On Formal Semantics of Statecharts as Supported by STATEMATE.

Erich Mikk , Yassine Lakhnech, Carsta Petersohn
Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Preußerstr. 1-9, D24105 Kiel, Germany
e-mail: {erm,yl,cp}@informatik.uni-kiel.de

Michael Siegel
Weizmann Institute of Science
Departement of Applied Mathematics and Computer Science
76100 Rehovot, Israel
email: mis@wisdom.weizmann.ac.il

Abstract

We formalize the rigorous but informal description of the semantics of statecharts given by Harel and Naamad in [3] which corresponds to the semantics underlying the commercial tool STATEMATE. We closely follow [3] to increase confidence that our semantics actually corresponds to their informal description.

In [3] the semantics is given by a detailed description of the so-called basic step algorithm. Based on a formalization of this basic step algorithm we associate to each statechart a transition system which defines its computations. This is the first step towards linking the language of statecharts as supported by STATEMATE with other automatic verification tools.

Our formalization uses Z notation rather than "standard mathematics". This allows to structure the definition of the formal semantics and to use tools like type-checkers.

1 Introduction

STATEMATE from iLogix[1] is a collection of tools for the specification, analysis, design, and documentation of complex reactive systems. The worldwide more than 2000 sold licenses are used with almost equal parity in the fields of aerospace, communications, electronics, and transportation. The graphical language of statecharts as proposed by David Harel [2] serves in STATEMATE for the description of control, data transformation, and timing issues of the system under development.

In the current paper we give a formal account of the semantics of statecharts as implemented in STATEMATE and described in [3].

Soon after its invention it turned out that the question of giving a semantics for statecharts was harder than expected. Since then more than 20 different semantics have been proposed, see, e.g., [16]. But none of these investigations deals with the STATEMATE variant of statecharts. This semantics is of particular practical interest since it underlines the tools of STATEMATE and is thus relevant for system designers in industry as well as developers of add-on tools for STATEMATE.

Statecharts as supported by STATEMATE comprises a rich graphical language complemented with programming language features like while loops, assignments, etc. As demonstrated in [3] the semantics of statecharts can be given in two steps. First, the graphical language is transformed into a sub-language; then the semantics is defined for this sub-language. We follow this approach and provide a formal semantics for this sub-language. In order to increase confidence that our formalization is indeed what Harel&Naamad intended and what is implemented in the

[1] The current URL of iLogix is http://www.ilogix.com/company/company.htm

STATEMATE tool we closely follow [3]. Harel&Naamad use several terms that have been introduced and defined in [4]; in the case of these terms we consequently follow the latter reference. The confidence that our formalization is indeed the STATEMATE semantics of statecharts can not be given by a formal proof because the reference point [3] is informal. However, for validating our formalization we give proof outlines for requirements from [3] that are not formalized as definitions in our semantics. Additionally, we validated our investigations by experiments with STATEMATE tools.

For sake of presentation we consider in this paper a sub-dialect of the language and focus on control issues. This includes defining when a transition is enabled, which states are exited, resp., entered when a transition is taken, which transitions can be taken in parallel, this is how to deal with priorities and non-determinism. We also consider inter-level transitions. We do not consider data transformations, history and timing issues. These aspects are orthogonal to the investigations of this paper in the sense that they do not interfere with basic concepts presented in the following sections. Integration of these issues is work in progress.

The semantics of statecharts is defined in [3] on the basis of the basic step algorithm. We formalize the basic step algorithm and associate a transition system to each statechart. This semantics serves as a link to transition system based verification tools. The implementation of a compiler from statecharts to Promela (the input language of the SPIN model checker [6]) is work in progress [12].

Our formalization uses Z notation rather than "standard mathematics". This allows to structure the definition of the formal semantics and to check the specification with the FUZZ type checker [14].

The rest of the paper is organized as follows. In Section 2 we give an introduction to statecharts. The syntax of statecharts is defined in Section 3, its semantics in Section 4. We end with related work and concluding remarks.

2 Introduction to Statecharts

The statecharts formalism is an extension of conventional finite state machines. Ordinary state transition diagrams are flat, unstructured, and inherently sequential, causing state explosion when modeling systems with parallel threads of control. In statecharts each state consists of a possibly empty hierarchy of statecharts modeling possibly concurrent, communicating state transition diagrams (see Fig. 1).

As in the formalism of Mealy machines, output (events in STATEMATE terminology) can be associated to a transition. Events are the means of communication between parts of a system. Communication is provided by the instantaneous *broadcast* mechanism. Transition labels in statecharts have the structure $Ev[Cond]/Act$ where Ev is a boolean combination of events, $Cond$ is a boolean condition, and Act is an action. All three parameters are optional. $Ev[Cond]$ is called the trigger part. If the transition source (or sources) is active, Ev occurs and $Cond$ is true then the transition is taken, resulting in the execution of Act, unless there is an enabled transition of higher priority or there is a nondeterministic choice. In the latter case one of the possible transitions is chosen nondeterministically. Events and conditions may refer to the current status of the system. The action part of a transition can generate new events and manipulate variables[2]. Timeout events and scheduling actions are available for the specification of timing aspects. We deal with a sub-dialect where transition labels are restricted as follows:

- only boolean combinations of predicates *in(st)* are allowed in expression $Cond$; informally, predicate *in(st)* is true iff the system currently resides in state *st*;
- the only effect of actions is the generation of events.

STATEMATE associates actions with entering and exiting of states. Our sub-dialect is restricted to generation of events in this part.

The operational semantics of statecharts as implemented in STATEMATE is a maximal parallelism semantics inspired by the synchrony hypothesis [1]. This semantics is implicitly defined by that part of the simulation tool that performs stepwise execution of statecharts. The heart of the simulation tool is the basic step algorithm that computes the next possible status ([3]) (or statuses in case of non-determinism) of the SUD.

Next, we illustrate the main ideas underlying this operational semantics by sketching a prefix of a computation of the statecharts given in Figure 1.

[2] For those who are familiar with STATEMATE: we do not consider the interplay between statecharts and activitycharts.

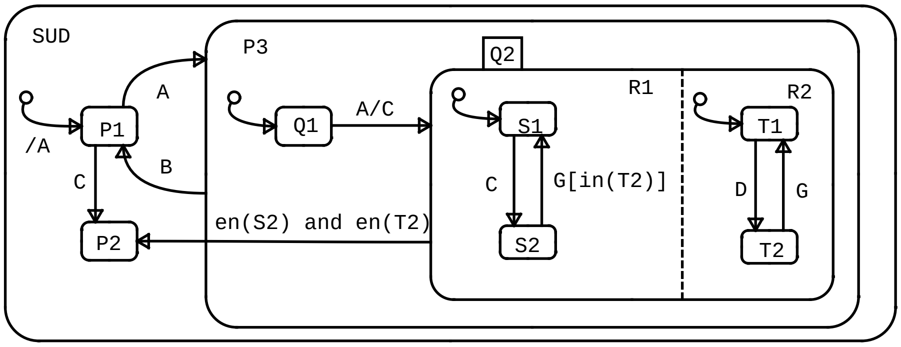

Figure 1: Running example.

State hierarchy: States form a state hierarchy: states contained within a state are called sub-states of this surrounding state; the surrounding state is higher in the hierarchy. In the example SUD is the highest state in the state hierarchy and P1, P2 and P3 are its sub-states. SUD is an ancestor of P1, P2 and P3.

Initial State: Initially, the system resides in a designated initial state which is not depicted in the statechart. In accordance with the STATEMATE tool this state is called INITIAL-SUD.

Default Connector: Default connectors are depicted as transitions emerging from small circles. The first step of the statechart SUD consists in entering state P1, as indicated by the default connector; since this default connector is labeled /A, the event A is generated as result of this transition.

Status and Active States: Computations are sequences of statuses. A status consists of two components: a set of states in which the computation currently resides (also called "active states") and a set of currently present events. At the moment, in our example SUD and P1 are active. Below, we generally mention only subsets of the set of active states; implicitly it is understood that all ancestor states of an active state are also active. The set of present events is $\{A\}$. Events provided by the environment are recorded in the event set of the status.

OR-state: State SUD is a so-called OR-state consisting of three sub-states P1, P2, and P3. Sub-states of an OR-state are exclusive: at any moment where SUD is active exactly one of its sub-states is active, too.

Basic State: P1 is a so-called basic state. Basic states does not have sub-states.

External Stimuli: A virtual environment generates events which are sensed by the system. Assuming that P1 has just been entered (so, event A has been generated) and the environment additionally provides event C a situation of non-determinism occurs.

Non-Determinism: Both transitions from P1 to P2 and from P1 to P3 are enabled since the events A and C are present. One of the transitions can be chosen to extend the current execution prefix.

Conflicting Transitions: Both transitions originate from P1 so they are in conflict, meaning that they can not be performed in the same step.

Duration of Events: Assume we have chosen the transition from P1 to P3 labeled by A. According to the default connector, state Q1 becomes active. Now, the transition from Q1 to Q2 is *not* enabled, unless the environment generates another A event. The A event which triggered the transition from P1 to P3 is no longer present. Events are only available in the step directly succeeding their generation.

AND-state: Currently Q1 is active. Assuming that the environment provides event A, state Q2 is entered and event C is generated. Q2 is a so-called AND-state consisting of the two parallel sub-states R1, R2. When entering Q2 both states R1 and R2 are entered simultaneously, so the states S1 and T1 become active. The fact that S1 (resp. T1) becomes active and not S2 (resp. T2) is implied by the fact that the default connector points to S1 (resp. T1). Note that only OR-states and basic states occupy space whereas AND-states are given only by their borderline.

Maximal Parallelism: Assume that the environment provides D directly after entering state Q2. Then, both transitions from S1 to S2 and from T1 to T2 are taken *simultaneously*, resulting in S2 and T2 being active. In case the environment had not provided event D, only the transition from S1 to S2 would have been performed.

Implicitly Generated Events: Whenever a state *st* is entered/exited as the result of executing a transition, implicitly the corresponding events *entered(st)/exited(st)* (abbreviated *en(st)/ex(st)*) are generated. So, the simultaneous entering of state S2 and T2 enables the transition from Q2 to P2 labeled 'en(S2) and en(T2)' in the next step.

Inter-level Transitions: The transition from Q2 to P2 is a so-called inter-level transition; it crosses the borderlines of the state P3. If this transition is taken then P3 is deactivated as well as all other sub-states of P3 that were active before taking it.

Transition Priority: Consider the situation where S2 and T2 are active and the transition from Q2 to P2 is enabled. Even if the environment provides event G, which enables both transitions from S2 to S1 (note, condition in(T2) evaluates to true) and the transition from T2 to T1, the transition from Q2 to P2 is performed since it has higher priority (no non-determinism arises). Priority between transitions is determined by comparing the scopes of enabled transitions.

Scope: Paper [3]: "The scope of a transition is the lowest OR-state which is neither exited nor entered by the respective transition." The scope of transition Q2 to P2 is state SUD whereas the scope of the transitions from S2 to S1 is R1 and from T2 to T1 is R2. If more than one transition is enabled, priority is given to that transition whose scope is highest in the state hierarchy. Consequently, priority is given to transition Q2 to P2. If scopes of transitions are identical, a situation of non-determinism arises. Note that non-determinism arises if both the transition from P3 to P1 labeled by B and the transition from Q2 to P2 are enabled.

Broadcast communication: Assume S2 and T2 are active and have not been entered in the same step (thus transition Q2 to P2 is not enabled). In case the environment provides event G both transitions from S2 to S1 and from T2 to T1 are taken. So, multiple parallel states can react simultaneously to the same events, i.e. events are broadcasted no matter whether they are generated internally, i.e. by the system itself, or by the environment.

This completes the list of concepts underlying the forthcoming formalization of the statecharts semantics.

3 Syntax of Statecharts

A statechart is given by a finite hierarchy of states, an initial state and a set of transitions. We describe the state hierarchy as a tree of states. Nodes of the tree are typed by elements of the set $\{AND, OR, BASIC\}$. Transitions are labeled by a pair consisting of a trigger and an action.

3.1 Harel&Naamad's language for the semantics definition

The essential difference between the graphical syntax of statecharts and the language that Harel&Naamad use for the semantics definition is in transitions. While in the graphical language as supported by STATEMATE a transition may consist of several components called transition segments, each of which may carry a label, the language used by Harel&Naamad uses the notion of full compound transitions (full CT's). This representation has several advantages. First, it leads to a concise description of when a full CT is enabled. Second, it allows for the definition of scope of the transition which is used to deal with nondeterminism and to associate priorities to transitions. The Section

"Compound transitions" in [3] discusses this in full detail. In the current paper we use the notion of full CT without defining formally how concrete transitions are transformed into full CT's. An example of full compound transitions is given in Section 3.4. More examples can be found in [3]. When working with full CT's the default connectors vanish from the original statecharts; their labels become part of full CT's labels. In the following we use the term "syntax of statecharts" for the language that Harel&Naamad use for the semantics definition.

3.2 States

State names and state types. We postulate a finite set of state names Σ and denote by $TYPE$ the types of states.

$$[\Sigma]$$
$$\exists\, n : \mathbb{N} \bullet \#\Sigma = n$$
$$TYPE ::= AND \mid OR \mid BASIC$$

Initial state. As described in Section 2 STATEMATE introduces for every statechart a fresh initial state $init$ and an initial transition emerging from $init$. We will see later how this initial transition is defined.

Harel&Naamad implicitly assume the existence of a further state which we will refer to as $root$. The purpose of root is to make the scope definition of [3] well-defined. In accordance with [3] $root$ is an OR-state with $init$ and the original statechart as direct sub-states.

Definition of state hierarchy. The state hierarchy $StateTree$ consists of the following components: the $root$ of the tree, the initial state $init$, the finite hierarchy function ρ which assigns a (possibly empty) set of direct sub-states to an ancestor state and the finite typing function ψ which assigns to each state its type. The schema $StateTree$ defines these objects.

$StateTree$

$$root : \Sigma$$
$$init : \Sigma$$
$$\rho : \Sigma \mathrel{+\!\!\!\!\!\!\twoheadrightarrow} \mathbb{F}\,\Sigma$$
$$\psi : \Sigma \mathrel{+\!\!\!\!\!\!\twoheadrightarrow} TYPE$$

$$\mathrm{dom}\,\rho \setminus \bigcup(\mathrm{ran}\,\rho) = \{root\} \wedge \psi(root) = OR$$
$$init \in \rho(root) \wedge \#(\rho(root)) = 2$$
$$\forall\, set : \mathbb{F}(\bigcup(\mathrm{ran}\,\rho)) \bullet (\exists\, el : set \bullet (\forall\, st : set \bullet el \notin \rho(st)))$$
$$\forall\, st : \bigcup(\mathrm{ran}\,\rho) \bullet (\exists_1\, anc : \mathrm{dom}\,\rho \bullet st \in \rho(anc))$$
$$\mathrm{dom}\,\rho = \mathrm{dom}\,\psi$$
$$\forall\, st : \mathrm{dom}\,\rho \bullet (\psi(st) = BASIC \Leftrightarrow \rho(st) = \emptyset)$$
$$\wedge\, (\psi(st) = AND \Rightarrow (\forall\, st : \rho(st) \bullet \psi(st) = OR))$$

Properties of root. State $root$ is the only state that has no ancestor. $root$ is an OR-state.

Properties of init. State $init$ is a sub-state of $root$. The root state has exactly two sub-states.

Tree properties. Necessary conditions for $root$ and ρ to form a tree are:

1. every non-root state must be accessible via ρ from $root$ (follows from the first and third predicate),
2. every non-root state has only one ancestor (fourth predicate),
3. ρ forms no cyclic paths (third and fourth predicate).

Type consistency. Every state has a type. Only basic states don't have sub-states. Direct sub-states of an AND-state must be OR-states.

3.3 Transitions

The main purpose of a full CT is to determine the states to be entered when the transition is taken. We introduce full CT's in two steps. We first give a definition of transitions without referring to any state hierarchy. Then, in the next sub-section we give a well-definedness condition that allows to compose a state hierarchy and a set of transitions into a well-defined statechart. All transitions in the resulting statechart are full CT's.

We define a finite set of primitive events.

$$[EV]$$
$$\exists\, n : \mathbb{N} \bullet \#EV = n$$

Event expressions are propositions over event names and constant $TRUE_E$ (which denotes the case where event e is omitted in a transition label).

$$EE ::= TRUE_E \mid B\langle\!\langle EV \rangle\!\rangle \mid NOT_E\langle\!\langle EE \rangle\!\rangle \mid AND_E\langle\!\langle EE \times EE \rangle\!\rangle \mid OR_E\langle\!\langle EE \times EE \rangle\!\rangle$$

Remark: The reader might find the function B strange; it performs a type conversion as required by Z.

Conditions are propositions over state references and constant $TRUE_C$ (which denotes the case where condition c is omitted in the transition label).

$$C ::= TRUE_C \mid In\langle\!\langle \Sigma \rangle\!\rangle \mid NOT_C\langle\!\langle C \rangle\!\rangle \mid AND_C\langle\!\langle C \times C \rangle\!\rangle \mid OR_C\langle\!\langle C \times C \rangle\!\rangle$$

As mentioned earlier we restrict the action part to the generation of events only. The schema $LABEL$ defines the set of labels.

$LABEL$
$event\_expr : EE$
$condition : C$
$action : \mathbb{F}\, EV$

A transition leads from a nonempty set of states denoted by $source$ to a nonempty set of states denoted by $target$. Transitions are labeled. The schema TR defines the set of transitions.

TR
$source : \mathbb{F}_1\, \Sigma$
$target : \mathbb{F}_1\, \Sigma$
$label : LABEL$

3.4 Well-defined statecharts

Well-formedness condition. Now we can give a well-definedness condition that allows to compose a state hierarchy and a set of transitions into a well-defined statechart. Our approach is to use the term *configuration* ([3]) – which is actually a semantical object – to define appropriate consistency conditions.

As mentioned earlier a computation of a statechart is a sequence of statuses. One component of the status is the set of currently active states that forms a configuration. This term is defined in the current section. Then we introduce functions to reason about state hierarchy and we identify a relation between basic states, configurations and an ancestor state. This relation helps us to define the target states of a full CT. Furthermore we define the term *orthogonality* ([3]) which is used for the definition of possible source and target states of full CT's.

Configurations. Paper [3]: "A configuration is a maximal set of states that the system can be in simultaneously. Given a root state R, a configuration (relative to R) is a set of states C obeying the following rules:

- C contains R.

- If C contains a state A of type OR-state, it must also contain exactly one of A's sub-states.
- If C contains a state A of type AND-state, it must also contain all of A's sub-states.
- The only states that are in C are those that are required by the above rules."

The following definition is a direct interpretation of the rules above and defines when a set of states $conf$ forms a configuration w.r.t. state top in a state hierarchy given by $tree$.

$$
\begin{array}{|l}
configuration\ \_ : \mathbb{F}_1(\Sigma \times StateTree \times \mathbb{F}_1\ \Sigma) \\
\hline
\forall\ top : \Sigma;\ tree : StateTree;\ conf : \mathbb{F}_1\ \Sigma \bullet \\
\quad configuration(top, tree, conf) \Leftrightarrow \\
\quad\quad top \in conf \\
\quad\quad \wedge\ (\forall\ st : conf \bullet \\
\quad\quad\quad (tree.\psi(st) = OR \Rightarrow (\exists_1\ sub : tree.\rho(st) \bullet sub \in conf)) \\
\quad\quad\quad \wedge\ (tree.\psi(st) = AND \Rightarrow tree.\rho(st) \subseteq conf) \\
\quad\quad\quad \wedge\ (st = top \vee st \neq top \wedge (\exists\ anc : conf \bullet st \in tree.\rho(anc))))
\end{array}
$$

Sub-states and ancestors We define extensions of ρ: its reflexive, transitive closure ρ^* and its non-reflexive, transitive closure ρ^+. The ancestor relation is expressed by Anc and the strict ancestor relation by $SAnc$.

$$
\begin{array}{|l}
\rho^*, \rho^+ : \Sigma \times StateTree \twoheadrightarrow\!\!\!\!\!\!\!\!\!\!\;\;\;\;\;\;\; \mathbb{F}\ \Sigma \\
Anc\ \_, SAnc\ \_ : \mathbb{F}_1(\Sigma \times \mathbb{F}_1\ \Sigma \times StateTree) \\
\hline
\forall\ top : \Sigma;\ tree : StateTree \bullet \\
\quad \rho^*(top, tree) = \{top\} \cup \bigcup\{subs : tree.\rho(top) \bullet \rho^*(subs, tree)\} \\
\quad \wedge\ \rho^+(top, tree) = \rho^*(top, tree) \setminus \{top\} \\
\forall\ anc : \Sigma;\ sts : \mathbb{F}_1\ \Sigma;\ tree : StateTree \bullet \\
\quad (Anc(anc, sts, tree) \Leftrightarrow sts \subseteq \rho^*(anc, tree)) \\
\quad \wedge\ (SAnc(anc, sts, tree) \Leftrightarrow sts \subseteq \rho^+(anc, tree))
\end{array}
$$

Next we define two functions which have been introduced in [4]: the lowest common ancestor (lca) and lowest common OR-ancestor ($lcoa$).

$$
\begin{array}{|l}
lca, lcoa : \mathbb{F}_1\ \Sigma \times StateTree \twoheadrightarrow\!\!\!\!\!\!\!\!\;\;\;\;\; \Sigma \\
\hline
\forall\ sts : \mathbb{F}_1\ \Sigma;\ tree : StateTree;\ anc : \Sigma \bullet \\
\quad (lca(sts, tree) = anc \Leftrightarrow \\
\quad\quad Anc(anc, sts, tree) \\
\quad\quad \wedge\ (\forall\ st : \Sigma \bullet Anc(st, sts, tree) \Rightarrow Anc(st, \{anc\}, tree))) \\
\quad \wedge\ (lcoa(sts, tree) = anc \Leftrightarrow \\
\quad\quad SAnc(anc, sts, tree) \wedge tree.\psi(anc) = OR \\
\quad\quad \wedge\ (\forall\ st : \Sigma \bullet SAnc(st, sts, tree) \wedge tree.\psi(st) = OR \Rightarrow Anc(st, \{anc\}, tree)))
\end{array}
$$

As observed by Harel&Naamad configurations w.r.t. the root are uniquely determined by their basic states. For the more general notion of configuration that we address the same property holds as stated in the next lemma.

Lemma **1** *Let* conf *be a configuration w.r.t. an* OR*-state* anc *in a state hierarchy* tree $\in StateTree$. *Let* bsts $\subseteq$ conf *be the set of all basic states in* conf. *Then* anc *is the lowest common* OR*-ancestor of* bsts*; and* bsts *determines configuration* conf *w.r.t.* anc *uniquely:*

$$lcoa(bsts, tree) = anc\ \forall\ c : \mathbb{P}\ \Sigma \bullet configuration(anc, tree, c) \wedge bsts \subseteq c \Rightarrow c = conf$$

Orthogonal states. Orthogonality is a term introduced in [4] to characterize possible sets of source states of full CT's. Our definition is inspired by this reference. Two states are orthogonal in the state hierarchy $tree$ if their lowest common ancestor is an AND-state.

$$orth\ \_ : \mathbb{F}(\Sigma \times \Sigma \times StateTree)$$

$$\forall\, s1, s2 : \Sigma;\ tree : StateTree \bullet orth(s1, s2, tree) \Leftrightarrow tree.\psi(lca(\{s1, s2\}, tree)) = AND$$

The orthogonality relation describes pairs of states that can be simultaneously active.

Scope of a transition. Scope expresses the idea of "influence area" of a transition. Paper [3]: "The scope of a CT tr is the lowest OR-state in the hierarchy of states that is a proper common ancestor of all the sources and targets of tr."

$$scope : TR \times StateTree \twoheadrightarrow \Sigma$$

$$\forall\, tr : TR;\ tree : StateTree \bullet scope(tr, tree) = lcoa(tr.source \cup tr.target, tree)$$

Paper [3]: "Scope is the lowest state in which the system stays without exiting and reentering when taking the transition."

Well-formed statecharts. Here we are more precise about full CT's. Harel&Naamad state the main property of full CT: "A full CT with scope S always exits a legal configuration relative to one sub-state of S, and enters a legal configuration relative to potentially (but not necessarily) another sub-state of S." This remark has been the inspiration for our definition of full CT's. Another citation gives further insight how they look like: "... the source of the full CT contains states only, and its target contains basic states ... only. Every two states in the source and every two states in the target must be mutually orthogonal. The target set must be maximal: if it contains a descendant of a component of an AND-state, then it contains descendants of all of its other components too."

Orthogonality and maximality of target states can be expressed using the definition of $configuration$. This is formalized in the schema SC below.

The consistency between the root, the initial state and the set of transitions is as follows: $root$ is neither a target nor a source of any transition; there exists a transition which source is $\{init\}$; state $init$ is not a target of any of transition.

These requirements are formalized in the schema SC.

SC

$$\sigma : StateTree$$
$$\tau : \mathbb{F}_1\ TR$$

$$\begin{aligned}
&\forall\, tr : \tau \bullet (\exists\, Uenter : \sigma.\rho(scope(tr, \sigma)) \bullet \\
&\quad (\exists\, conf : \mathbb{F}_1(\rho^*(Uenter, \sigma)) \bullet configuration(Uenter, \sigma, conf) \\
&\qquad \wedge\ tr.target = \{st : conf \mid \sigma.\psi(st) = BASIC\})) \\
&\quad \wedge\ (\forall\, s1, s2 : tr.source \bullet s1 \neq s2 \Rightarrow orth(s1, s2, \sigma))
\end{aligned}$$

$$\sigma.root \notin \bigcup\{tr : \tau \bullet tr.source\} \cup \bigcup\{tr : \tau \bullet tr.target\}$$
$$\exists\, tr : \tau \bullet tr.source = \{\sigma.init\}$$
$$\sigma.init \notin \bigcup\{tr : \tau \bullet tr.target\}$$

This definition is well-defined due to Lemma 1. In Figure 2 we transform the statechart in Figure 1 into a statechart that satisfies SC.

4 Semantics of Statecharts

The semantics of statecharts is a set of computations. A computation is a sequence of statuses. In order to define the transition relation we formalize the basic step algorithm from [3]. Then, we associate to each statechart a transition system which describes its set of computations.

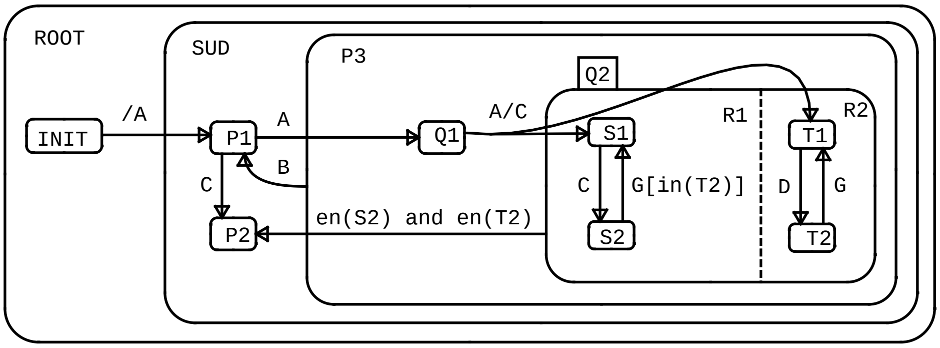

Figure 2: Running example: Well-defined statecharts

4.1 The basic step definition

In order to reason about transitions we investigate the following questions: when is a transition enabled, which states are exited, resp. entered when a transition is taken, when are transitions in conflict, how to deal with priority of transitions?

Enabledness of transitions. Paper [3]: "A CT tr is said to be enabled in a step if at the beginning of the step the system is in all states of its source set and its trigger is true." An event expression labeling a transition is evaluated w.r.t. a set of events.

$$event\_eval\ \_ : EE \leftrightarrow \mathbb{F}\,EV$$

$$\begin{aligned}
&\forall\, ex : EE;\ evs : \mathbb{F}\,EV \bullet event_eval(ex, evs) \Leftrightarrow \\
&\quad ex = TRUE_E \lor \\
&\quad ((\forall\, x : EV \bullet ex = B(x) \Rightarrow x \in evs) \\
&\quad \land (\forall\, x, y : EE \bullet (ex = NOT_E(x) \Rightarrow \neg\ event_eval(x, evs)) \\
&\qquad\qquad\qquad \land (ex = AND_E(x, y) \Rightarrow event_eval(x, evs) \land event_eval(y, evs)) \\
&\qquad\qquad\qquad \land (ex = OR_E(x, y) \Rightarrow event_eval(x, evs) \lor event_eval(y, evs))))
\end{aligned}$$

A condition labeling a transition is evaluated w.r.t. a set of states that form a configuration. The definition is similar to the definition of $eval\_eval$ and is omitted here.

$$cond\_eval\ \_ : C \leftrightarrow \mathbb{F}_1\,\Sigma$$

Enabledness of a transition tr w.r.t. configuration $conf$ and event set env is captured in the next definition.

$$enabled\ \_ : \mathbb{F}(TR \times \mathbb{F}_1\,\Sigma \times \mathbb{F}\,EV)$$

$$\begin{aligned}
&\forall\, tr : TR;\ conf : \mathbb{F}_1\,\Sigma;\ env : \mathbb{F}\,EV \bullet \\
&enabled(tr, conf, env) \Leftrightarrow (tr.source \subseteq conf \\
&\qquad\qquad\qquad \land\ event_eval(tr.label.event_expr, env) \\
&\qquad\qquad\qquad \land\ cond_eval(tr.label.condition, conf))
\end{aligned}$$

Exit and enter states. In contrast to non-hierarchical automata, in statecharts the states which are exited, resp., entered as a result of taking a transition, are not necessary identical to sources, resp., targets of that transition. Paper [3]: "When the transition tr is taken, all proper descendants of its scope in which the system resided at the beginning of the step are exited, and all proper descendants of the scope in which the system will reside as a result of executing

tr are entered." We define the function $exit$ and $enter$ for transition tr w.r.t. configuration $conf$ and state hierarchy $tree$:

$$\begin{array}{l}
exit, enter : TR \times \mathbb{F}_1\, \Sigma \times StateTree \twoheadrightarrow\!\!\!\!\!\!\mapsto \mathbb{F}_1\, \Sigma \\
\hline
\forall\, tr : TR;\ conf : \mathbb{F}_1\, \Sigma;\ tree : StateTree \bullet \\
\quad \exists\, Uexit, Uenter : tree.\rho(scope(tr, tree)) \bullet \\
\qquad Anc(Uexit, tr.source, tree) \wedge Anc(Uenter, tr.target, tree) \\
\qquad \wedge\ exit(tr, conf, tree) = conf \cap \rho^*(Uexit, tree) \\
\qquad \wedge\ (\exists\, subconf : \mathbb{F}_1\, \Sigma \bullet (enter(tr, conf, tree) = subconf \\
\qquad\qquad\qquad \wedge\ configuration(Uenter, tree, subconf) \wedge tr.target \subseteq subconf))
\end{array}$$

Note that $enter$ is well-defined (follows from the definition of target states of a transition as in SC).

Actions associated with states. Generation of events is associated with exiting and entering of states when a transition is taken. We define total injections $exited$ and $entered$ that convert states exited and entered, resp., to the corresponding events.

$$exited, entered : \Sigma \rightarrowtail EV$$

Conflicting transitions. Conflicting transitions can not be taken together in one execution step. Paper [3]: "We say that two CT's are in *conflict* if there is some common state that would be exited if any one of them were to be taken." A set of transitions trs is conflicting w.r.t. configuration $conf$ and state hierarchy $tree$ if it contains at least two conflicting transitions.

$$\begin{array}{l}
conflicting\ _ : \mathbb{F}(\mathbb{F}\ TR \times \mathbb{F}_1\, \Sigma \times StateTree) \\
\hline
\forall\, trs : \mathbb{F}\ TR;\ conf : \mathbb{F}_1\, \Sigma;\ tree : StateTree \bullet \\
\quad conflicting(trs, conf, tree) \Leftrightarrow \\
\qquad (\exists\, tr1, tr2 : trs \bullet tr1 \neq tr2 \wedge exit(tr1, conf, tree) \cap exit(tr2, conf, tree) \neq \emptyset)
\end{array}$$

Priority of transitions. Paper [3]: "Let tx and ty be two conflicting transitions, and let S_x and S_y be their scopes. Since these two transitions are in conflict, there must be a common state in their source sets, which implies that their scopes cannot be orthogonal or exclusive. Unless they are equal, one of the two scopes must be an ancestor of the other in the state hierarchy. Priority is given to the transition whose scope is higher in the hierarchy." In the next lemma we state the connection between the notions of 'priority' and 'being in conflict'.

Lemma **2** *Let* tx *and* ty *be two enabled transitions in a statechart* sc $\in$ SC *w.r.t. configuration* conf. *If*

$$SAnc(scope(tx, sc.\sigma), \{scope(ty, sc.\sigma)\}, sc.\sigma)$$

holds then tx *and* ty *are in conflict, i.e.* conflicting({tx,ty},conf,sc.σ) *holds, and* tx *has higher priority than* ty.

Status of a statechart. Since we consider only a subset of the language of [3] our status is a subset of status in [3]. We establish the connection between our variables and those of [3]:

csts: a set of states in which the system currently resides (this set must form a configuration);

events: a set of events that were generated internally in the previous step.

We add to status an additional variable **sc:** which is a concrete statechart. This variable remains unchanged during the forthcoming step algorithm. The schema $STATUS$ captures this information.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad STATUS \quad}} \\
csts : \mathbb{F}\,\Sigma \\
events : \mathbb{F}\,EV \\
sc : SC \\
\hline
configuration(sc.\sigma.root, sc.\sigma, csts)
\end{array}$$

Initial status. The only active states in the first status of every computation are $root$ and $init$. The set of events is empty.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad INIT \quad}} \\
STATUS \\
\hline
events = \emptyset \\
csts = \{sc.\sigma.root, sc.\sigma.init\}
\end{array}$$

Definition of the basic step. The description of the basic step algorithm in [3] begins with instructions how the environment can insert events into the status. We postpone the formalization of the interaction until the next subsection.

The following list is a citation from [3] accompanied with our remarks that explain how it relates to our definition in the Z schema $STEP$.

- Compute the set of enabled transitions (corresponds to the set ET).

- Remove from this set all transitions that are in conflict with an enabled transition of higher priority (corresponds to the set HPT).

- Split the set of enabled transitions into maximal non-conflicting sets (corresponds to the set MNS).

- If there are no enabled transitions then the step is empty else choose one of the sets nondeterministically for execution. Let EN be the choice (corresponds to the set EN).

- For each transition X in EN let S_x be the set of states exited and S_n be the set of states entered by X, resp.;

  - delete the states in S_x from the list of states where the system resides;
  - execute the actions associated with exiting the states in S_x (in our case the corresponding event is generated);
  - add to the list of states in which the system resides all the states in S_n;
  - execute the action of X.
  - execute the actions associated with entering the states in S_n (in our case the corresponding event is generated);

  (This corresponds to the assignments to the variables $csts'$ and $events'$, resp.)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_STEP\_\_\_} \\
\Delta STATUS \\
\hline
\textbf{let } ET == \{tr : sc.\tau \mid enabled(tr, csts, events)\} \bullet \\
\textbf{let } HPT == \{etr : ET \mid (\forall\, tr : ET \bullet \neg\, SAnc(scope(tr, sc.\sigma), \{scope(etr, sc.\sigma)\}, sc.\sigma))\} \bullet \\
\textbf{let } MNS == (\mu\, ncs : \mathbb{F}(\mathbb{F}\, HPT) \mid (\forall\, set : ncs \bullet \neg\, conflicting(set, csts, sc.\sigma)) \\
\qquad\qquad \wedge (\forall\, set : ncs;\ tr : HPT \bullet \neg\, conflicting(\{tr\} \cup set, csts, sc.\sigma) \Rightarrow tr \in set)) \\
\quad \bullet (\#MNS = 0 \Rightarrow csts' = csts \wedge events' = \emptyset) \\
\quad \wedge (\#MNS \neq 0 \Rightarrow (\exists\, EN : MNS \bullet \\
\qquad\quad \textbf{let } Exited == \bigcup\{tr : EN \bullet exit(tr, csts, sc.\sigma)\}; \\
\qquad\qquad Entered == \bigcup\{tr : EN \bullet enter(tr, csts, sc.\sigma)\} \bullet \\
\qquad\qquad (csts' = (csts \setminus Exited) \cup Entered \\
\qquad\qquad \wedge\ events' = \bigcup\{tr : EN \bullet tr.label.action\} \\
\qquad\qquad\qquad\qquad \cup\{st : Exited \bullet exited(st)\} \\
\qquad\qquad\qquad\qquad \cup\{st : Entered \bullet entered(st)\}))) \\
sc' = sc \\
\hline
\end{array}
$$

The following lemma claims that the definition is well-defined.

Lemma **3** *Whenever started in a configuration the algorithm terminates and results in a state where* csts *forms a configuration.*

The proof of this property relies on the definition of full CT's, the functions $exit$ and $enter$ and that transitions taken together in one step are orthogonal (the last property follows from the definition of conflict).

4.2 Transition system semantics

Given a statechart $sc \in SC$ we associate with it a transition system $TS = (STATUS, INIT, STEP)$, where $STATUS$ is the universe of states, $INIT$ is the set of initial states and $STEP$ is the transition relation.

In order to explain how computations are generated by such transition systems we investigate the communication of statecharts with the environment. The paper [3] does this in Section "Two models of time". There are two communication modes: in the synchronous time model the communication with the environment is performed after each basic step whereas in the asynchronous model the communication is performed after several basic steps that constitute a super-step. Hence the communication with the environment depends on the time model used.

We formalize the synchronous time model in the following. We use the notation $s_i \stackrel{STEP}{\longrightarrow} s_{i+1}$ to denote that the pair of statuses (s_i, s_{i+1}) is in relation $STEP$.

Definition **1** *A computation is an infinite sequence of statuses* $s_i \in STATUS(i \in \mathbb{N}_0)$*, such that*

$$\forall\, i : \mathbb{N}_0 \bullet (even(i) \Rightarrow s_i.events \subseteq s_{i+1}.events \wedge s_i.csts = s_{i+1}.csts)\, \forall\, i : \mathbb{N}_0 \bullet (odd(i) \Rightarrow s_i \stackrel{STEP}{\longrightarrow} s_{i+1})$$

This definition captures the interplay with the environment: environment steps, which possibly provide new events, alternate with basic steps of the system. This definition records both internally generated events as well as those events provided by the environment. In order to abstract from the explicit influence of the environment we propose the following alternative definition.

Definition **2** *A computation is an infinite sequence of statuses* $s_i \in STATUS(i \in \mathbb{N}_0)$*, such that*

$$\forall\, i : \mathbb{N}_0 \bullet (\exists\, s : STATUS \bullet (s_i.events \subseteq s.events \wedge s_i.csts = s.csts \wedge s \stackrel{STEP}{\longrightarrow} s_{i+1})))$$

This definition abstracts from the events generated by the environment needed to perform the step. The sequences record the internally generated events only.

The transitions system behind the asynchronous model is very similar except that the communication with the environment is permitted after a sequence of basic steps that reached a status from where no further transitions are enabled (this sequence of basic steps forms a so called "super-step").

5 Related Work

The survey [16] lists 20 different statecharts semantics. The first formal semantics appeared in [4]. Pnueli&Shalev ([13]) and Huizing & de Roever ([9, 8]) were the first to discuss which properties statecharts should have and how to design a semantics to obtain them. The first compositional semantics was given in [7].

In [15] a translation of statecharts to process algebra is given; this work can be seen as a link from statecharts to process algebra based tools. The variant of semantics they work with is that from [13].

Statecharts-like languages are Argos [11] and RSML [10, 5]. The syntax of both languages is very much inspired by statecharts but the semantical choices are different. In both cases the compositional semantics and compositional reasoning becomes easier because inter-level transitions are not allowed. Argos is based on the the synchrony hypothesis [1]. Leveson et all ([10, 5]) does not discuss timing issues of the RSML language and hence we do not know whether the synchrony hypothesis applies. A significant non-syntactical difference is that RSML does not associate priorities to transitions (like statecharts and Argos).

6 Conclusion and Further Work

We gave a formal semantics definition for statecharts as implemented in STATEMATE and described in [3]. Our semantics provides the link between STATEMATE and other automatic verification tools. We have implemented an experimental compiler [12] from statecharts to Promela/SPIN ([6]) based on this semantics. Our compiler uses the same data structures for the state hierarchy, transitions, etc. Hence, on one hand, we have shorter compiler design time; on the other hand, formal verification of the compiler becomes simpler since we do not have to deal with data refinement.

The choice of Z as specification language of the formalization was influenced (amongst other reasons) by the existence of the FUZZ type checker ([14]) which turned out to be very useful.

Future work is the formalization of transformations for obtaining full CT's, extension of the sub-dialect (the history concept, time, language of shared variables and while loops) and compositional semantics. Also we will continue the development of the above mentioned compiler.

Acknowledgment. We thank Amir Pnueli and the staff of iLogix Inc. in Israel for discussions and support in early phases of the project. Remarks and discussions with Prof. de Roever on earlier versions of this paper are gratefully acknowledged. The work of E. Mikk is supported by *Technologiestiftung Schleswig-Holstein* within the CATI project. The work of C. Petersohn is supported by *Deutsche Forschungsgesellschaft (DFG).*

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[2] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[3] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (Also available as technical report of Weizmann Institute of Science, CS95-31)*, 5(4):293–333, Oct 1996.

[4] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *Proc. 2nd IEEE Symp. on Logic in Computer Science*, pages 56–64. IEEE Press, 1987.

[5] M.P.E. Heimdahl and N.G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Trans. Soft Eng*, 22(6), June 1996.

[6] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[7] J.J.M Hooman, S. Ramesh, and W-P. de Roever. A compositional axiomatization of Statecharts. *Theoretical Computer Science*, 101:289–335, 1992.

[8] C. Huizing. *Semantics of reactive systems: comparision and full abstraction*. PhD thesis, Technical University Eindhoven, 1991.

[9] C. Huizing and W.-P. de Roever. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters*, 37:205–213, February 1991.

[10] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Trans. Soft Eng*, 20(9):684–707, September 1994.

[11] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Compositions. In *CONCUR'92*, number 630 in Lecture Notes in Computer Science, pages 550–564. Springer, 1992.

[12] E. Mikk, Y. Lakhnech, and M. Siegel. Towards Efficient Modelchecking Statecharts: A Statecharts to Promela Complier. In *3rd International SPIN Workshop*. University of Twente, April 97.

[13] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Proc. Symp. on Theoret. Aspects of Comput. Soft.*, number 526 in Lecture Notes in Computer Science, pages 244–264. Springer Verlag, 1991.

[14] J. M. Spivey. *The fUZZ Manual*. Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.

[15] A.C. Uselton and S.A. Smolka. A Process Algebraic Semantics for Statecharts via State Refinement. In *PROCOMET'94*, IFIP, 1994.

[16] M. von der Beek. A Comparision of Statechart Variants. In W.-P. de Roever H. Langmaack and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 863 in Lecture Notes in Computer Science, pages 128–148. Springer Verlag, September 1994.