equivalent textual form; this also illustrates a general way of translating class diagrams into a textual form. To start with, let us introduce *Object* to be the set of all objects. The class of states is a subset of objects. Every *State* object has an attribute *name* of type *Identifier*. We let $S \rightarrow T$ denote the set of all total functions from S to T.

$$State \subseteq Object$$
$$name \in State \rightarrow Identifier$$

States are either composite states or basic states, but no state can be both basic and composite. Furthermore, *State* is an abstract class, meaning that all objects of class *State* must belong to of one of its subclasses.

$$Composite \subseteq State \wedge Basic \subseteq State$$
$$Composite \cap Basic = \emptyset \wedge Composite \cup Basic = State$$

Likewise, composite states are either AND states or XOR states, but no state can be both an AND state and an XOR state. The class *Composite* is also abstract.

$$AND \subseteq Composite \wedge XOR \subseteq Composite$$
$$AND \cap XOR = \emptyset \wedge AND \cup XOR = Composite$$

Transitions are also objects. Each transition has an optional attribute *event* of type *Identifier*. Spontaneous transitions have no event name attached to them. We let $S \nrightarrow T$ denote the set of all partial functions from S to T.

$$Transition \subseteq Object$$
$$event \in Transition \nrightarrow Identifier$$

Conditions and statements are objects as well.

$$Condition \subseteq Object \wedge Statement \subseteq Object$$

The *guard* association relates every transition to at most one condition. Likewise, the *action* association relates every transition to at most one statement. We do not require that every condition and every statement relate to exactly one transition, as conditions and statements may appear as part of other conditions and statements, respectively. We let $S \rightarrowtail\!\!\!\!\!\!\nrightarrow T$ denote the set of partial, injective functions from S to T.

$$guard \in Transition \rightarrowtail\!\!\!\!\!\!\nrightarrow Condition$$
$$action \in Transition \rightarrowtail\!\!\!\!\!\!\nrightarrow Statement$$

The *outgoing* association relates every state to all the transitions leaving it. Any state may have zero or more transitions leaving it but every transition must have at least one state as origin. We let $S \leftrightarrow T$ denote the set of relations from S to T and $ran(R)$ the range of relation R.

$$outgoing \in State \leftrightarrow Transition$$
$$ran(outgoing) = Transition$$

The *incoming* association relates every transition to all the states to which it leads. Any state may have zero or more transitions leading to it but every transition must have at least one state as destination. We let $dom(R)$ denote the domain of relation R.

$$incoming \in Transition \leftrightarrow Transition$$
$$dom(outgoing) = Transition$$

The *init* association relates every XOR state to exactly one basic state, which we call the *init* state. This is the state from which all the initializing transitions are leaving, the destinations of which are the initial states. *Init* states do not appear graphically in the statecharts, or perhaps just as fat dots. They are added here for allowing initializing and proper transitions to be treated uniformly. We let $S \rightarrowtail T$ denote the set of all total, injective functions from S to T.

$$init \in XOR \rightarrowtail Basic$$

The *parent* association relates states to their parent states, which must be composite states. Every state has at most one parent and every composite state must have at least one child.

$$parent \in State \nrightarrow Composite$$
$$ran(parent) = Composite$$

We define the relation *children* to be the inverse of the function *parent*. We let R^{-1} denote the inverse of relation R.

$$children \;\widehat{=}\; parent^{-1}$$

Every AND state has at least two children and all children of AND states are XOR states. We let $R[S]$ denote the image of set S under relation R and $card(S)$ the cardinality of the set S.

$$\forall as \in AND \;.\; card(children[\{as\}]) \geq 2)$$
$$children[AND] \subseteq XOR$$

All basic states are either in the *init* or *parent* association.

$$ran(init) \cup dom(parent) = Basic$$

The root state is an XOR state. Every composite state is a descendant of root. We let R^* denote the transitive and reflexive closure of relation R.

$$root \in XOR$$
$$Composite \subseteq children^*[\{root\}]$$

The root state must not be the source or target of a transition.

$$root \notin dom(outgoing)$$
$$root \notin ran(incoming)$$

This completes the textual definition of statecharts. For brevity, we define the conditions of guards and the statements of actions only graphically by the class diagrams in Figures 3 and 4. Their textual counterpart is derived analogously.

4 Normalized and Flawed Statecharts

The purpose of defining normalized statecharts is that recognizing whether a statechart is flawed or illegal is simplified. Normalization is also a first step in translating to code.

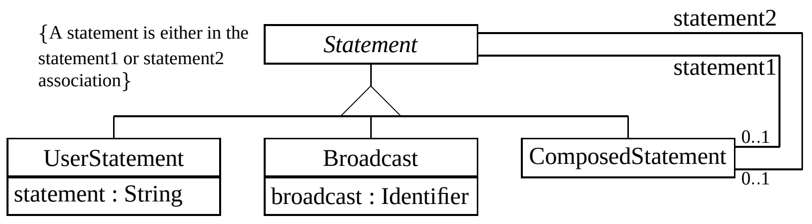


Fig. 3. Statements defined by a class diagram.

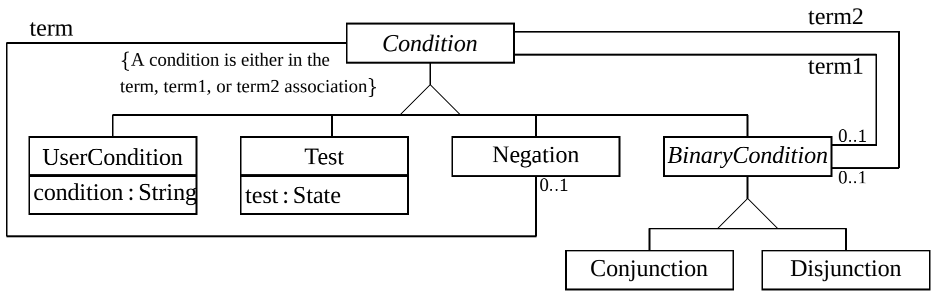


Fig. 4. Conditions defined by a class diagram.

Normalization adds those transition arrows to a representable statechart that are allowed to be left out. A statechart is normalized if two conditions hold, *targetsProper* and *transitionsComplete*, see Fig. 5 (a) and (b).

Targets of transitions must be either Basic or XOR states—if a target were an AND state, then that transition can be replaced by one that forks to all the XOR children of that AND state:

$$targetsProper \ \widehat{=}\ ran(incoming) \subseteq Basic \cup XOR$$

As all XOR states have an *init* state, this condition allows the set of all basic target states to be determined by transitively following the *init* association.

If an AND state is entered by a transition, then all XOR children must be entered by that transition as well. We define the *closest common ancestor* of a set *ss* of states to be that state that is an ancestor of each state in *ss* and all other common ancestors are also its ancestor, where each state is also its own ancestor. We let $x\ R\ y$ denote that the pair of x and y is in relation R. For any $ss \subseteq State$ we define $cca(ss)$ by:

$$c = cca(ss) \Leftrightarrow \forall s \in ss\ .\ (c\ parent^*\ s \wedge \forall a \in State\ .\ (a\ parent^*\ s \Rightarrow a\ parent^*\ c))$$

The closest common ancestor exists for any set of states that consists of non-*init* states. The *path* from state *s* to a set *ss* of children of *s* is the set of all states that are on the paths from *s* to a state of *ss*. Formally, $path(s,ss)$ is defined as those states that are descendants of *s* and ancestors of states in *s*, excluding *s* but including the states of *ss*.

$$paths(s,ss) \ \widehat{=}\ children^{+}[\{s\}] \cap parent^*[ss]$$

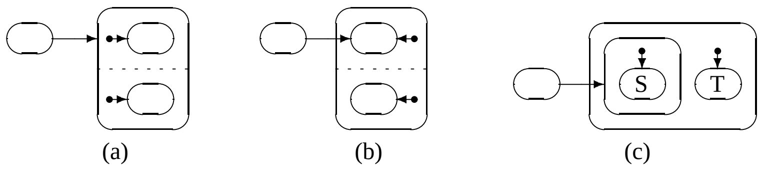


Fig. 5. (a) A statechart violating *targetsProper*. (b) A statechart violating *transitionsComplete*. (c) A statechart with two initial states *S* and *T*, both of which have reachable parents, but *S* is reachable and *T* is unreachable.

Following [5], the *scope* of a transition is the state closest to the root through which the transition passes.

$$scope(tr) \mathrel{\widehat{=}} cca(from(tr) \cup to(tr))$$

The states *entered* by a transition are all the states on the path from the scope of the transition to the targets of the transition. For symmetry, we define the states *exited* by a transition as all the states on the path from the scope of the transition to the sources of the transition.

$$entered(tr) \mathrel{\widehat{=}} path(scope(tr), to(tr))$$
$$exited(tr) \mathrel{\widehat{=}} path(scope(tr), from(tr))$$

This finally allows us to state the requirement that for all states entered by a transition, if the state is an AND state, then all children of that state must be entered by the transition as well. We let $R \rhd S$ denote the restriction of the range of relation R to set S, formally defined as $R; id(S)$.

$$transitionsComplete \mathrel{\widehat{=}} (entered \rhd AND); children \subseteq entered$$

Flawed statecharts are those that are legal but are likely incorrect. Flawed statecharts can appear while they are still being worked on, so by allowing them to be translated they can also be tested. However, *iState* issues a warning. More precisely, a statechart is *flawed* if a non-*init* state is unreachable, formally $\neg statesReachable$. Reachability is defined here solely based on the existence of transitions. A more involved definition taking the enabledness of transitions into account is possible, but quickly leads to conditions that are outside the scope of the tool.

A state is reachable if it is a target of a transition leaving a reachable state, or if it has a reachable descendant. The root state is assumed to be reachable. We define the relation *connected* to relate two states if there is a transition between them, including an *init* transition. We let $R; S$ denote the relational composition of relations R and S:

$$connected \mathrel{\widehat{=}} (outgoing; incoming) \cup init$$

Then $connected^*[\{root\}]$ is the set of all states explicitly reachable from root. States may be reachable even if there is no transition leading to them, but they have a reachable child, see Fig. 5 (c). We therefore consider the ancestors of reachable states as reachable as well. The set of reachable states must be equal to the set of non-*init* states.

$$statesReachable \mathrel{\widehat{=}} (connected^*; parent^*)[\{root\}] = ran(parent)$$

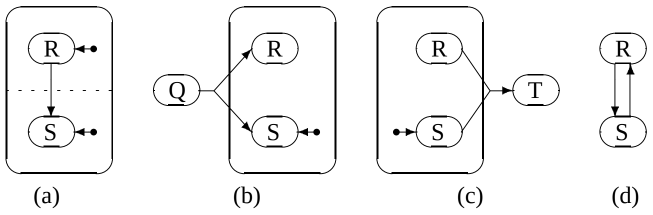


Fig. 6. Statecharts violating (a) *transitionsBetweenXORstates*, (b) *forksToANDstates*, (c) *joinsFromANDstates*, and (d) *spontaneousAcyclic*.

5 Legal Statecharts

Legal statecharts have to satisfy a number of conditions: *transitionsBetweenXORstates*, *forksToANDstates*, *joinsFromANDstates*, *spontaneousAcyclic*, *initTransitionsComplete*, *initToChildren*, *initNotTarget*, *initUnlabeled*, *noSpontaneousLeavingInitial*, *broadcastsAcyclic*, see Fig. 6 and 7. All legal statecharts can be translated to code, even if they are flawed.

No transition must cross a concurrency line. This would be the case if the scope of a transition is an AND state.

$$transitionsBetweenXORstates \mathrel{\widehat{=}} ran(scope) \subseteq XOR$$

Transitions must not fork to different children of an XOR state. More precisely, the closest common ancestor of any pair of targets of a transition must be an AND state.

$$\begin{aligned}&forksToANDstates \mathrel{\widehat{=}}\\ &\quad \forall tr \in Transition\ .\ (\forall s \in to(tr), t \in to(tr)\ .\ (s \neq t \Rightarrow cca(\{s,t\}) \in AND))\end{aligned}$$

Dually, transitions must not join from different children of an XOR state. More precisely, the closest common ancestor of any pair of sources of a transition must be an AND state.

$$\begin{aligned}&joinsFromANDstates \mathrel{\widehat{=}}\\ &\quad \forall tr \in Transition\ .\ (\forall s \in from(tr), t \in from(tr)\ .\ (s \neq t \Rightarrow cca(\{s,t\}) \in AND))\end{aligned}$$

A chain of spontaneous transitions must not contain a cycle. We define *spontaneousConnected* to be the relation between states with a spontaneous transition between them, including *init* transitions. We let $R \rhd S$ denote the restriction of the range of relation R to those elements not in S, formally defined as $R; id(ran(R) - S)$.

$$spontaneousConnected \mathrel{\widehat{=}} ((outgoing \rhd dom(event); incoming) \cup init$$

We let $id(S)$ denote the identity relation on set S.

$$spontaneousAcyclic \mathrel{\widehat{=}} id(Transition) \cap spontaneousConnected^{*} = \emptyset$$

Every XOR state has a basic *init* state and there must be a transition leaving it.

$$initTransitionsComplete \mathrel{\widehat{=}} ran(init) \subseteq dom(outgoing)$$

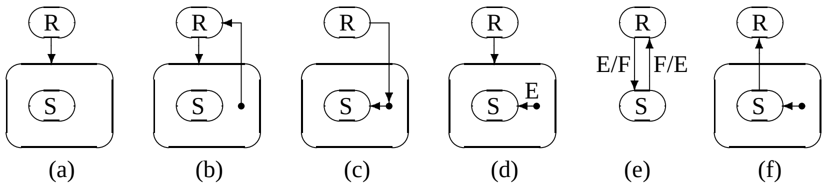


Fig. 7. Statecharts violating (a) *initTransitionsComplete*, (b) *initToChildren*, (c) *initNotTarget*, (d) *initUnlabeled*, (e) *broadcastsAcyclic*, and (f) *noSpontaneousLeavingInitial*

The transitions leaving the *init* state, the *init* transitions, must go to a child of the state to which the *init* state belongs.

$$initToChildren \mathrel{\widehat{=}} init ; connected \subseteq children$$

No *init* state must be a target of a transition.

$$initNotTarget \mathrel{\widehat{=}} ran(init) \cup ran(incoming) = \emptyset$$

We define *initTransitions* to be the set of all transitions leaving some *init* state.

$$initTransitions \mathrel{\widehat{=}} outgoing[ran(init)]$$

These transitions must not be labeled with an event or be associated with a guard or an action.

$$initUnlabeled \mathrel{\widehat{=}}$$
$$initTransitions \cap (dom(event) \cup dom(guard) \cup dom(action)) = \emptyset$$

Transitions leaving an initial state must not be spontaneous.

$$noSpontaneousLeavingInitial \mathrel{\widehat{=}} connected[initTransitions] \cap dom(event) = \emptyset$$

Finally, broadcasting must not be cyclic. For example, if on event *E* a transition broadcasts event *F* then no other transition must on event *F* broadcast event *E*. This generalizes to a cycle involving more than two transitions and it generalizes to the case when, say, on event *E* a transition does not itself broadcast event *F* but is followed by a chain of spontaneous transitions of which one broadcasts event *F*. We define *spontaneousSuccessor* to be the relation between transitions that relates a transition to all spontaneous transitions immediately following it.

$$spontaneousSuccessor \mathrel{\widehat{=}} incoming ; (outgoing \rhd dom(event))$$

The relation *transitionBroadcast* between *Transition* and *Identifier* relates each transition to all events that the action of the transition can broadcast, taking into account the way broadcast statement may be composed.

$$transitionBroadcast \mathrel{\widehat{=}} action ; (statement1 \cup statement2)^* ; broadcast$$

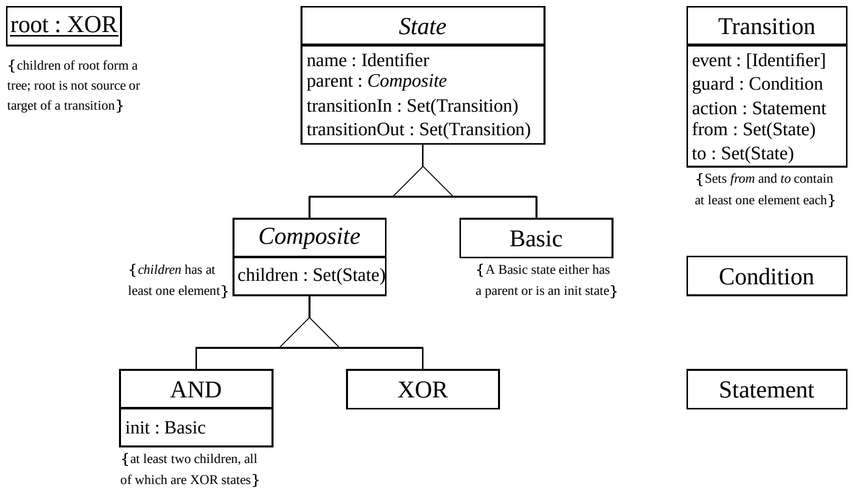

root : XOR
{children of root form a tree; root is not source or target of a transition}
State
name : Identifier
parent : Composite
transitionIn : Set(Transition)
transitionOut : Set(Transition)
Transition
event : [Identifier]
guard : Condition
action : Statement
from : Set(State)
to : Set(State)
{Sets from and to contain at least one element each}
Composite
children : Set(State)
{children has at least one element}
Basic
{A Basic state either has a parent or is an init state}
Condition
AND
init : Basic
{at least two children, all of which are XOR states}
XOR
Statement

Fig. 8. Refined class diagram of representable statecharts. Associations with a zero-or-one multiplicity are refined by attributes that can be *null*. Associations with higher multiplicity are replaced by *set*-valued attributes that must not contain *null*.

The relation *triggers* between identifiers representing events relates each event to all events possibly triggered by that event, taking into account that transitions may be followed by a spontaneous transition also broadcasting events.

$$triggers \ \widehat{=}\ event^{-1}; spontaneousSuccessor^{*}; transitionBroadcast$$

The condition that broadcasting must not be cyclic is expressed by stating that *triggers* is not cyclic.

$$broadcastsAcyclic \ \widehat{=}\ id(Identifier) \cap triggers^{*} = \emptyset$$

6 Refinement of the Statechart Model

We refine the model of representable statecharts by replacing associations with attributes. For brevity we give only the class diagram, see Fig. 8. On the refined model, we formulate algorithms for validating statecharts. In formulating the algorithms we use some object-oriented notation. Let x be an object reference, a an attribute, C a class.

$$\begin{array}{lcl} x.a & \widehat{=} & a(x) \\ x \ \mathbf{is}\ C & \widehat{=} & x \in C \end{array}$$

The procedure $cca(s,t)$ computes the closest common ancestor of non-*init* states s and t by first constructing paths from *root* to s and t, respectively, and then returning the last common state in those paths. The running time is bounded by the height of the tree, which is usually a small number.

```
**procedure** $cca(s, t : State) : State$
  **var** $l, m : seq(State)\ ;\ r : State$ ;
  **begin** $l := \langle s \rangle\ ;\ m := \langle t \rangle$ ;
    **while** $s \neq root$ **do begin** $s := s.parent\ ;\ l := l \circ \langle s \rangle$ **end** ;
    **while** $t \neq root$ **do begin** $t := t.parent\ ;\ m := m \circ \langle t \rangle \circ m$ **end** ;
    **repeat** $r := head(l)\ ;\ l := tail(l)\ ;\ m := tail(m)$
    **until** $l = \langle \rangle \vee m = \langle \rangle \vee head(l) \neq head(m)$ ;
    **return** $r$
  **end**
```

The procedure *scope*(tr) determines the scope of transition *tr* by repeatedly taking the closest common ancestor of the scope computed so far and an arbitrarily chosen element of the *to* and *from* set of that transition. It running time is proportional to the size of the *to* and *from* sets. We let $x :\in e$ denote the nondeterministic assignment of an element of the set e to x.

```
**procedure** $scope(tr : Transition) : State$
  **var** $r : State$ ;
  **begin** $\{tr.from \neq \emptyset \wedge tr.to \neq \emptyset\}$
    $r :\in tr.from \cup tr.to$ ;
    **for** $s \in tr.from \cup tr.to - \{r\}$ **do** $r := cca(r, s)$ ;
    **return** $r$
  **end**
```

The procedure *transitionsBetweenXORstates* checks for transitions crossing concurrency lines by checking whether the scope of each transition is and AND state. Its running time is proportional to the sum of the sizes of the *to* and *from* sets of each transition.

```
**procedure** $transitionsBetweenXORstates : boolean$
  **begin**
    **for** $tr \in Transition$ **do**
      **if** $scope(tr)$ **is** $AND$ **then return** $false$ ;
    **return** $true$
  **end**
```

The procedure *forksToANDstates* checks whether the targets of transitions go to different AND states. Its running time is proportional to the sum of the squares of the sizes of the *to* sets of all transitions.

```
**procedure** $forksToANDstates : boolean$
  **var** $ss : set(State)\ ;\ s : State$ ;
  **begin**
    **for** $tr \in Transition$ **do**
      **begin** $ss := tr.to$ ;
        **while** $ss \neq \emptyset$ **do**
```

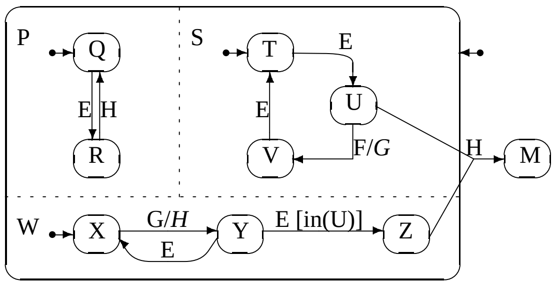


Fig. 9. An example with concurrent states, broadcasting, state tests, and nondeterminism.

```
            begin $s :\in ss$ ; $ss := ss - \{s\}$ ;
               for $t \in ss$ do if $cca(s,t)$ is $XOR$ then return $false$
            end
        end ;
    return $true$
end
```

An extended report contains the remaining algorithms [11]. In *iState* these are directly implemented in Java using the collection classes, some as part of the code generation.

7 Example

Below is the AMN code generated for the statechart of Figure 9. If the system is in substate *Y* of *W* and in substate *U* of *S*, then on event *E* the next substate of *W* could be either *X* of *Z*. This is reflected by the nondeterministic select statement in the B code. For Pascal and Java, a deterministic if-then-else statement is generated instead. A case statement is generated instead of a select statement if the selection is among different states, hence is deterministic. Pascal and B require that operations are defined before they are called. The code generator achieves this by a topological sort of the operations. For B, additionally auxiliary definitions have to be generated to avoid calls between operations. In any case, the resulting code will never contain circular calls.

```
MACHINE example
SETS
    ROOT = {noname0, M};
    S = {V, T, U};
    W = {Y, Z, X};
    P = {Q, R}
VARIABLES
    root, s, w, p
INVARIANT
    root ∈ ROOT ∧ s ∈ S ∧ w ∈ W ∧ p ∈ P
INITIALISATION
    root := noname0 || s := T || w := X || p := Q
DEFINITIONS
DEF_H ==
    IF root = noname0 THEN
        IF (w = Z) ∧ (s = U) THEN
```

```
            root := M
        ELSE
            IF p = R THEN
                p := Q
            END
        END
    END
;
DEF_G ==
    IF (w = X) ∧ (root = noname0) THEN
        DEF_H ||
        w := Y
    END

OPERATIONS
H = DEF_H
;
E =
    IF root = noname0 THEN
        CASE s OF
            EITHER V THEN s := T
            OR T THEN s := U
            OR U THEN skip
            END
        END ||
        SELECT w = Y THEN
            w := X
        WHEN (w = Y) ∧ (s = U) THEN
            w := Z
        ELSE skip
        END ||
        SELECT p = Q THEN
            p := R
        END
    END
;
G = DEF_G
;
F =
    IF (s = U) ∧ (root = noname0) THEN
        DEF_G ||
        s := V
    END
END
```

8 Discussion

We differ from UML statecharts in that we allow an *init*-state to have several outgoing transitions, like any other state, possibly leading to nondeterminism. On the other hand, we do require that the children of an AND state must be XOR states and not other AND states. Currently *iState* does not support enter and exit actions, internal actions which leave the system in the substate it was, history states for returning to the same substates from which a superstate was left, timeout events $timeout(E, d)$, which are generated d time units after event E is generated histories, overlapping states, sync states, and boolean expressions for events, e.g. $E \wedge F$. These remain the subject of ongoing research. We believe that most can be treated as straightforward extensions, possibly with the exception of the last one.

There has been some controversy whether the changes of a transition are visible in the current step or in the next step, see [5] for a recent discussion. Using parallel (independent) composition for updating the states of concurrent states we follow [5] by adopting the next step approach: updating concurrent states results in compositions of the from **SELECT** g **THEN** $x := e \dots \parallel$ **SELECT** h **THEN** $y := f \dots$, in which the initial values of x and y is taken for evaluating expressions e, f, g, h—a generalization of the multiple assignment $x, y := e, f$.

A number of tools support code generation from statecharts, including xjCharts, withClass, and Rhapsody. The code generated by Rhapsody differs from ours in being state-centric rather than event-centric: Methods corresponding to each state are generated. Events, represented by event objects, are passed to these methods. If a state has a transition using the passed event, it *consumes* the event by initiating the transition and stopping other states from seeing the event.

Another class of tools is based on a semantics of statecharts by (extended) state machines, for example [5–7]. The essence of these approaches is that statecharts are first compiled into a state transition table which is then interpreted by a universal algorithm. The states correspond to *configurations* of the statechart, i.e. they correspond to sets of states in which the statechart can be at any time. Our translation scheme avoids the introduction of configurations by having a separate variable for each state and it avoids the generation of state transition tables.

Acknowledgement. We are grateful to the reviewers for their careful comments.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
2. M. von der Beck. A comparison of statechart variants. In H. Langmaack, W.-P. deRoever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science Vol. 863, pages 128–148. Springer Verlag, 1994.
3. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
4. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1996.
5. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(5):293–333, 1996.
6. J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 – The Unified Modeling Language Beyond the Standard*, Lecture Notes in Computer Science Vol. 1723, Fort Collins, Colorado, 1999.
7. E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in Promela / Spin. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, 1998. IEEE Computer Society Press.
8. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddi, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
9. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
10. E. Sekerinski. Graphical design of reactive systems. In D. Bert, editor, *2nd International B Conference*, Lecture Notes in Computer Science Vol. 1393, Montpellier, France, 1998. Springer-Verlag.
11. E. Sekerinski and R. Zurob. From statecharts to code: A tool for the graphical design of reactive systems. Technical report, McMaster University, 2001.