Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

An Algebraic Theory Of Boundary Crossing Transitions

Arnab Ray [1,2]

Dept. of Computer Science, SUNY at Stony Brook, Stony Brook NY 11794-4400, USA

Rance Cleaveland [3]

Dept. of Computer Science, SUNY at Stony Brook, Stony Brook NY 11794-4400, USA

Arne Skou [4]

Dept. of Computer Science, Aalborg Univ, Fredriksbajersvej 7E, DK-9220 Aalborg, Denmark.

Abstract

This paper gives a process-algebraic semantics for the *hierarchical state machine* (HSM) fragment of Statecharts, in which state transitions are permitted to cross state boundaries. Although frowned upon by researchers as promoting unstructured modeling, such transitions are used extensively in practice to model parameterized start states and conditional exit states. The purpose of this work is to develop a compositional semantics for HSMs that may be fit together with compositional semantic accounts for Statecharts without boundary-crossing transitions in order to arrive at a compositional theory for virtually the whole Statecharts language. Our technical development consists of a process algebra for HSMs that is equipped with an operational semantics, an argument that bisimulation is a congruence for the algebra, a syntax-directed translation procedure for HSMs into the process algebra, and an equational axiomatization of the algebra.

Key words: Statecharts, Process Algebra, Compositional Semantics, Formal Methods

[1] Research supported by Army Research Office grants DAAD190110003 and DAAD190110019, and National Science Foundation grants CCR-9988489 and CCR-0098037

[2] Email: `arnabray@cs.sunysb.edu`

[3] Email: `rance@cs.sunysb.edu`

[4] Email: `ask@cs.auc.dk`

1 Introduction

Statecharts [6] is a visual language for specifying reactive, embedded and real-time systems. This formalism extends finite-state machines with concepts of hierarchy (state refinement), concurrency, and priority (preemption among transitions). The success of Statecharts in the software engineering community is founded on its easy-to-learn graphical syntax and its support for the hierarchical modeling of complex control software. Dialects of the language [22] may be found in several commercial design notations, including ROOM [19] , STATEMATE [7], Stateflow [10] and UML [4].

Despite its transparent syntax, equipping Statecharts language with a formal operational semantics has proved to be a challenge. Pnueli and Shalev [18] gave the first authoritative operational semantics for the notation. Their approach, however, was not compositional: transitions of a given chart could not be inferred purely from the transitions of its sub-charts, and thus compositional reasoning techniques could not be employed on Statecharts. Various researchers have studied different approaches to overcome this compositionality problem, either by enriching the information stored in a transition [13,20] or redefining transitions as sequences of sub-transitions [11,12]. However, these latter works typically consider only subsets of Statecharts in which *boundary-crossing transitions* are disallowed. The argument offered for this restriction is that such transitions are inherently "unstructured" and their use should be disallowed.

On the other hand, boundary-crossing transitions are quite widely used in practice, and accounting for them compositionally remains an important open problem. In this paper, we address this question by defining a process algebra into which the hierarchical state-machine fragment of Statecharts can be translated in a structure- and semantics-preserving manner. The algebra encodes boundary-crossing transitions using ideas adapted from notions of *method-invocation* and *exception-handling* found in contemporary programming languages. Bisimulation is shown to be a congruence for this algebra, and our translation is structure-preserving; thus, our semantics is compositional. An axiomatization for bisimulation equivalence is also given for the process algebra, as is a discussion of the design decisions we took in designing the language. We also briefly discuss how the language may be combined with the SPL process algebra of [11], which handles the concurrency- and preemption-based features of Statecharts. The final section contains our conclusions and directions for future work.

2 Motivations

Our goal in this paper is to give an operational, compositional semantics for boundary-crossing transitions. The motivation for compositionality is obvious: for a modeling language to "scale" it must be possible to define the semantics of subsystems without reference to how they are embedded within larger systems. Over the past decade, operational approaches to defining language semantics have also predominated, and are especially important for modeling languages for tool-support reasons: tools like simulators and model checkers require an operational semantics for their modeling notations.

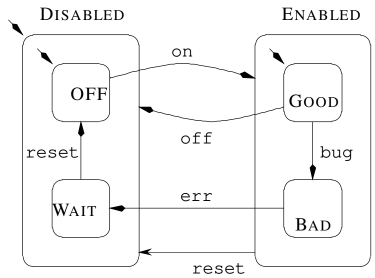


Fig. 1. Boundary-crossing transitions.

The need for boundary-crossing transitions is perhaps less obvious, and we devote the rest of this section to this point. As an illustration of the utility of boundary-crossing transitions in practice, consider the sample Statechart given in Fig. 1 (for clarity, we have simplified the traditional Statecharts transition labels; in particular, the labels contain no *action* component). The diagram depicts a highly abstracted view of one of the authors' personal digital assistant (PDA), which has the following behavior: it may be switched on and, if it is functioning normally, it may be switched off. However, while the machine is switched on a bug sometimes occurs; the PDA becomes disabled and does not respond to being switched on or off until the machine is reset.

Fig. 1 models the PDA using two fundamental modes: DISABLED and ENABLED, with the former being the initial configuration. Each of these fundamental modes has two submodes: OFF and WAIT in the case of DISABLED, and GOOD and BAD in the case of ENABLED. In each case, the first submode mentioned is the "default" submode within the enclosing mode. When the PDA is in the OFF submode of DISABLED, it may switched on (the `on` transition); the PDA then transitions to the default submode (i.e. GOOD) of the ENABLED mode. In this submode the machine may be switched off, or a bug may occur. In the latter case, a transition to the BAD submode occurs, with an error transition then leading to the WAIT submode of the DISABLED mode. In this submode the machine cannot be switched on or off until it is reset. At any point of time in which the machine is ENABLED, the machine may be reset and move to DISABLED wherein the machine goes to the default state of DISABLED ie OFF.

This example highlights the main uses to which boundary-crossing transitions are put in system modeling: *conditional exits*, and *parameterized start states*. The transition labeled `on` exemplifies the former: this transition indicates that mode DISABLED can be only be exited via ON when its submode is OFF; this exit transition is otherwise disabled. The transition labeled `err` also illustrates the conditional-exit phenomenon, but it also highlights how start states can be "parameterized", since the target of this transition is the WAIT submode of DISABLED. In effect, this transition stipulates that when the DISABLED mode is entered via this transition, the starting submode is specified by the transition itself, rather than by the default (submode Off).

The motivation for these two uses of boundary-crossing transitions is evident: they keep models simple by supporting greater reuse of sub-models. In the case above, boundary-crossing transitions could be avoided, but at the cost of having to replicate the Enabled mode so that different start states could be designated. (i.e. there would be a version of Disabled having Wait as its initial submode) Such replication is wasteful and contributes to versioning issues, since different versions of the same sub-model have to be kept consistent with one another. Boundary-crossing transitions offer an elegant solution to this problem. The goal of this paper is to show how a compositional operational semantics may be given for formalisms that have boundary-crossing transitions.

Related work

Several papers have considered the problem of an operational semantics of Statecharts. Pnueli and Shalev [18] have defined the reference operational semantics for Statecharts. Their approach, however, is global, in that it requires an examination of the entire structure of a Statechart term in order to compute its behavior. For large models, this is problematic; in such cases, for reasons of efficiency, one would like to "separately compile" sub-models and then combine sub-model transitions into global transitions. The Pnueli-Shalev semantics does not support such an approach. These observations have led other researchers to define compositional operational approaches to the semantics of Statecharts using a variety of approaches based on process algebra [11,13,20]. However, these frameworks disallow boundary-crossing transitions, arguing that such transitions are "abstraction-breaking" and thus undesirable. In this paper we argue against this point of view and indeed show that boundary-crossing transitions can be accounted for while preserving compositionality.

Huizing, Gerth and de Roever have given a compositional denotational semantics for a variant of Statecharts in [9] that includes boundary-crossing transitions. Their approach relies on the use of "partial transitions", or *unvollendetes*, as a means of detaching a sub-model of a Statechart from the rest of the model. These "dangling transitions" may be seen as entry / exit points into a model, which is a notion exploited in our semantics as well. However their semantics, is trace-based: systems are modeled via sequences of transitions leading from "incoming" partial transitions to "outgoing" ones. Such a semantics is appropriate when a model is deterministic, as earlier versions of Statecharts were. However the semantics of Pnueli and Shalev allows nondeterminism, and purely trace-based models like the one in [9] suffer from an inability to properly to model the *conjunctive nondeterminism* (i.e. "OR-waiting") that is predominant in event-based system modeling.

Related work on giving meaning to boundary-crossing transitions have been published by Mikk, Lakhnech and Siegel in [15]. It introduces extended hiererachical automata (extended HA) that acts as an intermediate format to facilitate the linking of new tools to the [8] Statemate environment. Closely similar to the Argos language [14] (which do not support boundary crossing transitions) extended HA provide support for boundary crossing transitions by using prioritized transtions.

Other formalisms have also used ideas similar to the entry/exit points we define. The idea of splitting transitions in order to localize information has been used in ROOM [19]. In [2], Alur and Grosu define a new Statecharts-type language with compositional semantics and modular reasoning. They also deal with boundary crossing transitions by splitting them into entry and exit points. However, their language contains no events, and their semantics, which is trace-based, has the same problems in modeling nondeterminism and conjunctive nondeterminism that the Huizing-Gerth-de Roever work does.

3 Hierarchical State Machines

This section introduces *Hierarchical State Machines* (HSMs), which are a fragment of conventional Statecharts. Like Statecharts, HSMs permit the definition of hierarchical state machines with boundary-crossing transitions; unlike Statecharts, HSMs have no provision for concurrency, and transition labels may only contain simple events (no trigger/action pairs, no negated events, etc.). In this paper we study HSMs rather than Statecharts in order to study boundary-crossing transitions; at the end, however, we discuss how our work may be combined with that of [11] in order to obtain a compositional theory treating these other language features as well.

3.1 Syntax of HSMs

In order to formalize the notion of HSMs we first introduce the concept of *state frames*, which define hierarchical state structures such as the one in Fig. 1.

Definition 3.1 The set $\mathbb{F}$ of *state frames* is the least set such that $\langle S, s_I, e\rangle \in \mathbb{F}$ when:

(i) S is a finite, nonempty set of *states*;

(ii) $s_I \in S$ is the *start state*; and

(iii) $e : S \rightarrow \mathbb{F}$, the *embedding function*, is a partial function.

Note that $\mathbb{F}$ is defined inductively, and that the definition is well-formed, since the occurrence of $\mathbb{F}$ in Part iii of the definition appears covariantly. The inductiveness of the definition also ensures the absence of cyclic structures." Intuitively, a state frame contains a set of states and a start state, with each state in turn potentially containing a state frame nested within in it, as defined by the embedding function. The form of the definition also ensures that the nesting structure within a state frame state frame can only be finitely deep, since the "base case" of the inductive definition introduces state frames whose embedding function is everywhere undefined.

Example 3.2 Fig. 2 depicts the state frame $\langle\{s_1, s_2\}, s_2, e\rangle$ graphically. Note that e

is partial: $e(s_1)$ is defined, while $e(s_2)$ is not. States, like s_2, for which the embedding function is undefined are sometimes referred to as *basic* states in Statecharts terminology.

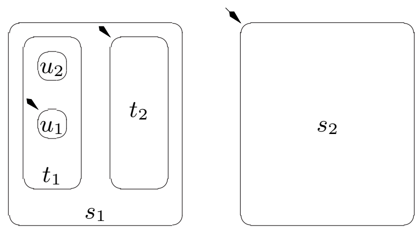


Fig. 2. A state frame.

State frames will be used to define the states of HSMs. To define the transition relation of an HSM, we introduce the notion of *transition endpoints* of a state frame. In what follows, we use $\cdot$ to represent list concatenation, ε for the empty list, and $\preceq$ to represent the prefix ordering on lists (eg $a \preceq a.b$ ie a is a prefix of $a.b$ and would hence come before it in the ordering) . We also abuse notation by identifying single-element lists with the element they contain; so if S is a set and $s \in S$, then we also treat s as a list whose only element is s.

Definition 3.3 The *transition endpoints*, $\mathbb{E}(F)$, of state frame $F = \langle S, s_I, e\rangle$ are defined inductively as follows.

(i) $S \subseteq \mathbb{E}(F)$.

(ii) If $e \in S$, $e(s)$ is defined, and $\ell' \in \mathbb{E}(e(s))$, then $s \cdot \ell' \in \mathbb{E}(F)$.

Intuitively, transition endoints may be thought of as lists of states.

Example 3.4 In the state frame F given in Fig. 2,

$$\mathbb{E}(F) = \{s_1,\ s_2,\ s_1 \cdot t_1,\ s_1 \cdot t_2,\ s_1 \cdot t_1 \cdot u_1,\ s_1 \cdot t_1 \cdot u_2\}.$$

To define HSMs formally, we first fix a set $\mathbb{L}$ of *transition labels*. An HSM now consists of a state frame and a transition relation involving labels and the transition endpoints of the frame.

Definition 3.5 An HSM has form $\langle S, s_I, e, \rightsquigarrow\rangle$, where:

- $F = \langle S, s_I, e\rangle$ is a state frame.
- $\rightsquigarrow \subseteq \mathbb{E}(F) \times \mathbb{L} \times \mathbb{E}(F)$ is the *transition relation*.

Example 3.6 Fig. 3 contains a HSM whose state frame is given by Fig. 2.

3.2 Operational semantics of HSMs

We now define the operational semantics of HSMs by showing how they can be translated into standard, "flat", labeled transition systems. Before making this definition, we first introduce an auxiliary notion on state frames.

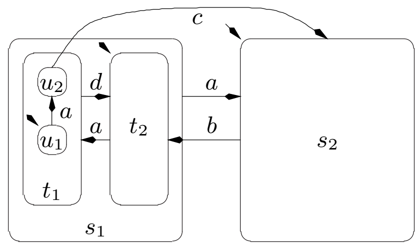


Fig. 3. A hierarchical state machine.

Definition 3.7 Let $F = \langle S, s_I, e\rangle$ be a state frame, and let $\ell \in \mathbb{E}(F)$. Then $\max_F(\ell)$ is defined as follows.

$$\max_F(\ell) = \begin{cases} \ell & \text{if } \ell \in S \text{ and } e(\ell) \text{ undefined} \\ \ell \cdot \max_{F'}(s'_I) & \text{if } \ell \in S \text{ and } e(\ell) = F' = \langle S', s'_I, e'\rangle \\ s \cdot \max_{e(s)}(\ell') & \text{if } \ell = s \cdot \ell' \text{ and } \ell' \in \mathbb{E}(e(s)) \end{cases}$$

Intuitively, $\max_F(\ell)$ extends ℓ into a "maximal" transition endpoint (i.e. one whose last state component is guaranteed to be a basic state) by padding ℓ appropriately with initial states. We now give the semantics of HSMs as follows.

Definition 3.8 Let $H = \langle S, s_I, e, \rightsquigarrow\rangle$ be a HSM, and define $F_H = \langle S, s_I, e\rangle$. Then the *labeled transition system* $\mathrm{LTS}(H) = \langle S_H, s_H, \longrightarrow_H\rangle$, where the state set S_H, initial state $s_H \in S_H$, and transition relation $\longrightarrow_H \subseteq S_H \times \mathbb{L} \times S_H$ are given as follows.

(i) S_H contains the maximal (with respect to $\preceq$) transition endpoints of F:

$$S_H = \{\ell \in \mathbb{E}(F) \mid \forall \ell' \in \mathbb{E}(F).\, \ell \preceq \ell' \implies \ell = \ell'\}.$$

(ii) $s_H = \max_{F_H}(s_I)$.

(iii) $\ell \xrightarrow{a}_H \ell'$ if there exists $\ell_1 \preceq \ell$ such that $\ell_1 \overset{a}{\rightsquigarrow} \ell'_1$ and $\ell' = \max_H(\ell'_1)$.

Example 3.9 With H, H_F given as in Fig. 3, we have:

$$S_H = \{s_1 \cdot t_1 \cdot u_1, s_1 \cdot t_1 \cdot u_2, s_1 \cdot t_2, s_2\}$$
$$s_H = \max_F(s_2) = s_2$$

The transition relation $\longrightarrow_H$ consists of the following.

$$s_1 \cdot t_2 \xrightarrow{a}_H s_2$$
$$s_1 \cdot t_2 \xrightarrow{a}_H s_1 \cdot t_1 \cdot u_1$$
$$s_1 \cdot t_1 \cdot u_1 \xrightarrow{a}_H s_2$$
$$s_1 \cdot t_1 \cdot u_2 \xrightarrow{a}_H s_2$$
$$s_1 \cdot t_1 \cdot u_1 \xrightarrow{a}_H s_1 \cdot t_1 \cdot u_2$$

$$s_2 \xrightarrow{b}_H s_1 \cdot t_2$$
$$s_1 \cdot t_1 \cdot u_2 \xrightarrow{c}_H s_2$$
$$s_1 \cdot t_1 \cdot u_1 \xrightarrow{d}_H s_1 \cdot t_2$$
$$s_1 \cdot t_1 \cdot u_2 \xrightarrow{d}_H s_1 \cdot t_2$$

3.3 Compositionality

From the definition of HSMs it is apparent why compositionality is problematic: while the states of a HSM are defined compositionally, the edge relation $\rightsquigarrow$ is given globally. This gives the definition of the semantic transition relation $\longrightarrow_H$ a non-compositional flavor: in particular, transitions for a state are not inferable based on the basis transitions of sub-states. This is the reason that boundary-crossing transitions have not been considered in compositional treatments of Statecharts semantics.

However, the discussion in Section 2 hints at a compositional approach to a semantics: treat states as having "entry" and "exit" points into their internal structure. Rather than atomic entities, boundary-crossing transitions may be thought of as consisting of multiple "segments" that connect entry / exit points appropriately. The information about these transition segments is stored locally in the process definitions; when the components are composed, the transition segments are "stitched" together to form the global transition.

Fig. 4 is Fig. 1 redrawn by explicitly showing entry and exit points by blackened triangles.

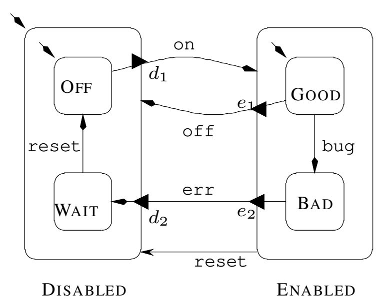
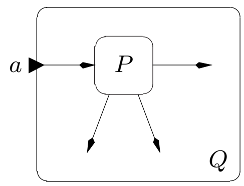


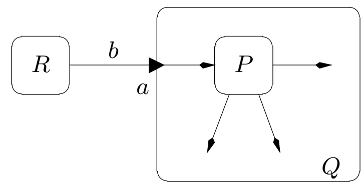
Fig. 4. Entry and exit points.

The next section formalizes these intuitions in the setting of process algebra. Specifically, we define a process algebra containing notions of entry and exit points, together with operators for "entering" and "exiting" process terms. We treat these notions semantically by in essence viewing entry points as analogous to "method declarations", with entry being like method invocation, and exit points as exceptions and exiting as exception handling. We establish compositionality for the algebra by showing that bisimulation [16] is a congruence for the algebra. Finally, we use this algebra as a vehicle for giving a compositional semantics, via a syntax-directed translation, for HSMs.

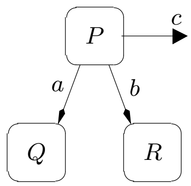
The notions of entry and exit points have been used in other graphical state-machine-based formalisms, most notably ROOM [19] and the hierarchic state machines of [2]. Neither, however, provides an algebraic treatment of these notions, and the semantics of ROOM, while precise, is not formal. As noted earlier, the state machines of [2] do not contain the notion of event, and the trace-based semantics there, while adequate for that framework, suffers from the usual problems in the presence of conjunctive (i.e. "or-waiting") nondeterminism that is inherent in event-based notations like HSMs.



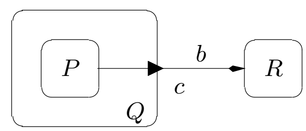
(1) $Q \triangleq (a)P$



(2) $R \triangleq b.<a>Q$



(3) $P \triangleq a.Q + b.R + \lceil c \rceil$



(4) $Q = \lceil c \rceil \; [\rangle \; (\lfloor c \rfloor b.R)$

Fig. 5. HSMs and associated HPA terms

4 A Process Algebra for HSMs

The syntax of HPA, our process algebra for HSMs, is parameterized with respect to the set $\mathbb{L}$ of transition labels used in the definition of HSMs and a countably infinite set $\mathbb{C}$ of process identifiers. In what follows, we assume $a \in \mathbb{L}$ and $X \in \mathbb{C}$. The terms of HPA may now be given as follows.

$$\begin{array}{rll} P ::= & nil \mid X \mid a.P \mid P + P & \text{(Regular CCS)} \\ \mid & P[\rangle P & \text{(Embedding)} \\ \mid & (a)P & \text{(Entry point)} \\ \mid & <a>P & \text{(Enter)} \\ \mid & \lceil a \rceil & \text{(Exit point)} \\ \mid & \lfloor a \rfloor P & \text{(Handle exit)} \end{array}$$

In the sequel we use $\mathbb{T}$ to represent the set of all HPA terms.

4.1 Understanding HPA

HPA extends regular CCS [16] (i.e. CCS without parallel composition, restriction or relabeling) with operators for “embedding” one process within another (this operator is reminiscent of the disabling operator of LOTOS [3], and is also used as an embedding operator in [20]), for defining entry and exit points, and for exercising these entry and exit points. An intuitive reading of the operators is as follows. The term nil represents the terminated process that perform no actions, while $a.P$

means a process that can perform an a action and become P. $P + P$ means a process's behavior is determined by the non-deterministic choice operator $+$. X constitutes an "invocation" of a process declared via an equation $X \triangleq P$. In $P[\rangle Q$, P is allowed to perform its actions until Q "disables" it by performing an action; after this point, the process behaves like Q. The term $(a)P$ may be thought of as the declaration of a "method" a with body P that, when invoked, behaves like P. Process $<a>P$ "invokes" method a in P, causing control to shift to the body of the method. Process $\lceil a \rceil$ attempts to "exit at a"; this is analogous to raising an exception named a. Handlers for such exits / exceptions may be defined using $\lfloor a \rfloor P$, with the $[\rangle$ operator being used to connect processes that raise exceptions with their handlers.

Fig. 5 contains examples highlighting the HSM intuitions behind these constructs. Example (1) shows the intended correspondence between (a) and entry points: $(a)P$ allows a transition to "attach" to P via entry point a.

Example (2) shows how another process R uses Q's entry point on a; the net intended effect is that in R, a b transition will lead to sub-state P with the larger state Q. Note that there are two transition segments: one from R to the boundary of Q and another from that point to the boundary of P. The full transition is formed by joining these segments together.

Example (3) shows how exit points may be defined. Here P can evolve into Q or R but it can also do an exit through an exit point c.

Finally, Example (4) shows how an exit point c is handled. Here P defines an exit point c and crosses Q's boundary and becomes R. Another thing to note here is the $[\rangle$ operator; in addition to its use as an "embedding" operator it also is responsible for "connecting" exits to handlers.

4.2 Operational Semantics of HPA

To formalize this semantics we use the SOS approach [17]. The rules are contained in Fig. 6. As is standard in process algebra, we assume the existence of an environment of equations of form $X \triangleq P$ associating terms to process identifiers.

Before discussing the rules we first note that the labels on the transitions have a richer structure than those for HSMs. In particular, in addition to elements of the set $\mathbb{L}$, transitions may be labeled by:

- (a) existence of an entry point / method declaration a
- $\lceil a \rceil$ capability of exiting via / raising exception a
- $\lfloor a \rfloor$ capability of attaching to, or *handling*, exit / exception a.

We use $(\mathbb{L})$, $\lceil \mathbb{L} \rceil$ and $\lfloor \mathbb{L} \rfloor$ to denote the set of all entry, exit and "handle" actions, respectively.

Rules (P), (S1), (S2) and (R) are standard from CCS, and we do not elaborate on them here. Rules (Em1)–(Em3) define the behavior of the embedding operator. (Em1) stipulates that $P_1[\rangle P_2$ can perform the non-exit transitions of P_1 while P_2 idles, while (Em2) states that this process can engage in exactly the non-handler

$$\text{(P)}\ \boxed{\dfrac{}{a.P \xrightarrow{a} P}} \quad \text{(S1)}\ \boxed{\dfrac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'}} \quad \text{(S2)}\ \boxed{\dfrac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'}} \quad \text{(R)}\ \boxed{\dfrac{P \xrightarrow{a} P'}{X \xrightarrow{a} P'}\ [X \triangleq P]}$$

$$\text{(Em1)}\ \boxed{\dfrac{P_1 \xrightarrow{a} P_1'}{P_1 [\rangle P_2 \xrightarrow{a} P_1' [\rangle P_2}\ [a \notin \lceil \mathbb{L} \rceil]} \quad \text{(Em2)}\ \boxed{\dfrac{P_2 \xrightarrow{a} P_2'}{P_1 [\rangle P_2 \xrightarrow{a} P_2'}\ [a \notin \lfloor \mathbb{L} \rfloor]}$$

$$\text{(Em3)}\ \boxed{\dfrac{P_1 \xrightarrow{\lceil a \rceil} P_1', \quad P_2 \xrightarrow{\lfloor a \rfloor} P_2', \quad P_2' \xrightarrow{b} P_2''}{P_1 [\rangle P_2 \xrightarrow{b} P_2''}}$$

$$\text{(Entry)}\ \boxed{\dfrac{}{(a)P \xrightarrow{(a)} P}} \quad \text{(Enter)}\ \boxed{\dfrac{P \xrightarrow{(a)} P', \quad P' \xrightarrow{b} P''}{\texttt{<}a\texttt{>}P \xrightarrow{b} P''}} \quad \text{(Ex)}\ \boxed{\dfrac{}{\lceil a \rceil \xrightarrow{\lceil a \rceil} \mathit{nil}}} \quad \text{(H)}\ \boxed{\dfrac{}{\lfloor a \rfloor P \xrightarrow{\lfloor a \rfloor} P}}$$

Fig. 6. SOS rules for HPA.

transitions of P_2, with P_1 being disabled in the process. (Em3) establishes that P_2 can "handle" exits enabled by P_1. Rule (Entry) states that $(a)P$ enables an entry point a, while (Enter) establishes that $\texttt{<}a\texttt{>}P$ can "enter" the a entry point of P. Rules (Ex) and (H) define the behavior of $\lceil a \rceil$, which is an exit point, and $\lfloor a \rfloor P$, which is capable of attaching to exit point a.

Note that rules (Em3) and (Enter) use *chaining* among the premises: the target of one transition in the premise is used as the source of another transition in the premise.

Example

The following is an encoding of the "decorated" HSM in Fig. 4.

$$\begin{aligned}
\textsc{Disabled} &\triangleq (\textsc{Off} + (d_2).\textsc{Wait})\ [\rangle\ \text{DO} \\
\textsc{Off} &\triangleq \lceil d_1 \rceil \\
\textsc{Wait} &\triangleq \texttt{reset}.\textsc{Off} \\
\textsc{Enabled} &\triangleq \textsc{Good}\ [\rangle\ \text{EO} \\
\textsc{Good} &\triangleq \texttt{bug}.\textsc{Bad} + \lceil e_1 \rceil \\
\textsc{Bad} &\triangleq \lceil e_2 \rceil \\
\text{DO} &\triangleq \lfloor d_1 \rfloor \texttt{on}.\textsc{Enabled} \\
\text{EO} &\triangleq \lfloor e_1 \rfloor \texttt{off}.\textsc{Disabled} + \lfloor e_2 \rfloor \texttt{err}.\texttt{<}d_2\texttt{>}\textsc{Disabled} + \texttt{reset}.\text{DISABLED}
\end{aligned}$$

Using the SOS rules one can infer the existence of transitions such as the following.

$$\text{DISABLED} \xrightarrow{(d_2)} \text{WAIT} \,[\rangle\, \text{DO} \qquad \text{DISABLED} \xrightarrow{\texttt{on}} \text{ENABLED}$$
$$\text{ENABLED} \xrightarrow{\texttt{bug}} \text{BAD} \,[\rangle\, \text{EO} \qquad \text{BAD} \,[\rangle\, \text{EO} \xrightarrow{\texttt{err}} \text{WAIT} \,[\rangle\, \text{DO}$$

4.3 Compositionality

Process algebra has a rich meta-theory for bisimulation that has been developed around SOS operational definitions. One such result may be found in [21], which asserts that, provided the SOS rules for a process algebra fall within a specific syntactic format presented in that paper, bisimulation equivalence [16] will be a congruence for that algebra. The SOS rules of HPA satisfy this criterion, and thus it immediately follows that bisimulation is a congruence for HPA.

5 Converting HSMs to HPA

In the previous two sections we developed the theories of HSMs and HPA, with a view toward using the latter to give a compositional semantics of the former. We complete this program in this section by giving a structure-based translation of HSMs into HPA terms.

The algorithm below takes as its input a HSM and outputs the corresponding PA term, together with a list of process-identifier declarations used in the term. The key elements of the translation are: (1) the use of the embedding operator $[\rangle$ to embed an HSM inside the state of another HSM; (2) the handling of "incoming" boundary-crossing transitions using the (a) and $<a>$ constructs; and (3) the encoding of "outgoing" boundary-crossing transitions via $\lceil a \rceil$ and $\lfloor a \rfloor$ constructs. More detailed information can be found in the comments in the pseudo-code.

Input:

HSM H = $\langle S, s_I, e, \rightsquigarrow \rangle$

Output:

$\langle X_{s_I}, E \rangle$, where:

- X_{s_I} is a variable associated with s_I
- E is a list of HPA declarations of the form $X_s \triangleq P_s$ indexed by $s \in \mathbb{E}(\langle S, s_I, e \rangle)$.

Assume:

Existence of injective function $f : \rightsquigarrow \times \mathbb{E}\langle S, s_I, e \rangle \to \mathbb{L}$ that, given an edge and a transition endpoint, returns a unique label.

algorithm Trans ($\langle S, s_I, e, \rightsquigarrow \rangle$)
 return TransAux (ε, $\langle S, s_I, e \rangle$)
end algorithm Trans

/* TransAux does the "real work" of the translation. The parameter ℓ records the "path" in the HSM to the state frame that is the second parameter */

```
algorithm TransAux (ℓ, ⟨S, s_I, e⟩)
    E := ε /* E is list of equations to return */
 /* Generate equation for each state in S. */
    for each s ∈ S do:
        e_O := nil; /* e_O will be term for "outer part" of s */
        if e(s) is undefined then e_I := nil /* e_I: "inner part" of s */
        else ⟨X′, E′⟩ := TransAux (ℓ · s, e(s)); /* recursive call */
        E := E · E′;
        e_I := X′
    /* Now handle transitions involving states in S.*/
        for each t = ⟨ℓ_1, a, ℓ_2⟩ ∈⇝ do
        /* Case when transition originates from the current state.*/
            if ℓ_1 = ℓ · s then
                if ℓ_2 = ℓ · s′ then /* non-BCT */
                    e_O := e_O + a.X_{s′}
                else if ℓ_2 = ℓ · s′ · ℓ′_2 /* BCT entering sibling state*/
                    then e_O := e_O + a.<f(t, ℓ · s′)>X′_s
                else e_O := e_O + ⌈f(t, ℓ)⌉ /* Outgoing for parent */

        /* Case when transition originates from substate of current state.*/
            else if ℓ_1 = ℓ · s · ℓ′_1 some ℓ′_1 then
                if ℓ_2 = ℓ · s′ then /* transition target is sibling of current state */
                    e_O := e_O + ⌊f(t, ℓ · s)⌋a.X_{s′}
                else if ℓ_2 = ℓ · s′ · ℓ′_2 then /* transition target is child of sibling */
                    e_O := e_O + ⌊f(t, ℓ · s)⌋a.<f(t, ℓ · s′)>X′_s
                else /* transition is BCT for parent also */
                    e_O := e_O + ⌊f(t, ℓ · s)⌋⌈f(t, ℓ)⌉
        /* Case when transition is incoming to current state.*/
            else if ℓ_2 = ℓ · s · s′ · ℓ′_2 then
                if ℓ′_2 = ε then /* i.e. transition ends at s' */
                    e_I = e_I + (f(t, ℓ · s))X_{ℓ·s′}
                else e_I := e_I + (f(t, ℓ · s))<f(t, ℓ · s · s′)>X_{ℓ·s·s′}
            od;
        E := (X_s ≜ (e_I [⟩ e_O)) · E;
    od;
    return ⟨X_{s_I}, E⟩
end algorithm TransAux
```

Correctness of the Translation

We now outline the proof of correctness of the translation procedure. Our general approach will be to show that one can build a bisimulation-like relation between HSM states and HPA terms that relates the HSM initial "semantic state" to the HPA term returned by our translation.

Of course, HPA terms are capable of transitions that HSM states are not: in particular, transitions labeled by (a), $\lceil a \rceil$ and $\lfloor a \rfloor$ cannot be matched by HSM states. So the relation that we construct will only consider elements of $\mathbb{L}$; the other transitions in the HPA terms may be thought of as the "infrastructure" that is needed for compositionality.

Theorem 5.1 *Let $H = \langle S, s_I, e, \rightsquigarrow \rangle$ be an HSM, with* LTS$(H) = \langle S_H, s_H, \longrightarrow_H \rangle$ *and $Trans(H) = \langle P, E \rangle$ the HPA term and supporting declarations constructed by our algorithm. Then there exists $R \subseteq S_H \times \mathbb{T}$ such that $s_H \, R \, P$, and such that whenever $s \, R \, t$, then* for any $a \in \mathbb{L}$*:*

(i) *If $s \xrightarrow{a}_H s'$ then there exists t' such that $t \xrightarrow{a} t'$ and $s' \; R \; t'$.*

(ii) *If $t \xrightarrow{a} t'$ then there exists s' such that $s \xrightarrow{a}_H s'$ and $s' \; R \; t'$.*

6 Axiomatizing HPA

Although we have referred to HPA as a process algebra, we have not yet endowed it with any algebraic structure. In this section we address this issue by giving a sound and complete equational axiomatization for the finite (i.e. identifier-free) fragment of HPA.

Another apparently relevant result from the SOS meta-theory of process algebra may be found in [1]; in that paper, a method is given for automatically generating sound and complete axiomatizations for bisimulation from SOS rules. However, the rules for HPA fall outside the format considered in that paper, so we cannot use that work.

Table 1 lists the axioms; we comment on them briefly here. (A1)–(A4) are the standard monoid and absorption laws for CCS. The remaining equations are designed to eliminate occurrences of $[\rangle$ and $<a>$ from terms: key among these are (Em5), which accounts for the interaction between exit points and exit handlers, and (En4), which captures the interplay between entry points and the enter operation.

Proving these laws sound and complete may be done using standard techniques [16]. Soundness follows from the construction of appropriate bisimulations. Completeness relies on the definition of normal syntactic forms; the definition of these may be intuited from the definition of r in the prelude to Axioms (Em3)–(Em6) in the table. Although laborious, the details are routine and are omitted.

Table 1
Axiomatizing HPA for bisimulation.

(A1)	$x + y = y + x$	(A2)	$x + (y + z) = (x + y) + z$
(A3)	$x + nil = x$	(A4)	$x + x = x$
(Em1)	$nil \,[\rangle\, x = x$	(Em2)	$(x + y) \,[\rangle\, z = (x \,[\rangle\, z) + (y \,[\rangle\, z)$

In (Em3)–(Em6), let

$$r' = \sum_i a_i.x_i + \sum_j (b_j)y_j + \sum_k \lceil c_k \rceil$$
$$r = r' + \sum_l \lfloor d_l \rfloor z_l$$

(Em3)	$(a.x) \,[\rangle\, r = a.(x \,[\rangle\, r) + r'$	(Em4)	$((a)x) \,[\rangle\, r = (a)(x \,[\rangle\, r) + r'$
(Em5)	$(\lceil a \rceil) \,[\rangle\, r = r' + \sum_{d_l = a} z_l$	(Em6)	$(\lfloor a \rfloor x) \,[\rangle\, r = \lfloor a \rfloor (x \,[\rangle\, r) + r'$
(En1)	$<a>nil = nil$	(En2)	$<a>(b.x) = nil$
(En3)	$<a>(x + y) = (<a>x) + (<a>y)$	(En4)	$<a>(a)x = x$
(En5)	$<a>(b)x = nil$ if $a \neq b$	(En6)	$<a>\lceil b \rceil = nil$
(En7)	$<a>(\lfloor b \rfloor x) = nil$		

7 Discussion

In this section we present the rationales for some the decisions we took in developing the work in this paper. We also outline how HPA may be combined with the process algebra given in [11] in order to obtain a compositional semantic theory for Statecharts with boundary-crossing transitions.

To begin with, one may wonder why we elected to give our compositional semantics of HSMs indirectly, via a translation into a process algebra. Both pragmatic and philosophical considerations played a role in this decision. On the one hand, the work grew out of an effort to model and verify a controller for a battlefield medical device (an automatic resuscitator). In that study the state-machine models we were working with had boundary-crossing transitions, but the verification tool we were using (the CWB-NC [5]) had only a textual interface. We therefore had to come up with a textual language for boundary-crossing transitions, and that notation played a major part in the work here. At a more conceptual level, however, compositionality is an inherent feature of process algebras; when attempting to define a compositional semantics for a new modeling-language feature, a natural strategy to pursue involves defining an algebra for the feature so that the rich meta-theory of process algebra may be brought to bear. (In our case, we got "bisimulation congruence" for free because of this.) Once one has an understanding of the basic conceptual issues, one may then explore more direct semantics accounts.

Another issue involves the design of the process algebra HPA. In general, three main considerations guided our development of this language: HSMs should be easy to encode in the language; the operational and equational theory should be clean; and the language should be easy to combine with the process algebra in [11], which is used as a vehicle for giving a compositional semantics for Statecharts without boundary-crossing transitions. Other considerations, such as how to handle various types of nondeterminism, were generally left aside. Thus, for example, the process

$$<a>((a)b.nil + (a)c.nil)$$

is bisimilar to the process $b.nil + c.nil$, even though some may argue that the existence of two "different" a method declarations / entry points in the body of the $<a> \cdots$ term ought to lead to a nondeterministic (internal) choice between the $b.nil$ and $c.nil$. However, our needs in this paper do not depend on reflecting such internal choices, we elected not to try to force them into the language. Similar considerations underpin our treatment of exceptions / exit points ($\lceil a \rceil$) and exit handling ($\lfloor a \rfloor P$). The language in its current form requires that exits be handled immediately by a handler; in particular, in the process

$$(\lceil a \rceil \,[\rangle\, nil) \,[\rangle\, \lfloor a \rfloor P$$

the a-handler *cannot* handle the a-exit because of the intervening nil. For the purposes of translating HSMs this is not problematic, and indeed the axiomatization of this language is slightly simpler because of this design choice. However, one could imagine redesigning the language to allow exceptions / exits to be handled more flexibly (by, e.g., eliminating the side conditions in the SOS rules (Em1) and (Em2) for $[\rangle$); the resulting axiomatization, however, is more complex.

We close this section by commenting on how HPA may be enriched in order to model the other features of Statecharts, including concurrency and preemption. In [11] a process algebra, SPL, was introduced in order to give a compositional semantics to Statecharts without boundary-crossing transitions. The syntax of SPL is parameterized with respect to a set of labels (event names); it shares some operators with HPA ($+$, recursion), has some operators that are not present at all in HPA (emit, parallel composition, restriction, delay), and has some that are similar to, but differ in key respects, from those in HPA (prefix, disabling, enabling). The operational semantics is given via a transition relation that is labeled with two sets of events: those emitted, and those that must be absent from the environment. It it straightforward to merge HPA into SPL as follows: add entry, enter, exit and handle operators; use the SPL notion of prefixing; adapt the definitions of parallel composition and restriction to be sensitive to HPA transition labels; and modify the SPL disabling operator in order to introduce an exit-handling capability. Since the transition labels of SPL and HPA are disjoint, the modifications are not difficult.

However, the resulting language is somewhat large; for this reason we have focused our technical development on HPA in this paper.

8 Conclusions and Directions for Future Work

In this paper we have developed a compositional operational semantics for hierarchical state machines (HSMs). HSMs enrich traditional state machines with capabilities for embedding state machines within states of other machines and for transitions that cross state boundaries. Traditional wisdom has been that such boundary-crossing transitions are inherently non-compositional; we prove that this need not be the case. The semantics relies on a structure-preserving translation into a process algebra for which bisimulation equivalence is a congruence. This last fact ensures that the semantics of HSMs is also compositional. We also give a sound and complete axiomatization of the process algebra for bisimulation equivalence.

As future work, we wish to study the incorporation of our ideas into richer calculi, such as the Statecharts process algebra of [11], which includes treatments for the other constructs in Statecharts that are not present in HSMs. We also think ideas from [12] may be adopted to this paper in order to give a direct compositional semantics to HSMs rather than an indirect one, via translation, as is done in this paper.

References

[1] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994.

[2] Rajeev Alur and Radu Grosu. Modular refinement of hierarchic reactive machines. *Principles of Programming Languages(POPL) 390-402*, 2000.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[4] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[5] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 41(1):39–47, 2002.

[6] D.Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming,8*, pages 231–274, 1987.

[7] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: the STATEMATE Approach*. McGrawHill, 1998.

[8] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[9] C. Huizing, R. Gerth, and W.P. de Roever. CAAP '88, 13th colloquium on trees in algebra and programming. In Max Dauchet and Maurice Nivat, editors, *CAAP*, volume 299 of *Lecture Notes in Computer Science*, pages 271–294, Nancy, France, March 1988. Springer.

[10] The MathWorks Inc. `www.mathworks.com/products/stateflow`.

[11] G. Luettgen, M. von der Beeck, and R. Cleaveland. Statecharts via process algebra. In J. Baeten and S. Mauw, editors, *Tenth International Conference on Concurrency Theory (CONCUR '99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 399–414, Eindhoven, The Netherlands, August 1999. Springer-Verlag.

[12] G. Luettgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In D. Rosenblum, editor, *Eighth International Symposium on Foundations of Software Engineering*, pages 120–129, San Diego, California, November 2000. ACM.

[13] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of Statecharts. In U. Montanari and V. Sassone, editors, *Seventh International Conference on Concurrency Theory (CONCUR '96),*, volume 1119 of *Lecture Notes in Computer Science*, pages 687–702, Pisa, Italy, August 1996. Springer-Verlag.

[14] F. Maraninchi. The argos language:graphical representation of automata and description of reactive systems. *IEEE Workshop on Visual Languages*, 1991.

[15] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. *Asian Computer Science Conference (ASIAN'97), LNCS 1345*, 1997.

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[17] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[18] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software: Proc. of the International Conference TACS'91*, pages 244–264. Springer, Berlin, Heidelberg, 1991.

[19] B. Selic, G. Gullelson, and P. Ward. *Real Time Object Oriented Modelling and Design*. J.Wiley, 1994.

[20] A.C. Uselton and S.A. Smolka. A compositional semantics for Statecharts using labeled transition systems. In B. Jonsson and J. Parrow, editors, *Fifth International Conference on Concurrency Theory (CONCUR '94)*, volume 836 of *Lecture Notes in Computer Science*, pages 2–17, Uppsala, Sweden, August 1994. Springer-Verlag.

[21] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In B. Jonsson and J. Parrow, editors, *Fifth International Conference on Concurrency Theory (CONCUR '94)*, volume 836 of *Lecture Notes in Computer Science*, pages 433–448, Uppsala, Sweden, August 1994. Springer-Verlag.

[22] M. von der Beeck. A comparison of statecharts variants. *Lecture Notes in Computer Science*, 863:128–128, 1994.