Specification of Hybrid Systems in CRP

R.K. Shyamasundar*
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay 400 005, India
e-mail: shyam@turing.tifr.res.in

Abstract

In this paper, we show that the paradigm of *Communicating Reactive Processes* (CRP) unifying asynchronous and perfectly synchronous mechanisms can be naturally extended for the specification of *continuous* computations and provide a convenient formalism for specifying hybrid systems. Further, it can be used for specifying asynchronous systems operating in dense real-time domains.

1 Introduction

Programming languages based on the *perfect synchrony paradigm* have proven useful for programming reactive systems. Perfect synchrony paradigm essentially states that the considered reactive programs respond in no time and produce their outputs synchronously with their input. One of the main reasons for its success is that it permits the programmer to focus on the logic of reactions and makes it possible to use several automata-based verification systems for correctness proofs. Further, the correctness proofs of programs follow their implementation very closely and hence, are more robust and reliable. However, the application is limited to clocked systems. If we look at complex reactive system specifications such as process control systems and robotic applications, the need for features such as (i) asynchronous events (events that can happen arbitrarily close to each other), (ii) integration of discrete and continuous components (continuous components may cause continuous change in the values of some state variables according to some physical law), and (iii) explicit clock times, become apparent. In [4], a paradigm referred to as *Communicating Reactive Processes* (CRP) that provides a unification of perfect synchrony and asynchrony has been presented. In this paper, we discuss an extension of CRP, referred to as *Timed* CRP, that

- models *continuous* computations and thus, provides a convenient formalism for specifying hybrid systems, and
- models asynchronous systems operating in dense real-time domains.

Consider specifying hybrid systems that combine discrete and continuous components possibly with the need to reference clocks explicitly. For consistency, it is necessary to have a consistent assumption about the progress of the computation as the system evolves. In the timed CRP, it has been possible to provide

*The work was partially supported by IFCPAR (Indo-French Centre for the promotion of Advanced Research), New Delhi, India.

such a consistent assumption through the clocked semantics of ESTEREL and the interpretation of the clocks in terms of the `exec` primitive. One of the interesting features is that hybrid systems without explicit references to clocks can be specified by a subset of timed CRP consisting of just the statements of ESTEREL and the `exec` primitive. The full version of timed CRP can specify dense asynchronous systems with explicit references to clocks.

2 Hybrid Systems

Hybrid systems are systems that combine discrete and continuous computations. Hybrid system model contains activities that modify their variables continuously over intervals of positive duration in addition to the familiar transitions that change the values of variables instantaneously, representing the discrete components. Many systems that interact with a physical environment such as a digital module controlling a process or a manufacturing plant, a digital-analog guidance of transport systems, control of a robot, flexible manufacturing systems etc., can benefit from the study of hybrid models. In the sequel, we use the word *asynchronous* to mean that initiation and completion cannot happen simultaneously; in other words, the initiation and completion of an asynchronous activity are observable. Various abstract models for systems for handling real-time and *continuous* computations have been proposed recently in [6,7,8]. An extension of Statecharts for the specification of hybrid systems has been described in [6]. In this paper, we broadly use the definitions of hybrid systems as defined in [6,7] and the main characteristics of the hybrid model is briefed below.

A *continuous or a hybrid* component is defined as an *asynchronous activity* characterized by $(\mathcal{V}, \mathcal{A}, \Theta, \mathcal{F}, \mathcal{E})$ where:

- $\mathcal{V}$ is the set of *continuous*, V_c and *discrete*, V_d, variables that are modified by continuous and discrete activities respectively.
- $\mathcal{A}$ is a finite set of activities: Each activity, $\alpha \in \mathcal{A}$ is a conditional differential equation of the form, $p \rightarrow \dot{x} = e$, where p is a predicate over V_d called the activation condition of α, $x \in V_c$ is a continuous state variable and e is an expression over V. We say that the activity α governs variable x. Activity α is said to be *active* in state s if its activation condition p holds in s; otherwise, α is said to be *passive*. It is required that the activation conditions of the activities that govern the same variable x be exhaustive and exclusive, i.e., exactly one of them holds on any state.
- $\mathcal{E}$ is a set of important events: These are all the observable discrete events relative to the finite set of activities.
- Θ defines the initial condition.
- $\mathcal{F}$ is the set of differential equations satisfying the conditions given in [7].

3 Timed CRP

A CRP program consists of a network $M_1 \; // \; M_2 \; \cdots \; // \; M_n$ of ESTEREL reactive programs or *nodes*, each having its own input/output reactive signals and its own notion of an instant. The network is asynchronous and node M_i is locally reactively driving a part of a complex network which is handled globally by the network. Asynchronous communication between nodes is achieved through the `exec` primitive and `channel` declarations. The informal and formal semantics of some of the important constructs of the language are given below.

3.1 Informal Semantics

3.1.1 Basic ESTEREL

The execution of an ESTEREL program associates a sequence of output events with a sequence of input events. The program repeatedly receives an input event E_i from its environment and reacts by building an output event E_i'. The events E_i and E_i' are synchronous in the sense that the external observer observes the input/output event as a single event $E_i \cup E_i'$. The production of an output event from an input event is called a *reaction*. The flow of time is defined by the sequence of reactions and hence,*reaction* is also termed an *instant*. Note that at any instant there is at least one input signal. A signal is present at an instant if and only if it is received as input from the environment or emitted by the program itself at that instant. At each instant, each input or local signal is consistently seen as present or absent by all statements - thus, ensuring determinism. By default, signals are absent. The notion of an instant leads to consistent definitions of temporal expressions.

In order to provide the ability to quantify the progress of computation as the system evolves, we assume that every input event or instant consists of a special clock signal of type *real* referred to as `tick`.

The list ESTEREL kernel statements is given below:

```
nothing
halt
emit S
stat1; stat2
loop stat end
present S then stat1 else stat2 end
do stat watching S
stat1 || stat2
trap T in stat end
exit T
signal S in stat end
```

The kernel statements are imperative in nature, and most of them are classical in appearance. Instantaneous control transmission appears everywhere. The `nothing` statement is purely transparent: it terminates immediately when started. An `emit S` statement is instantaneous: it broadcasts `S` and terminates right away, making the emission of `S` transient. In `emit S1; emit S2`, the signals `S1` and `S2` are emitted simultaneously. In a signal-presence test such as `present S ...`, the presence of `S` is tested for right away and the `then` or `else` branch is immediately started accordingly. In a `loop` *stat* `end` statement, the body *stat* starts immediately when the loop statement starts, and whenever *stat* terminates it is instantaneously restarted afresh; to avoid infinite instantaneous looping, it is required that the body of a loop should not terminate instantaneously when started.

The `watching` and `trap-exit` statements deal with behavior preemption – the most important feature of ESTEREL. In the watchdog statement `do` *stat* `watching S`, the statement *stat* is executed normally up to proper termination or up to future occurrence of the signal `S`, which is called the guard. If *stat* terminates strictly before `S` occurs, so does the whole `watching` statement; then the guard has no action; otherwise, the occurrence of `S` provokes immediate preemption of the body *stat* leading to immediate termination of the whole watching statement. Note that nesting watching statements provides for priorities. For details on `trap`, the reader is referred to [3].

3.1.2 Asynchronous Tasks

In a very general way, asynchronous tasks are those tasks which do take time; that is, the time between initiation and completion is observable. In the terminology of ESTEREL , this can be interpreted to mean that there will be at least one instant between initiation and completion. The `exec` primitive provides the interface between ESTEREL and asynchronous tasks.

Task Declaration

An asynchronous task is declared as follows:

```
task task_id (par_lst) [(CONT: DE; var_lst)] return signal_nm (type);
```

where

- `par_lst` gives the list of *discrete* parameters (reference or value);
- the signal returned by the task is given by the `signal_nm` with its type after the keyword `return`; it is possible to have multiple return signals.
- the optional phrase `[CONT:DE ; var_lst]` is used if the underlying task has a *continuous* component. Here, `DE` denotes the underlying differential equation and `var_lst` denotes the list of *continuous* variables. If the differential equation of the form $\dot{x} = e$ satisfies the conditions stated earlier, then the value of the continuous variables can be computed at various instants (e.g., linear hybrid systems).

A typical example of the declaration of a hybrid component is given by:

`task MOUSE (y) (CONT:` $\dot{X}_m = -V_m$; x_m`) return Mouse ({sanct,kill})`

Task Instantiation

Instantiation of the task is done through the primitive `exec`. For example, the above task can be instantiated from an ESTEREL program as follows:

`exec MOUSE (y)` (x_m)

The execution of the above statement starts task `MOUSE` and waits for its termination; of course, the number and type of arguments, and the return signal type should match with the task declared. In other words, `exec` requests the environment to start the task and then waits for the return signal (which also indicates the termination of the task).

Since there can be several occurrences of `exec T` in a module for the same task `T`, several simultaneously active tasks having the same name can coexist. To avoid confusion among them one can assign an explicit label to each `exec` statement; a labeled `exec` statement is of the form `exec L:T`. The label name must be distinct from all other labels and input signal names. An implicit distinct label is given to `exec` statements.

The primitive `exec` provides an interface between ESTEREL and the asynchronous environment can be seen by the following interpretation. Given an `exec` statement labeled L, the asynchronous task execution is controlled from ESTEREL by three implicit signals `sL` ((output signal), `L` (input signal), and `kL` (output signal) corresponding to starting the task, completion and killing the execution of the task respectively. The output *start* signal `sL` is sent to the environment when the `exec` statement starts. It requests the start of an asynchronous incarnation of the task. The input *return signal* `L`, is sent by the environment when the task incarnation is terminated; it provokes instantaneous termination of the `exec` statement. the output *kill signal* `kL` is emitted by ESTEREL if the `exec` statement is preempted before termination, either by an enclosing watching statement or by concurrent exit from an enclosing trap. The return signals corresponding to the `exec` label can be used for declaring incompatibility with other input signals (this becomes handy in declaring channels).

3.1.3 Interpretation of a Global Clock in terms of `exec`

Consider a task CLOCK which accesses a global clock and sends the alarm signals at the times requested. Declaration of the task CLOCK in timed CRP takes the form:

```
task CLOCK(d) return alarm ({real})
```

Now, an alarm after ℓ units can be instantiated by

```
exec CLOCK(ℓ)
```

Since the `exec` statement is not instantaneous, it follows that $\ell > 0$. The semantics of `exec` permits the use of multiple instantiations on the same and different nodes. This feature makes it possible to use CLOCK as a global clock for the different components.

3.1.4 Timed Statements

The primitive timed statements of Timed CRP are given below where `tick` denotes the clock signal.

a. *await* with *ticks*:
   `await tick(d)`: This statement delays the next reaction till the instant $t + d$ is crossed where t is the value of CLOCK at the current instant. In other words, it corresponds to a delay of d. `await tick(0)` is not valid as it instantaneously terminates. In ESTEREL `await tick` corresponds to awaiting for the next input signal or the instant.

b. *watching with ticks*:
   `do` *stat* `watching tick(d)`: This statement gives the *time limit* to the execution of its body *stat*. Let us assume $d > 0$. In this case, the body starts as soon as the `watching` statement starts. If the body terminates or exits a trap strictly before a timeout of d from the current instant, so does the `watching` statement; otherwise the `watching` statement terminates as soon as the value of `tick` signal reaches $t + d$. As the `watching` statement

is active at the current instant, $d = 0$ corresponds to the instantaneous termination of the body without being executed.

Restrictions: As `tick` is always present, the following restrictions are placed:

1. It is not permitted to use `tick` in expressions of `await Immediate` and `watch immediate` statement constructs.
2. The expressions in the `present` statement constructs are not allowed to reference clocks explicitly.

3.1.5 Using Timed Statements

1. *Specifying Time Bounds*
   (a) `await tick(ℓ)` specifies a delay of ℓ.
   (b) Specifying timeout (maximum duration) of `u` can be done by:
   `do S watching tick(u) end`
   (c) Enforcing a time interval of $[\ell, u]$ for producing a *good_part* in the usual producer-consumer problem (cf. [6]) is given below:

```
do
  [exec good_part || await tick(ℓ)]
  watching tick(u)
end
```

2. Actual time for the transition/task with a timeout of `d` is specified below:

```
x:=0;
do
 exec T; (* Let us assume that rT is the return signal *)
watching tick(d)
 emit abort ; (* aborted due to timeout *)
end
present rT then x := ?tick
```

   In the above program, `x` would have the actual value of time taken on proper termination; if aborted due to timeout then x will have value zero and also signal `abort` will be emitted.

3. Specifying time for multiple rendezvous and interrupt servicing can be done in the same manner.

3.1.6 Illustrative Example

Let us consider a variant of the Cat and Mouse problem described in [6] with the addition of a reward. The addition reflects the use of continuous variables during the computation. The variant of the problem is defined below.

At time $t = 0$, a mouse starts running from a certain position on the floor in a straight line towards a hole in the wall, which is at a distance X_0 from the initial position. The mouse runs at a constant velocity V_m. After Δ time units, a cat is released at the same initial position and chases the mouse at velocity V_c along the same path. The competition is as follows:

- CAT: It has to catch the MOUSE before it reaches the hole; reward is the MOUSE itself.
- MOUSE: It has to find a sanctuary in the hole before the CAT catches; it gets a reward of FOOD proportional to the distance the CAT did not cover.

Program in timed CRP is given below: (EXP denotes some pure function):

```
module WATCHER;
  task MOUSE (x) (CONT: Ẋ = −V) return MOUSE()
  task CAT (y) (CONT: Ẏ = −V) return CAT()
   loop
   do
     do
       [   exec MOUSE (x)
              ||
          await tick(Δ);
          exec CAT (y)
       ]
     watching  CAT
     timeout emit CAT_Wins;
     end;
    watching MOUSE
    timeout  emit MOUSE_WINS (FOOD(EXP(?tick,y))
    end;
    end
end module
```

In the programs there is a priority for the MOUSE; that is, if both reach the destination at the same time, MOUSE is declared the winner.

3.1.7 Formal Semantics

Given a module M and an input event I, the behavioral semantics determines a transition $M \underset{I}{\overset{O}{\longrightarrow}} M'$ where O is the generated output event and M' is another module suited to perform further reactions. The derivative M' has the same interface as M and differs only by its body. The reaction to a sequence of input events is computed by chaining such elementary transitions. The transition relation is defined by using an auxiliary inductive relation

$$< stat, \rho_C > \xrightarrow[E \cup \{\texttt{tick}\}]{E', L.k} < stat', \rho'_{C'} >$$

on statements where

1. The old and the new signal environments are defined by,
   - E is the current event to which $stat$ reacts; the return signal (input signal) of `exec` appears in E.
   - E' is the event emitted by $stat$ in response to E; the start and kill signals (output signals) of `exec`'s appear in E'.
   - L is the set of labels of the `exec` statements currently active in $stat$;
   - The integer *termination level* k (cf. [3]) determines how control is handled. In each reaction, any statement can behave in three ways: it can terminate and release the control, it can retain the control and wait for further events, or it can exit a trap. We set $k = 0$ for proper termination, and $k = 1$ for waiting; other values correspond to trap-statements.
   - Since signals are broadcast, $stat$ receives the signals it emits and E' will always be contained in E.
2. The old and the new states are defined by,

- ρ_C is the memory environment with C denoting the set of continuous variables corresponding to the active continuous tasks at the beginning of the transition.
- $\rho'_{C'}$ is the memory environment with C' denoting the value of continuous variables immediately at the end of the transition.
- For each $x \in C$, x is assigned a value given by the differential equation (say, $\dot{X} = exp$).

The relation between the two transition systems is given by $M \xrightarrow[I]{O} M'$ iff $< stat, \rho > \xrightarrow[I \cup O\{tick\}]{O,L.k} < stat', \rho' >$ where $stat$ and $stat'$ are the bodies and ρ and ρ' are the states of M and M' respectively; we assume the harmless restriction that $stat$ cannot internally emit input signals.

With respect to the set $\mathcal{L}$ of active `exec` labels, we use the notation,
$E\#L = E \cup \{kL | L \in \mathcal{L} \text{ and } sL \notin E\} \; - \; \{sL | L \in \mathcal{L} \text{ and } sL \in E\}$
to denote the set obtained from E by killing `exec` 's started before the current instant and ignoring the one's started at the current instant.

Transition Rules for Constructs without explicit time

For lack of space, we describe only the transition rules for `exec`, `watching` and `present` statements ignoring the state component.

The rule (*exec_start*) is used to start an `exec` statement. A distinct label is used for keeping track of several instances and the statement is rewritten into an auxiliary `exec_wait` statement which has been introduced for the sake of convenience. The rule (*exec_wait1*) expresses the termination of the asynchronous task with the receipt of the return signal from the task. The (*exec_wait2*) rule reflects the waiting for the arrival of the return signal from the task.

(exec_start)

$$\texttt{exec } L : P \xrightarrow[E]{sL,\{L\},1} \texttt{exec\_wait L : P}$$

(exec_wait1)

$$\frac{L \in E,\ P \in TASK}{\texttt{exec\_wait } L : P \xrightarrow[E]{\emptyset,\emptyset,0} \texttt{nothing}}$$

(exec_wait2)

$$\frac{L \notin E,\ P \in TASK}{\texttt{exec\_wait } L : P \xrightarrow[E]{\emptyset,\{L\},1} \texttt{exec\_wait } L : P}$$

The rule for `watching` statement is given using a `present` statement that will behave as the required guard at future instants. For remembering the particular `exec`'s that should be killed if preemption occurs because of the presence of the signal `S` at some future instant, the `present` statement is decorated with the set of currently active exec-labels `L`. It can be seen that the set `L` is used in rule *present1* to generate the appropriate kill signals when the then-branch is taken (that is at preemption time).

(watching)

$$\frac{stat \xrightarrow[E]{E',L,k} stat'}{stat\ \texttt{watching}\ S \xrightarrow[E]{E',L,k} \texttt{present}_L\ S\ \texttt{else}\ stat'\ \texttt{watching}\ S}$$

(present1)

$$\frac{S \in E, \quad stat_1 \xrightarrow[E]{E'_1,L_1,k_1} stat'_1}{\texttt{present}_L\ S\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \xrightarrow[E]{E'_1\#L,L_1,k_1} stat'_1}$$

(present2)

$$\frac{S \notin E, \quad stat_2 \xrightarrow[E]{E'_2,L_2,k_2} stat'_2}{\texttt{present}_L\ S\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \xrightarrow[E]{E'_2,L_2,k_2} stat'_2}$$

Transitions Rules for Constructs with Time

Let $\texttt{tick}(\delta_+)$ denote the incremental increase from the previous reading of the signal **tick**. Then the basic rules using time are given below:

(watching_t1)

$$\frac{stat \xrightarrow[E\cup\{\texttt{tick}(\delta_+)\}]{E',\{L\},k} stat',\ d > 0,\ d-\delta\ > 0}{\texttt{do}\ stat\ \ \texttt{watching}\ \ \texttt{tick}(d) \xrightarrow[E\cup\{\texttt{tick}(\delta_+)\}]{E',L,k} \texttt{do}\ \ stat'\ \ \texttt{watching}\ tick(d-\delta)}$$

(watching_t2)

$$\frac{stat \xrightarrow[E\cup\{\texttt{tick}(\delta_+)\}]{E',\{L\},k} stat',\ d > 0,\ d-\delta\ = 0}{\texttt{do}\ stat\ \ \texttt{watching}\ \ tick(d) \xrightarrow[E\cup\{\texttt{tick}(\delta_+)\}]{E'\#L,\emptyset,k} \texttt{nothing}}$$

The rule (*watching_t1*) reflects the perceiving of the advancement of the clock by the task whenever an event happened in the system. Every task perceives the same time. The time/alarm at d units from the point of watching becomes an input signal.

Rule (`watching_t2`) corresponds to timeout d. At this instant, the statement is preempted along with the asynchronous tasks L it is awaiting for.
Note: In the full ESTEREL notation (cf. [3]) $s^+(v1)$ is used to denote the value, $v1$, for signal, s, at the present instant and $s^-(v2)$ to denote the value in the previous instant. Using this notation, the expression $d - \delta > 0$ in the premise of (*watching_t1*), corresponds to $(d - |\mathtt{tick}^+(v_1)| - |\mathtt{tick}^-(v_2)|) > 0$; in other words, $v_2 - v_1 = \delta$.

Continuous Components

Let `exec` $L : \dot{Y} = e$ denote an asynchronous task denoting a continuous component where L is the label associated with the task and Y_L denotes the continuous variables associated with the task. In the following, rules for the basic transitions are given showing the changes to the continuous variables explicitly. In rule `exec_H_start`, the expression $\rho[Y_L \leftarrow \theta_L]$ denotes the assignment to the continuous variables when the task is initiated; signal environments remaining the same as given earlier. In rules `exec_H_wait1`, and `exec_H_wait2`, the expression $\rho[Y_L \leftarrow e(Y_L, \delta)]$ denotes updating of the continuous variables after the time advances by δ as per the differential equation given. In case, the continuous component is based on handshake communications, we need to have an additional rule to keep track of the instant at which the actual handshake/rendezvous starts (such a rule is not necessary in pure CRP).

(exec_H_start)

$$< \mathtt{exec}\ L : \dot{Y} = e,\ \rho > \xrightarrow[E]{sL, \{L\}, 1} < \mathtt{exec\_wait}\ L : \dot{Y} = e,\ \rho[Y_L \leftarrow \theta_{L_0}] >$$

(exec_H_wait1)

$$\frac{L \in E,\ P \in TASK}{< \mathtt{exec\_wait}\ \ L : \dot{Y} = e,\ \rho > \xrightarrow[E \cup \mathtt{tick}(\delta_+)]{\emptyset, \emptyset, 0}} \quad < \mathtt{nothing}, \rho[Y_L \leftarrow e(Y_L, \delta)] >$$

(exec_H_wait2)

$$\frac{L \notin E,\ P \in TASK}{< \mathtt{exec\_wait}\ L : \dot{Y} = e,\ \rho > \xrightarrow[E \cup tick(\delta_+)]{\emptyset, \{L\}, 1}} \quad < \mathtt{exec\_wait}\ L : \dot{Y} = e,\ \rho[Y_L \leftarrow e(Y_L, \delta)] >$$

4 Behaviour of Timed CRP Programs

In this section, we illustrate that a timed CRP defines a hybrid transition system [6] and show the relation to other models. For lack of space, we only state the propositions with an informal sketch of the ideas without proof.

Definition 1: A *history* of a CRP node is a sequence of events $E_1, \cdots, E_n, \cdots$; for convenience, we denote E_i by $I_i.O_i$ where I_i and O_i denote the input and the output events in the ith instant respectively. In the *timed CRP* node every instant consists of the input signal `tick`.

Definition 2: A *history* is said to be *timed CRP valid* if it satisfies the following properties:

1. The history satisfies all the declared exclusion relations.
2. $\forall i,$ `tick` $\in I_i$, every input instant contains the special signal `tick` with values. For instance, the signal `tick(v)` in I_n indicates an elapse of `v` $\in \mathcal{R}$ units of time from the last occurrence of `tick`.
3. Signals from asynchronous/hybrid components satisfy:
   - An event is received only after requested (which has not been killed).
   - The start- and receive- of an asynchronous request cannot happen in the same instant.
4. $\forall i,\ |I_i| + |O_i| < \infty$. That is, it satisfies the property of *finite variability*, namely, the state changes only finitely often throughout any finite interval of time. That is, between any two consecutive input instants containing `tick` there can be only a finite sequence of events.

Definition 3: A progressive time sequence is an infinite sequence of time elements $\theta : t_0, t_1, \cdots$, where $t_i \in T$, for each $i = 0, 1, \cdots$ satisfying

- $t_0 = 0$
- Time does not decrease. That is, for every $i \geq 0, t_i \leq t_{i+1}$.
- Time eventually increases beyond any bound. That is, for every time element $t \in T, t_i > t$ for some $i \geq 0$; this ensures the non-Zeno requirement.

Theorem 1: The history of timed CRP valid programs forms a progressive time sequence.
Proof: The proof is based on considering the sets of sequences of accumulated `tick`'s at each instant (i.e., the global time at each instant) and show that each sequence is progressive and hence, non-Zeno.

Using properties of the timed CRP execution sequences (history) such as confluency, relation between halted execution sequences and causal freeness etc., we establish the relationship of timed CRP with other models such as *timed transition systems*, *hybrid traces* and *timed graphs* (cf. [1,6,7,8]).

Theorem 2: The timed CRP defines a hybrid transition system [6].
Proof: The proof is based on showing that time and discrete variables are not changed at the same time and time progresses synchronously on all processes.

Theorem 3: *Causally correct* timed CRP programs defines *timed graphs*.

5 Conclusions

We have described timed extensions of the CRP and shown that it can be conveniently used for specifying asynchronous timed systems and hybrid systems. Additionally, it can be used to specify and check the continuous presence of signals for a given duration [5]. One of the distinct features is that hybrid systems that do not reference clocks explicitly can be specified using timed CRP without the timed statements. Further, it is possible to relax the conditions on $\mathcal{F}$ used in the hybrid model, if there is no need to compute the values of continuous variables in other components.

As mentioned already, [6] describe an extension of Statecharts for specifying hybrid systems; in their language extension, they achieve the power through the addition of features such as `watching` on variables, channel variables and preemption constructs to Statecharts. In timed CRP the availability of the CLOCK as an asynchronous task avoids the need of watching variables – thus, conforming with the ESTEREL philosophy. We feel that the availability of asynchronous tasks and rendezvous communications makes it convenient to specify classes of hybrid systems naturally, even those requiring multiple rendezvous. Many implementations for the asynchronous tasks are possible leading to natural specifications for hybrid systems. Currently, we are investigating the possible protocols that would be best suited for a variety of applications. From the point of view of specification of hybrid systems, it is not clear whether there is a need to use the power of specifying nondeterminism of the network. A study in this direction is being taken as a positive answer would enable the adaptation of verification tools.

Acknowledgment

The work on synchronous programming has been due to the inspiration of Gèrard Berry. It is a pleasure to thank Amir Pnueli whose lectures on hybrid systems at TIFR clarified various subtle aspects of hybrid system specification and F. Boussionot and S. Ramesh for many valuable discussions.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, *Model Checking for Real-time Systems*, 5th IEEE LICS, pp. 414-425, June 1990.

[2] G. Berry (1992), *A hardware implementation of pure Esterel*, *Sadhana*: Academy Proceedings in Engineering Sciences, Indian Academy of Sciences, Special Issue on Real Time edited by RK Shyamasundar, 17 (1):95-139, 1992.

[3] G. Berry and G. Gonthier (1988), *The Esterel synchronous programming language: Design, semantics, Implementation*, Rapport de Recherche 842, INRIA 1988, Science of Computer Programming, Vol. 19, No.2, Nov. 92, pp. 87-152.

[4] G. Berry, S. Ramesh and R.K. Shyamasundar, *Communicating Reactive Processes*, 20th ACM Symposium on Principles of Programming Languages, South Carolina, Jan. 1993, pp. 85-99.

[5] L.Y. Liu and R.K. Shyamasundar, *RT-CDL: A real time design language and its semantics*, IFIP 89, North-Holland Publishing Co., pp. 21-26.

[6] Y. Kesten and A. Pnueli, *Timed and hybrid Statecharts and their textual presentation*, LNCS 571, pp. 591-619, 1992.

[7] Z. Manna and A. Pnueli, *Models for Reactivity*, to appear in Acta Informatica, 1993 (an earlier version presented at the 25th Anniversary of INRIA).

[8] X. Nicollin, J. Sifakis, and S. Yovine, *From ATP to timed graphs and hybrid systems*, LNCS 600, pp. 549-572, 1992.