A (More) Formal Definition of Communicating Real-Time State Machines *

Technical Report No. 93-08-01

Alan C. Shaw

Abstract

The language of communicating real-time state machines is defined precisely in three parts. First, the syntax of a single machine and of a set of connected machines are described. Then, the static semantics is described as the set of execution paths obtained through a static analysis. Finally, the dynamic semantics is defined by specifying a simulation algorithm that produces execution traces or histories. The most difficult and interesting aspect is that dealing with time.

1 Introduction

Communicating Real-Time State Machines (CRSMs), a requirements and design specification language for real-time systems, were introduced and defined informally in [Shaw 92]. They are universal state machines, with guarded commands as transitions, synchronous IO communication over unidirectional channels much like Hoare's CSP [Hoare 85], and mechanisms for specifying execution times of transitions and for accessing real-time, using a continuous time model. The original paper described the CRSM notation, presented many applications examples, outlined an algorithm for simulating the execution of a system of CRSMs, and discussed some methods for reasoning about system behaviors in terms of CRSM event traces.

A discrete time version of CRSMs has been implemented [Raju & Shaw 92]. Further, a verifier for the discrete time version based on timed reachability graphs has been developed and built [Raju 93]. In the verifier paper, Raju presents a "tuple" definition for the form or syntax of discrete time CRSMs.

Our purpose here is to provide a more precise definition of the semantics of CRSMs. We do this in three parts. First, the syntax of an individual CRSM and of a system of CRSMs is described. Then, the static semantics is defined as the execution histories or traces that can be obtained through a

*This research was supported in part by the National Science Foundation under grant number CCR-9200858.

static analysis. Finally, the dynamic semantics is specified operationally by means of a simulation algorithm that generates execution histories.

2 Syntax

2.1 General CRSM

A CRSM is a tuple $M = (S,\ I,\ O,\ V,\ G,\ C,\ E,\ T,\ s_0)$.

1. S is a finite set of states.
2. I and O are each finite sets of input and output channels, respectively. Excepting the special real-time clock CRSM (defined below in Section 2.2), I always includes a timer channel RT_M. Every channel also has an associated *type*. The type of RT_M is the non-negative real numbers. $I \cap O = \emptyset$

   Aside: The type of a channel specifies the data type of the message transmitted across the channel.
3. V, G, C, and E are finite sets of variables, guards, commands, and expressions, respectively, in some programming language P.

   Aside: Elements of V may have structure, e.g., records, arrays, and lists.
4. V is a finite set of typed variables.
5. G is a finite set of side-effect-free Boolean expressions of P, composed from elements of V and constants.
6. $C = C_{internal} \cup C_{io}$. An element of $C_{internal}$ may be any terminating IO-free, sequential program in P or it may be any identifying name.

   Aside: A name denotes some physical activity; a program describes a computation.

   C_{io} may contain input commands $ch_i(v)?$ or output commands $ch_o(expr)!$ where $ch_i \in I$, $ch_o \in O$, v is a variable of P of the type of ch_i, and $expr$ is an expression of P of the type of ch_o.

   Aside: The *IO* syntax is borrowed from CSP([Hoare85]), because the semantics are similar.
7. E is the set of non-negative real-valued expressions from P composed from V, constants, and the special symbol "∞".

   Aside: The elements of $E \times E$ denote lower and upper time bounds for the termination of a command.

8. T is a finite set of transitions. $T \subseteq S \times S \times G \times C \times E \times E$.

9. s_0 is the start or initial state of M. $s_0 \in S$.

2.2 Real-Time Clock CRSM

Corresponding to every general CRSM M is a real-time clock CRSM denoted RTC_M.

$$RTC_M = (\{s_0\}, \emptyset, \{RT_M\}, \{rt\}, \{true\}, \{RT_M(rt)!\}, \{0, \infty\}, \{(s_0, s_0, true, RT_M(rt)!, 0, \infty)\}, s_0)$$

The variable rt is the *only* system-wide global variable and is directly accessible *only* by the clock CRSMs. It has a value from the non-negative reals.

> *Aside:* rt contains global real-time. RTC_M makes rt available to M and also provides for timeouts.

2.3 Machine Connections

Two machines M_1 and M_2 may be composed for concurrent execution, provided that $I_1 \cap I_2 = \emptyset$ and $O_1 \cap O_2 = \emptyset$, where I_1, O_1, I_2, and O_2 are the input and output channels of M_1 and M_2, respectively. The channels in $I_1 \cap O_2$ and $O_1 \cap I_2$ are then defined to be *connected*. The composition is represented as $M_1 \parallel M_2$.

> *Aside:* For every machine M we have the composition $M \parallel RTC_M$. The channel RT_M is connected. Machines communicate with each other through connected channels. Note that this is not the traditional definition of "composition"; $M_1 \parallel M_2$ does not define a new CRSM.

The composition is generalized to a set of machines $\{M_1, M_2, \ldots, M_m\}, m \geq 2$. The machines may be composed, provided that $I_i \cap I_j = \emptyset$ and $O_i \cap O_j = \emptyset$, for all $i \neq j$ and $i, j = 1, \ldots, m$, where I_i and O_i are the input and output channels, respectively, for M_i. The channels in $I_i \cap O_j$ and $O_i \cap I_j$, for all $i \neq j$ and $i, j = 1, \ldots, m$, are defined as connected. The composition is denoted $M_1 \parallel M_2 \parallel \ldots \parallel M_m$.

With the exceptions of possible channel connections and the global real-time variable rt, machines are independent. For any two machines M_1 and M_2, neither of which is a real-time clock machine, it is required that $S_1 \cap S_2 = \emptyset$ and $V_1 \cap V_2 = \emptyset$, where $S_1, V_1, S_2,$ andV_2 are states and variables of M_1 and M_2, respectively.

> *Aside:* The variables V of any general machine M are thus *local* to M.

2.4 Closed System of CRSMs

A *closed* system of CRSMs is a set of composed machines $M_1 \parallel M_2 \parallel \ldots \parallel M_m,\ m \geq 2$, such that *all* channels are connected.

> *Aside:* This means that there are no "dangling" channels – every input has a corresponding output, and vice versa. A closed system is meant to completely model both a real-time system and its environment. Alternatively, a closed system could be defined as a pair $< \mathcal{M}, \mathcal{C} >$, where $\mathcal{M}$ is a set of CRSMs and $\mathcal{C}$ is a set of unidirectional channels connecting pairs of machines. This emphasizes the external view of a system which can be characterized solely by its IO activity.

In the following, we will assume that all systems are closed.

3 Execution Histories

The *static* semantics of a general CRSM and of a closed system of CRSMs are defined by describing all their possible execution paths. These paths are also called *histories* or *traces*. We ignore the dynamic effects caused by guards and the timing expressions.

> *Aside:* The result will be many more traces than are possible.

3.1 Trace for a Single CRSM

A *well-formed* execution history for a CRSM $M = (S, I, O, V, G, C, E, T, s_0)$ is defined by a sequence

$$h\ = < s_0,\ (c_1, t_1),\ s_1,\ (c_2, t_2),\ \ldots,\ s_{i-1},\ (c_i, t_i),\ s_i,\ \ldots >$$

satisfying the following for all $i > 0$:

1. $s_i\ \in\ S$.
2. $c_i\ \in\ C$.
3. There exists a transition $\tau\ \in\ T$ such that $\tau = (s_{i-1},\ s_i,\ g,\ c_i,\ e_1,\ e_2)$ for $g\ \in\ G$ and $e_1, e_2\ \in\ E$.
4. $t_i\ =\ t_{i-1} + \delta + t$, where t is finite $(t < \infty)$ and $0\ \leq\ lb(e_1)\ \leq\ t\ \leq\ ub(e_2)\ \leq\ \infty$, with lb and ub being the lower and upper bounds of their expressions, respectively. t_0 is a non-negative real.

Aside: t_i gives the time that command c_i terminates and state s_i is entered. Its range is determined statically by the lower and upper bounds on the expressions. δ is a "small" positive real number and defines a minimum time for all transitions; equivalently, M will spend at least δ time in each state. The use of δ also assures that time progresses forward in an infinite trace, and that a finite number of transitions are executed in a finite amount of time. $t_i - t_{i-1}$ is defined to be finite; thus commands always terminate.

A history can be either finite or infinite. A finite trace has the form:

$$h = < s_0, (c_1, t_1), s_1, \ldots, s_{n-1}, (c_n, t_n), s_n >, \quad n > 0$$

$$h = < s_0 >, \quad n = 0$$

Aside: A finite trace corresponds to a deadlock. In the case here (one machine), it indicates that a state s_n has been reached with no outgoing transitions.

Let $\mathcal{L}(M)$ be the set of all well-formed execution histories for machine M.

Aside: $\mathcal{L}(M)$ is, in general, uncountable since the time intervals specified in $E \times E$ range over the reals.

3.2 Trace for a Closed System

Given a closed system of CRSMs

$$\mathcal{M} = M_1 \parallel M_2 \parallel \ldots \parallel M_m \ , \quad m \geq 2,$$

a *well-formed* execution history for $\mathcal{M}$ is defined by forming a time-ordered interleave of elements, one from each of $\mathcal{L}(M_1), \mathcal{L}(M_2), \ldots, \mathcal{L}(M_m)$. Let $\mathcal{L}(\mathcal{M})$ be the set of all such well-formed traces. Any $h \in \mathcal{L}(\mathcal{M})$ can be written as a concatenation of subsequences from the elements of each $\mathcal{L}(M_i)$:

$$\begin{aligned} h &= x_1 \oplus x_2 \oplus \ldots \oplus x_m \oplus y_1 \oplus y_2 \oplus \ldots \oplus y_m \oplus z_1 \oplus z_2 \oplus \ldots \oplus z_m \oplus \ldots \\ &= < a_0, a_1, \ldots, a_i, \ldots > \ , \end{aligned}$$

where $\oplus$ means sequence concatenation.
h has the properties:

1. $x_i \oplus y_i \oplus z_i \oplus \ldots \in \mathcal{L}(M_i)$ for $i = 1, \ldots, m$.
2. $x_i \neq <>$, the empty sequence. Any of the other subsequences $y_i, z_i, \ldots$ could be empty.

Aside: This allows for finite traces and for all possible interleaves. $x_i \neq <>$ because the start state for every machine exists in every trace.

3. Each a_i is either a machine state s_i or a pair (c_i, t_i).

4. For all $i > 0$ and $j > i$, if $a_i = (c_i, t_i)$ and $a_j = (c_j, t_j)$, then $t_i \leq t_j$.

   Aside: This provides for the time ordering.

5. IO commands are synchronized. Formally, for every $a_i = (c_i, t_i)$,

   (a) if $c_i = ch(v)?$, then there exists an a_j in the trace, $i \neq j$, such that $a_j = (ch(expr)!, t_j)$ and $t_i = t_j$ for the same channel ch.

   (b) if $c_i = ch(expr)!$, then there exists an a_j in the trace, $i \neq j$, such that $a_j = (ch(v)?, t_j)$ and $t_i = t_j$ for the same channel ch.

   $ch(v)?$ and $ch(expr)!$ on the same channel ch are said to be *complementary* IO commands.

   Aside: When it is necessary to identify states and commands with machines explicitly, one can prepend the machine name to them, i.e., $M_k.s_i$ or $(M_k.c_i, t_i)$.

3.3 Projections

A projection of an execution trace h is the sequence obtained by retaining only a subset of the terms of h; alternatively, it is one obtained by removing some subset. In particular, there are at least three projections of interest:

1. Individual Machine History

   We denote by $h|M$ the trace obtained from h by removing all states and (command, time) pairs from h, *except* those of machine M.

   Aside: If $\mathcal{L}(\mathcal{M})$ are the well-formed traces for a closed system containing M and $\widehat{\mathcal{L}}(M)$ is $\{h|M : h \in \mathcal{L}(\mathcal{M})\}$, then $\widehat{\mathcal{L}}(M) \subseteq \mathcal{L}(M)$ because the IO of M is now synchronized.

2. All-Channel Behavior

   The all-channel behavior is the trace obtained from h by removing all states from h and all (command, time) pairs from h, *except* pairs with IO commands.

3. External Behavior

   The *external* behavior of a trace h, denoted $h|\mathcal{IO}$, is obtained by removing the timer channel IO commands from the all-channel behavior of h. A non-empty external projection $h|\mathcal{IO}$ can be written as the sequence:

$$h|\mathcal{IO} = < (c_0, t_0), (c_1, t_1), \ldots, (c_i, t_i), \ldots >,$$

where for all $i \geq 0, t_{2i} = t_{2i+1}, c_{2i}$ and c_{2i+1} are complementary IO commands, and $t_{2i+1} \leq t_{2i+2}$. The $2i$th and $(2i+1)$th elements of $h|\mathcal{IO}$ can be combined into a single element, for all i, to produce a new sequence $b(h|\mathcal{IO})$ that contains only the channel name, message sent, and time:

$$b(h|\mathcal{IO}) \ = < (ch_0, expr_0, t_0), \ \ldots, (ch_i, expr_i, t_i) \ \ldots >,$$

where for all $i \geq 0, \ t_i \leq t_{i+1}, \ ch_i$ is the IO channel in the complementary commands c_{2i} and c_{2i+1} of $h|\mathcal{IO}$, $expr_i$ is the expression in the sender machine, and t_i is the time used in the $2i$th and $(2i+1)$th terms of $h|\mathcal{IO}$. A particular *instance* β of the external behavior represented by $b(h|\mathcal{IO})$ is a sequence obtained by substituting a possible value for each message in b.

$$\beta = < ch_0, v_0, t_0 >, \ \ldots, < ch_i, v_i, t_i >, \ \ldots >,$$

where v_i is a possible value of the message $expr_i$.

> *Aside:* The external behavior is the IO history, excluding the clock. When traces are restricted to dynamic execution paths (next section), the instances β define the external environment behaviors and the system responses (requirements).

4 Operational Semantics

The semantics of a closed system $\mathcal{M}$ of CRSMs is defined by considering the *dynamic* execution paths through the system. These paths are obtained by including the effects of guards which enable or disable transitions, the actual values of the time expressions which are functions of the state variables, and the earliest-transition-first policy for deciding which transition to take. The result is a set of execution traces $\mathcal{L}'(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{M})$. Corresponding to each trace $h \in \mathcal{L}'(\mathcal{M})$ is an external behavior $b(h|\mathcal{IO})$ and an associated instance β.

Our methodology is to describe an algorithm for producing any prefix of a history $h \in \mathcal{L}'(\mathcal{M})$. This simulation algorithm, first outlined informally in [Shaw92], has two forms of non-determinism – that associated with selecting a particular time within a given interval for a computation (internal command) and the arbitrary dealing with ties on the earliest-transition-first policy. $\mathcal{L}'(\mathcal{M})$ is defined implicitly by the set of all possible choices for these nondeterminisms. The algorithm also gives a precise definition of "time" – the values of the t_i in histories and the values that the real-time variable rt take.

4.1 Variables, Events, and Time

The local variables V of any machine M are changed only upon execution of a command. Associated with the execution of a command c in a transition $\tau = (r, s, g, c, e_1, e_2)$ in M are two *events* or markers, the start of the execution, denoted c_s, and the end of the execution, denoted c_e. These events have corresponding occurrence times t_{c_s} and t_{c_e}, respectively. t_{c_e} is also the time that M completes the transition and enters state s; call this time t_s for the particular execution. $t_s \ = \ t_{c_e}$.

Aside: M can be viewed as being in state r until t_{c_s}, then executing the command c taking $t_{c_e} - t_{c_s}$ time, and then entering state s. During execution of c, the state of M is undefined. From Section 3.1, we have $t_{c_s} - t_r \geq \delta$.

The real-time variable rt is updated consistently to be identical with the occurrence times of the c_s and c_e events as commands in $\mathcal{M}$ get executed.

A transition $\tau = (r, s, g, c, e_1, e_2)$ is *enabled* in state r if its guard $g(V)$ is true at time t_r. A command can only be a candidate for execution if its transition is enabled.

Aside: Since elements of V are local (excepting rt) and can only change upon execution of a command, $g(V)$ need only be evaluated once on each entry to state r.

If a transition is enabled, its time interval expressions $e_1(V)$ and $e_2(V)$ may be evaluated, using the values of V at t_r. If $e_1(V) > e_2(V), c$ cannot be selected for execution (a "semantic" error); also, if $c \in C_{internal}$, then c cannot be selected for execution unless $e_2(V) < \infty$.

Let $\tau = (r, s, g, c, e_1, e_2)$ be an enabled transition in machine M. If $c \in C_{internal}$ and c is selected for execution, then $t_{c_e} = t_{c_s} + t$, where $0 \leq e_1(V) \leq t \leq e_2(V) < \infty$ and $t_{c_s} = t_r + \delta$.

Aside: t gives the execution time of c and is assumed to be finite. t can be an arbitrary value in the time interval.

If $c \in C_{io}$ and c is selected for execution, then there exists a partner machine M' also selected for execution with enabled transition $\tau' = (\, r',\ s',\ g',\ c',\ e_1',\ e_2'\,)$ and

1. c and c' are complementary IO commands on the same channel.
2. $t_{c_s} = t_{c'_s} = t_{c_e} = t_{c'_e}$; $t_{c_s} \neq \infty$.

   Aside: IO is instantaneous and occurs at the same (global) time on both machines.

3. $0 \leq e_1(V) \leq t_{c_s} - (t_r + \delta) \leq e_2(V)$

   $0 \leq e_1'(V') \leq t_{c'_s} - (t_{r'} + \delta) \leq e_2'(V')$

4. either $t_{c_s} = t_r + \delta + e_1(V)$ or $t_{c'_s} = t_{r'} + \delta + e_1'(V')$

   Aside: IO occurs at the earliest possible time and is only possible in the time intervals defined by (e_1, e_2) and (e_1', e_2'). Note that e_2 or e_2' could be ∞.

The effect of an executed IO $ch(v)?$ on machine M with $ch(expr)!$ on its partner M', is the instantaneous assignment

$$v \ := \ expr$$

in M and the null statement in M', where v is a local variable of M.

> *Aside:* If M' is M's clock machine, $M' = RTC_M$, then the execution of $c = RT_M(x)?$ on M will produce the assignment $x := rt$, where $rt = t_{c_s}$. (See next section.)

4.2 Simulation Algorithm

The major data structure is an event record which keeps track of the last event that occurred on each machine and is used for the computation of possible next events. An event record for a machine M has three fields: (event_type, transition, time), where event_type may be either *cs* (for *command start*), or *ce* (for *command end*), transition is an element $\tau \ \in \ T$, and time gives the time of the event.

Initially, rt and all elements of V for each machine are initialized, the history trace sequence is initialized with the start state of all machines, and each machine's last event is set to type *ce* with a fictitious starting transition and an initial time. More precisely, we start with the code:

$rt \ := \ t_0$

$h \ := \ < >$

for each M **do**

append $(h, \ M.s_0)$

$M.le \ := \ (ce, \ (*, s_0, *, *, *, *), \ t_0)$

end

where $t_0 \geq 0$, the append function inserts an element at the end of a sequence, le is the last event record, and "$*$" is a "don't care" indicator. When the context is obvious, we will omit the "$M.$" notation.

The algorithm for each step in the simulation is essentially the three-phase one presented in [Shaw 92]. Errors in the time bound expressions, the "semantic" errors mentioned in Sec. 4.1, are not checked for, but this check could be added easily.

Phase 1:

Construct a set of possible next events NEL(M) for each machine M. Each element of NEL(M) is an event record.

for each M **do** NEL(M) := $\emptyset$

for each M **do**

case le. event_type **of**

cs: NEL(M) := $\{(ce, le.\text{transition}, M.t_{c_e})\}$
/* t_{c_e} for this command, an internal one, was computed in Phase 3. */

ce: **for each** $\tau = (s, u, g, c, e_1, e_2)$ **s.t.** le.transition = $(*, s, *, *, *, *)$ and $g(V)$ **do**
/* M has entered state s. The transition τ is enabled. */

case command_type(c) **of**

internal: NEL(M) := NEL(M) $\cup$ $\{(cs, \tau, le.\text{time} + \delta)\}$

io : **if** $(cs, \tau, *) \notin$ NEL(M) **then** /* NEL(M) doesn't have it already. */

for each $M'.\tau' = (s', u', g', c', e_1', e_2')$ **s.t.** ch is the channel in c,
M' is M's IO partner on ch, $M'.le$.event_type = ce,
$M'.le$.transition = $(*, s', *, *, *, *)$,
c' is an IO command on ch, and $g'(V)$ **do**

$t_{c_e} := M.le.\text{time}$

$t_{c_e'} := M'.le.\text{time}$

$t := \max(t_{c_e} + e_1(V),\ t_{c_e'} + e_1'(V))$

if $t \le t_{c_e} + e_2(V)$ and $t \le t_{c_e'} + e_2'(V')$ **then**

NEL(M) := NEL(M) $\cup$ $\{(cs, \tau, t + \delta)\}$

NEL(M') := NEL(M') $\cup$ $\{(cs, \tau', t + \delta)\}$

end

end

end

Phase 2:

Let $\mathrm{EV} = \bigcup_{i=1}^{M} \mathrm{NEL}(M_i)$. Select the set EV_{next} of next events that are to be simulated according to an earliest-event-first policy. $\mathrm{EV}_{next} \subseteq \mathrm{EV}$.

An element (event record) $x \in \mathrm{EV}_{next}$ will have the following properties:

1. x.time $\leq y$.time for all $y \in \mathrm{EV}$.

   Aside: This assures that only the earliest events are in EV_{next}.

2. If $x \in \mathrm{NEL}(M)$ for a machine M, then $\nexists y \in \mathrm{NEL}(M)$ $s.t.$ $y \in \mathrm{EV}_{next}$.

   Aside: At most one event from each machine is in EV_{next}.

3. If x is a command start for an IO command, then $\exists\ y \in \mathrm{EV}_{next}$ $s.t.$ y is the command start for the complementary IO command of x.

   Aside: This assures that both sender and receiver are selected on an IO.

In addition, there are no events $y \in (\mathrm{EV} - \mathrm{EV}_{next})$ $s.t.$ y satisfies the above three properties along with the members of EV_{next}.

Aside: EV_{next} is thus a *maximal* subset of EV satisfying the above properties. EV_{next} is not unique. There may be many maximal subsets, leading to different execution paths.

Below is an algorithm for computing any EV_{next}.

t_{min} := minimum $(x.\text{time})$ /* Compute the earliest-time. */
$x \in \mathrm{EV}$
EV_{next} := $\emptyset$

for each M **do**

NEL(M) := $\{x : x \in \mathrm{NEL}(M),\ x.\text{time} = t_{min}\}$
/* NEL(M) now contains only the earliest-time events. */

while NEL(M) $\neq \emptyset$ **do**

x := select(NEL(M)) /* Pick an element of NEL(M).*/

NEL(M) := NEL(M) $-$ $\{x\}$

$c := x.\text{transition.command}$

if $c \in C_{internal}$ **then**

$\mathrm{EV}_{next} := \mathrm{EV}_{next} \cup \{M.x\}$

NEL(M) := $\emptyset$

else /* $c \in C_{io}$ Let ch be c's channel and M' be c's IO partner for channel ch. */

if $\exists\ x' \in \mathrm{NEL}(M')$ **s.t.** $x'.\text{transition.command} = c' \in C_{io}, ch$ is c''s channel, and $x.\text{time} = t_{min}$ **then**

$\mathrm{EV}_{next} := \mathrm{EV}_{next} \cup \{M.x, M'.x'\}$

NEL(M) := NEL(M') := $\emptyset$

end
end

Phase 3:

Perform the simulation step leading to the events in EV_{next}. This involves the possible execution of a command and updating of the local variables of a machine, inserting new values in some last event records, updating rt, and appending the appropriate elements to the execution trace h.

$rt := t_{min}$

Aside: t_{min} is thus the new time reported by the clock machine.

while $\text{EV}_{next} \neq \emptyset$ **do**

$x :=$ remove(EV_{next}) /* Remove an arbitrary event record from EV_{next}. */
/* Let $x = M.(a,\ (*, s, *, c, e_1, e_2),\ *)$ */

if $c \in C_{internal}$ **then**

case a **of**

cs : $le := (cs,\ x.\text{transition},\ rt)$

$M.t_{c_e} := rt +$ choose_time($e_1(V), e_2(V)$) /* Compute t_{c_e} for Phase 1. */
/* e_1 and e_2 are evaluated and a time duration is selected in their range. */

ce : $le := (ce, x.\text{transition}, rt)$

execute_program(c) /* This changes the data state V in general. */

append($h, (M.c, rt)$)

append($h, M.s$)

else /* $c \in C_{io}$. Let $M'.(*, (*, s', *, c', *, *), *) = x' \in \text{EV}_{next}$ be the IO partner event of x. */

$M.le := (ce, x.\text{transition}, rt)$

$M'.le := (ce, x'.\text{transition}, rt)$

$\text{EV}_{next} := \text{EV}_{next} - \{x'\}$

execute_io(c, c') /* Perform IO assignment on target machine. */

append($h,\ (M.c, rt)$)

append($h,\ (M'.c', rt)$)

append($h,\ M.s$)

append($h,\ M'.s'$)

end

Phases 1, 2, and 3 are repeated either forever or until EV= $\emptyset$ after Phase 1, to produce an infinite or finite trace, respectively. All possible prefixes could be obtained in principle by trying all possible orderings of the M_i in the global "**for each** M" statement in Phase 2, by trying all possible orders in the "select(NEL(M))" statement in Phase 2, and by trying all "possible" reals in the interval $(e_1(V), e_2(V))$ in Phase 3.

> *Aside:* The technique mentioned above for obtaining all possible prefixes can, in general, produce repeats of the same prefix or trace. The Phase 3 algorithm (and initialization) can be modified easily to generate any of the projections described in Section 3, e.g. $h|\mathcal{IO}$, the combination $b(h|\mathcal{IO})$, or its instantiation β.

5 Conclusions

Our (more) formal semantics has clarified several ideas that appeared in the original CRSM paper. One is the notion of a closed system with all channels connected. A second is the definition of execution traces and their relation to time. A third is the meaning of real-time, involving both the execution times for transitions and the global real-time accessible through the clock machines.

Acknowledgement

Thanks to Sitaram Raju for his careful reading and comments.

References

[Hoare 85] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, NJ, 1985.

[Raju 93] S. Raju, "An Automatic Verification Technique for Communicating Real-Time State Machines," TR #93-04-08, Dept. of Computer Science & Engineering, University of Washington, Seattle, April 1993.

[Raju & Shaw 92] S. Raju and A. Shaw, "A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines," TR #92-10-03, Dept. of Computer Science & Engineering, University of Washington, Seattle, October 1992. A revised version is in publication in the journal *Software-Practice & Experience*.

[Shaw 92] A. Shaw, "Communicating Real-Time State Machines," *IEEE Trans. on Software Eng.*, Vol. 18, No. 9, September 1992, pp. 805-816.